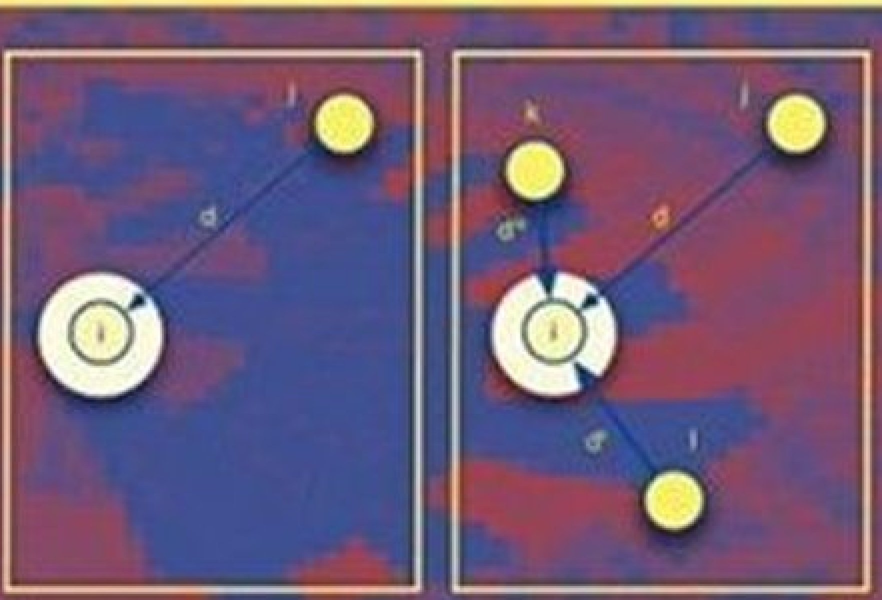
State-of-the-Art Survey

LNCS 5454

Michael Butler
Cliff Jones
Alexander Romanovsky
Elena Troubitsyna (Eds.)

Methods, Models and Tools for Fault Tolerance



Springer

Lecture Notes in Computer Science 5454

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison
Lancaster University, UK

Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler
University of Surrey, Guildford, UK

Jon M. Kleinberg
Cornell University, Ithaca, NY, USA

Alfred Kobsa
University of California, Irvine, CA, USA

Friedemann Mattern
ETH Zurich, Switzerland

John C. Mitchell
Stanford University, CA, USA

Moni Naor
Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz
University of Bern, Switzerland

C. Pandu Rangan
Indian Institute of Technology, Madras, India

Bernhard Steffen
University of Dortmund, Germany

Madhu Sudan
Massachusetts Institute of Technology, MA, USA

Demetri Terzopoulos
University of California, Los Angeles, CA, USA

Doug Tygar
University of California, Berkeley, CA, USA

Gerhard Weikum
Max-Planck Institute of Computer Science, Saarbruecken, Germany

Michael Butler
Cliff Jones
Alexander Romanovsky
Elena Troubitsyna (Eds.)

Methods, Models and Tools for Fault Tolerance

Springer

Volume Editors

Michael Butler
University of Southampton
School of Electronics and Computer Science
Highfield, Southampton, SO17 1BJ, UK
E-mail: mjb@ecs.soton.ac.uk

Cliff Jones
Newcastle University, School of Computing Science
Newcastle upon Tyne, NE1 7RU, UK
E-mail: cliff.jones@ncl.ac.uk

Alexander Romanovsky
Newcastle University, School of Computing Science
Newcastle upon Tyne, NE1 7RU, UK
E-mail: alexander.romanovsky@ncl.ac.uk

Elena Troubitsyna
Åbo Akademi University, Department of Computer Science
Lemminkaisenkatu 14 A, 20520 Turku, Finland
E-mail: Elena.Troubitsyna@abo.fi

Library of Congress Control Number: Applied for

CR Subject Classification (1998): D.4.5, C.2.4, D.1.3, D.2, F.2.1-2, D.3, F.3

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-642-00866-6 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-00866-5 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

springer.com

© Springer-Verlag Berlin Heidelberg 2009

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12632466 06/3180 5 4 3 2 1 0

Preface

The growing complexity of modern software systems increases the difficulty of ensuring the overall dependability of software-intensive systems. Complexity of environments, in which systems operate, high dependability requirements that systems have to meet, as well as the complexity of infrastructures on which they rely make system design a true engineering challenge.

Mastering system complexity requires design techniques that support clear thinking and rigorous validation and verification. Formal design methods help to achieve this. Coping with complexity also requires architectures that are tolerant of faults and of unpredictable changes in environment. This issue can be addressed by fault-tolerant design techniques. Therefore, there is a clear need of methods enabling rigorous modelling and development of complex fault-tolerant systems.

This book addresses such acute issues in developing fault-tolerant systems as:

- Verification and refinement of fault-tolerant systems
- Integrated approaches to developing fault-tolerant systems
- Formal foundations for error detection, error recovery, exception and fault handling
- Abstractions, styles and patterns for rigorous development of fault tolerance
- Fault-tolerant software architectures
- Development and application of tools supporting rigorous design of dependable systems
- Integrated platforms for developing dependable systems
- Rigorous approaches to specification and design of fault tolerance in novel computing systems

The editors of this book were involved in the EU (FP-6) project RODIN (Rigorous Open Development Environment for Complex Systems), which brought together researchers from the fault tolerance and formal methods communities. In 2007 RODIN organized the MeMoT workshop[1] held in conjunction with the Integrated Formal Methods 2007 Conference at Oxford University. The aim of this workshop was to bring together researchers who were interested in the application of rigorous design techniques to the development of fault-tolerant software-intensive systems.

We proposed to the authors of the best workshop papers to expand their work and a number of well-established researchers working in the area to write invited chapters. This book contains the refereed and revised papers that came

[1] The proceedings of the Workshop on Methods, Models and Tools for Fault Tolerance are at http://rodin.cs.ncl.ac.uk/deliverables.htm

in response. Twelve of the papers are reworked from the workshop; three papers are invited.

The editors would like to thank the reviewers: Elisabeth Ball, Jeremy Bryans, Joey Coleman, Alan Fekete, Michael Fisher, John Fitzgerald, Michael Harrison, Alexei Iliasov, Michael Jackson, Linas Laibinis, Qaisar Ahmad Malik, Annabelle McIver, Larissa Meinicke, Luc Moreau, Luigia Petre, Martha Plaska, Mike Poppleton, Brian Randell, Colin Snook and Divakar Yadav.

We would particularly like to thank Louise Talbot, who has efficiently handled the collation of this book.

Both in organizing MeMoT 2007 and in publishing this edited book, we are aiming to build a network of researchers from the wider community to promote the integration of dependability and formal methods research. We hope that you will find this volume interesting and encourage you to join the interest group of the EU FP-7 Deploy project (Industrial Deployment of System Engineering Methods Providing High Dependability and Productivity) that in particular aims at establishing closer collaboration between dependability and formal methods research.

December 2008

Michael Butler
Cliff Jones
Alexander Romanovsky
Elena Troubitsyna

Table of Contents

Part I: Formal Reasoning about Fault Tolerant Systems and Protocols

Part II: Fault Tolerance: Modelling in B

Part III: Fault Tolerance in System Development Process

Part IV: Fault Tolerant Applications

Graphical Modelling for Simulation and Formal Analysis of Wireless Network Protocols

A. Fehnker[1], M. Fruth[2], and A.K. McIver[3]

[1] National ICT Australia, Sydney, Australia*
ansgar@nicta.com
[2] Computing Laboratory, Oxford University UK**
m.fruth@comlab.ox.ac.uk
[3] Dept. Computer Science, Macquarie University, NSW 2109 Australia,
and National ICT Australia
anabel@ics.mq.edu.au

Abstract. It is well-known that the performance of wireless protocols depends on the quality of the wireless links, which in turn is affected by the network topology. The aim of this paper is to investigate the use of probabilistic model checking in the analysis of performance of wireless protocols, using a probabilistic abstraction of wireless unreliability.

Our main contributions are first, to show how to formalise wireless link unreliability via probabilistic behaviour derived from the current best analytic models [12], and second, to show how such formal models can be generated automatically from a graphical representation of the network, and analysed with the PRISM model checker.

We also introduce CaVi, a graphical specification tool, which reduces the specification task to the design of the network layout, and provides a uniform design interface linking model checking with simulation. We illustrate our techniques with a randomised gossiping protocol.

Keywords: Graphical modelling, simulation, lossy communication channels, probabilistic model checking, wireless networks.

1 Introduction

Wireless networks comprise devices with limited computing power together with wireless communication. Protocols for organising large-scale activities over these networks must be tolerant to the random faults intrinsic to the wireless medium, and their effectiveness is judged by detailed performance evaluation. One of the major factors impacting on the accuracy of an evaluation method is the underlying mathematical model used for the "communication channels". The most accurate models account for unreliabilities induced by noise and interference amongst close neighbours. Conventional analysis methods rely on simulators

* National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

** This work was in part supported by the EPSRC grant EP/D076625/2.

M. Butler et al. (Eds.): Fault Tolerance, LNCS 5454, pp. 1–24, 2009.
© Springer-Verlag Berlin Heidelberg 2009

[9,8] incorporating some measure of random faults, however simulation in this context suffers from a number of well-documented problems [7,3] — most notable is that accurate channel models validated against physical data do not normally feature. This leads to unrealistic results of performance analyses, which can vary widely between different simulators.

An alternative to simulation is formal modelling and analysis, which is normally ideally suited to investigating complex protocols, and gives access to profiles of performance which exhaustively range over worst- and best-case behaviour. Inclusion of realistic models of wireless communication implies appeal to analytical formulae to determine the effect on performance of the spatial relationships between nodes, such as the distance and density of near neighbours. These context-dependent details however are not easily added to textual-style formal modelling languages, and indeed they militate against a clear and modular specification style.

In this paper we overcome these difficulties by proposing a simple graphical style of specification. We exploit the observations that (a) the distance between and density of nodes in a network is the major factor impacting on the integrity of wireless communication (together with physical parameters such as transmission strength); that (b) this unreliability can be abstracted to a probability that packets are lost; and that (c) the simplest way to express the crucial spatial relationships is graphically, so that the details of the abstracted probabilities are suppressed, and computed automatically from the graphical representation.

Besides its simplicity, the graphical style has other benefits in that it allows designers to visualise various performance indicators such as best- or worst-case signal strength between pairs of nodes, or the nodes' individual power consumption. Similarly the critical events occurring in a sample experiment may be "stepped through" in a typical debugging style. Finally — unlike other graphical visualisation tools — it acts as a "bridge" between formal analysis and the more conventional simulation, providing the option to investigate performance using probabilistic model checking, or to carry out more traditional system-wide simulation experiments. In both cases realistic models for wireless communication play a fundamental role.

Our specific contributions are

1. CaVi a graphical user interface specialised for modelling networks comprising wireless nodes. The tool gives immediate access to crucial performance indicators such as signal strength between pairs of nodes;
2. A translation from a CaVi model to either a formal transition-style model suitable for model checking in the PRISM model checker [10] or as input to the recently-developed Castalia simulator [1]. Castalia is novel in that it incorporates an accurate wireless channel model. The PRISM models are the first such formal models which take network topology into account. At present both Castalia and PRISM capture only flooding and gossiping protocols [5,6].

In Sec. 2 and Sec. 3 we describe the context of wireless applications, and the challenges that arise in their formal modelling. In Sec. 4 we describe a well-known analytic model for describing unreliability of wireless links and explain how that can be used to compute the probabilistic abstractions. In Sec. 5 we illustrate how this can be incorporated in PRISM formal models for wireless protocols, and illustrate the effect on performance analysis. In Sec. 6 we introduce CaVi the graphical specification tool, and finally in Sec. 7 we demonstrate the techniques we have assembled with a case study based on gossiping.

2 Wireless Communication and Performance Modelling

In abstract terms a wireless network consists of a collection of nodes deployed over a two-dimensional area which together run a combination of protocols in order to achieve some specific goal. During operation the nodes routinely communicate using *wireless links* which are known to be highly unreliable, and indeed can have a significant impact on the overall performance of the system. In particular not only does the reliability of the wireless links attenuates as the distance between nodes extends, but it also falls off as the density of closely clustered nodes increases, since simultaneous broadcasts from separate sources can interfere and be effectively destroyed.

Thus the operability of the wireless network can depend as much on the topology of the network as on the correctness of underlying protocols. In particular the design of protocols are specifically intended to tolerate or reduce, as much as possible, the frequency of faults arising due to the unreliability involved in wireless communication. This paper is concerned with methods and tool support to help designers understand and evaluate the effectiveness of their designs.

With this goal in mind we set out the three challenges implied by the specification and performance evaluation of emerging wireless network protocols.

1. **Network specification:** As mentioned above the network behaviour depends critically on the network topology, suggesting that the topology should be encoded as part of the specification.

   Our first problem is *how to incorporate details of distance and relative clustering as part of the specification without leading to an infeasibly complicated specification language?*
2. **Realistic mathematical models:** Currently simulation is the major tool for evaluating performance of wireless networks. Whilst emerging simulators are beginning to account for accurate mathematical models of communication [1], simulation still suffers from several drawbacks. Aside from the underlying mathematical model being remote from the specifier, the resulting performance analysis is essentially “second order”, in the sense that it relies on a large number of simulation runs and, by implication, costly and time consuming.

   An alternative approach for protocol analysis is probabilistic model checking, so far under-explored as an evaluation method in the wireless domain. Model checking appears to overcome some of the problems surrounding simulation: the constructed models — Markov-style — are under direct control of the specifier, and the analysis involves direct algorithmic exploration of the associated mathematical structure. Thus the effect network parameters have on performance can be analysed relatively easily. Despite this appealing access to sensitivity analysis, a typical formal model checking approach assumes unrealistically that the links are either completely reliable, or uniformly unreliable, which is not the case.

Our second problem is *how should realistic wireless communication models be incorporated into formal model checking?*

3. **Scale versus accuracy:** Even if the modelling problem can be solved, the model checking technique is still only viable for small to moderately-sized networks, depending on the details of the protocol. Thus simulation is still the only feasible option for investigating system-wide properties over large networks. This indicates that if the analysis demands both an accurate evaluation of how parameters affect performance *and* a study of global network properties that both model checking *and* simulation are called for, with a consequent separate modelling effort for each.

   Our third problem is *how can the benefits of system-wide analyses be combined with the accuracy of model checking without doubling the modelling effort?*

In what follows we address all three problems. For problem (2) we extend probabilistic model checking in a novel way to account for unreliability of wireless links; for problem (1) we also introduce a graphical specification tool to make transparent the relevant details of the network topology, whilst still accounting for them in the analysis; and finally, for problem (3), we explore how the graphical specification can provide a bridge between model checking and simulation with minimal duplication of modelling effort.

To extend probabilistic model checking, we render the unreliability of wireless communication as a probability that packets get lost. To ensure that this abstraction is as realistic as possible we compute the probabilities using an analytic formula validated against experimental field data [12]. We show how these probabilities can be translated in the PRISM model checker [10] allowing accurate formal model checking to be performed after all.

Next, inspired by other graphical tools [9], we propose a graphical style of specification to reduce the specification effort, exploiting two of the above observations: specifying topological details can be done most naturally by drawing a diagram, and the “run time” probabilities of communication failure — accounting for both distance between and density of nodes — can be computed automatically from the corresponding graphical representation.

Our graphical specification tool CaVi simplifies the specification task of communication details; moreover for simple protocols it can act as a uniform modelling language combining large-scale performance analyses based on simulation with the accurate sensitivity analysis offered by model checking.

In the remainder of the paper we set out the details.

2.1 The PRISM Model Checker

The PRISM model checker [10] takes a specification of a system using a modelling language for describing probabilistic state transition systems. In such a model, a system is regarded as a collection of "communicating" modules, each one consisting of a set of guarded commands, with each command composed of the guard (a predicate on the variables) and a probabilistic update relation (the probabilistic assignment to variables). Modules can communicate by enforcing the assumption that same-labelled guarded commands must fire simultaneously; additionally information can be shared between modules by their reading the values of others' variables. Once specified the PRISM model checker constructs an internal representation of the system model — a Markov-style transition system for the composition of specified modules — which can then be analysed exhaustively relative to a specified property using a suite of numerical algorithms.

Property specification is via probabilistic temporal logic [2], which is expressive enough to describe many performance style properties; PRISM computes the best- and worst-case probability of satisfaction. In this paper we shall use that to analyse whether nodes eventually receive a message sent in a network protocol.

The importance of this approach (as compared to simulation for example) is that precise probabilistic results are computed exhaustively and are relevant to the entire set of executions, rather than a simulated subset.

3 Modelling Lossy Wireless Communication

Wireless nodes typically broadcast a message on a particular frequency — in the case that several nodes broadcast using the same frequency at approximately the same time, the messages can interfere so that the receiving node only detects noise. In this section we discuss the effect on performance evaluation of the precise modelling assumptions used in the formalisation of unreliable wireless channels.

Consider the simple network in Fig. 2 depicting a four node network. Suppose now that *Source* attempts to send a message to *Target*, but that they are too far apart to be connected directly by a wireless link. In this case *Source* must rely on relaying the message via the intermediate nodes $Node_A$ and $Node_B$. We consider a simple communication protocol in which *Source* broadcasts a message to be picked up by $Node_A$ and $Node_B$, both of which then forward the message on to *Target*.

Depending on the assumptions in the mathematical model used to handle the reliability of communication, very different conclusions as to the behaviour of the system can be drawn. To illustrate this we provide three simple formal models of Fig. 2, each based on different assumptions, and we discuss the implications of each one.

We model the behaviour of each node as a simple state transition system; in this small example we assume that the behaviour of *Source*, $Node_A$ and $Node_B$ is given by the systems set out in Fig. 1, leaving the assumptions about the reliability of communications to vary only in the model for *Target*[1], set out below. *Source* has only one action, to send a message, whilst each of $Node_A$ and $Node_B$ receive the message, synchronising on the event **recv**, and attempt to forward it to *Target*. The message is deemed to have been delivered successfully on the occurrence of either $\mathbf{send}_\mathsf{A}$ or $\mathbf{send}_\mathsf{B}$. The case of messages interference is modelled by the event **clash**.

Next we illustrate how the analysis depends crucially on the assumptions used in the formal model. The definitions below define three possible scenarios, each one formalising a different assumption concerning simultaneous broadcasts.

$$Source \mathrel{\hat{=}} \left(\begin{array}{l}\mathbf{var}\ t\colon \{sending,\ sent\}\\ \mathbf{recv}:\ \ (t = sending) \rightarrow\ \ t\!:=\ sent\end{array}\right)$$

$$Node_A \mathrel{\hat{=}} \left(\begin{array}{l}\mathbf{var}\ f_\mathsf{a}\colon \{listen,\ sending\}\\ \mathbf{recv}:\ \ (f_\mathsf{a} = listen) \rightarrow\ \ f_\mathsf{a}\!:=\ sending\\ \mathbf{send}_\mathsf{A}:\ \ (f_\mathsf{a} = sending) \rightarrow\ \ f_\mathsf{a}\!:=\ listen\\ \mathbf{clash}:\ \ (f_\mathsf{a} = sending) \rightarrow\ \ f_\mathsf{a}\!:=\ listen\end{array}\right)$$

The definition for $Node_B$ is the same as for $Node_A$, except that the state variable is f_b and the second event is named $\mathbf{send}_\mathsf{B}$.

Fig. 1. Behaviour of *Target* and intermediate nodes $Node_A$ and $Node_B$

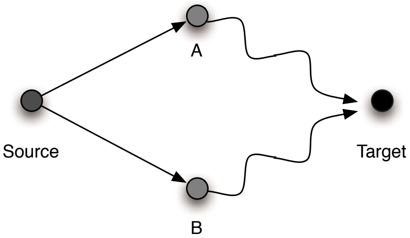


Fig. 2. Illustrating interference

(1) Worst case interference assumption: A worst case interference assumption implies that if messages are sent (almost) simultaneously from $Node_A$ and $Node_B$, then they will certainly interfere with each other. The model for $Target_1$ at Fig. 3 encodes this assumption by the state registering *noise* on the execution of the event **clash**. The whole system is now given by

$$System_1 \quad \hat{=} \quad Source \| Node_A \| Node_B \| Target_1\ ,$$

with synchronisation between same-named events. Not surprisingly, the probability of the *Target*'s reception of the message is computed as 0.

[1] Strictly speaking we should also include the possibility of unreliability in the communications for $Node_A$ and $Node_B$, but for the moment we assume that *Source*'s communications to $Node_A$ and $Node_B$ are fully reliable.

$$Target_1 \mathrel{\hat{=}} \left(\begin{array}{l} \textbf{var } src\colon \{listen,\ receive,\ noise\} \\ \textbf{send}_{\mathsf{A}} :\ ((src = listen) \land (\mathsf{s}_{\mathsf{B}} \neq sending)) \rightarrow\ src\colon=\ receive; \\ \textbf{send}_{\mathsf{B}} :\ ((src = listen \land (\mathsf{s}_{\mathsf{A}} \neq sending)) \rightarrow\ src\colon=\ receive; \\ \textbf{clash} :\ (src = listen \land \mathsf{s}_{\mathsf{A}} = \mathsf{s}_{\mathsf{b}} = sending) \rightarrow\ src\colon=\ noise; \end{array} \right)$$

Fig. 3. Worst-case interference assumption

(b) Best case interference assumption: Alternatively we could encode the most optimistic assumption, that the *Target* receives the message if either one of $Node_A$ or $Node_B$ forward the message. $Target_2$ in Fig. 4 encodes this best-case assumption — it does not include an event clash in its repertoire of events, but rather only the possibility of receiving from either $Node_A$ or $Node_B$, either possibility being considered depending on which of $Node_A$, or $Node_B$ is ready to send. In this model,

$$System_2 \quad \hat{=} \quad Source \| Node_A \| Node_B \| Target_2 \ ,$$

the Target is certain to receive the message.

$$Target_2 \mathrel{\hat{=}} \left(\begin{array}{l} \textbf{var } src\colon \{listen,\ receive,\ noise\} \\ \textbf{send}_{\mathsf{A}} :\ ((src = listen) \land (\mathsf{s}_{\mathsf{A}} = sending)) \rightarrow\ src\colon=\ receive; \\ \textbf{send}_{\mathsf{B}} :\ ((src = listen \land (\mathsf{s}_{\mathsf{B}} = sending)) \rightarrow\ src\colon=\ receive; \end{array} \right)$$

Fig. 4. Best-case interference assumption

(c) Average case interference assumption: In reality experiments have shown that the situation lies somewhere between those worst- and best-case scenarios, and in fact the precise positioning of the *Target* relative to $Node_A$ and $Node_B$ can be crucial to the overall reliability of message relay from *Source*: if *Target* is located close to *both* the intermediate nodes (for example symmetrically between them), then their simultaneous forwarding of the message will interfere and *Target* will not get it. Conversely if *Target* is placed too far afield then, in any case, the signal strength of the received messages will be so weak as to effectively disconnect *Target* from the network.

We formalise this average-case assumption in $Target_3$ set out in Fig. 5. Here on execution of the event clash there is a probability p_r of either one of the messages (from $Node_A$ or $Node_B$) arriving uncorrupted. The probability that *Target* now receives the message in the system defined by

$$System_3 \quad \hat{=} \quad Source \| Node_A \| Node_B \| Target_3$$

is now *at least* p_r.

As we shall see, the precise value of p_r — referred to below as the *link probability* — depends on a number of factors, including the distance and spatial orientation of $Node_A$ and $Node_B$ from *Target*, and from each other, and thus p_r itself can be thought of as an abstraction for the topological details of the network. In the next section we describe how that is done.

$$
Target_3 \mathrel{\hat{=}} \left(\begin{array}{l}
\textbf{var } \mathit{src} \colon \{\mathit{listen},\ \mathit{receive},\ \mathit{noise}\} \\
\textbf{send}_{\mathsf{A}} : \ ((\mathit{src} = \mathit{listen}) \wedge (\mathsf{s}_{\mathsf{B}} \neq \mathit{sending})) \rightarrow \ \mathit{src} := \ \mathit{receive}; \\
\textbf{send}_{\mathsf{B}} : \ ((\mathit{src} = \mathit{listen} \wedge (\mathsf{s}_{\mathsf{A}} \neq \mathit{sending})) \rightarrow \ \mathit{src} := \ \mathit{receive}; \\
\textbf{clash} : \ (\mathit{src} = \mathit{listen} \wedge \mathsf{s}_{\mathsf{A}} = \mathsf{s}_{\mathsf{b}} = \mathit{sending}) \rightarrow \ \mathit{src} := \ \mathit{receive}\ {}_{\mathsf{p_r}}\!\oplus \mathit{noise};
\end{array}\right)
$$

The probability p_r captures the uncertainty of receiving either one (but not both) of $Node_A$ or $Node_B$'s forwarded message. Its precise value depends on the relative distances between $Node_A$, $Node_B$ and *Target*.

Fig. 5. Average-case interference assumption

4 Formal Abstractions of Signal Strength and Interference

In this section we discuss how the link probability mentioned above can be calculated more generally within arbitrary networks to take account of the distance between nodes and their relative clustering. We also discuss the other network parameters impacting on the probability.[2]

Consider first the simple case set out at Fig. 6(a) of two nodes i and j a distance $d(i,j)$ apart, with j sending a message to i. The probability that i receives j's message is computed as a function of the *signal-to-noise ratio*, $SNR_{i,j}$ which is the ratio of the *power* of the received message at i $(rx_{i,j})$, and the *noise* $(bgN_{i,j})$ generated in part by the other activities of the network, as well as general conditions of the environment. Thus $SNR_{i,j} \mathrel{\hat{=}} rx_{i,j}/bgN_{i,j}$. We discuss first the analytic formulae for the latter two quantities.

Power and Noise Levels: The signal strength of the received message $rxbB_{i,j}$ depends on the distance $d(i,j)$ between i and j, and the power at which j transmits, tx_j, and is given by the formula

$$rxbB_{i,j} \quad \hat{=} \quad tx_j - PLd_0 - 10(pLE)\,\log_{10}(d(i,j)/d_0)\ , \tag{1}$$

where pLE is called the *path loss exponent*, and can be thought of as the rate at which the signal strength deteriorates with distance, and d_0 and PLd_0 are scaling constants determined by the environment. The power at the receiver can now be computed directly:

$$rx_{i,j} \quad \hat{=} \quad 10^{rxdB_{i,j}/10} \tag{2}$$

Next we compute the background noise. In the simple case depicted in Fig. 6(a) where there are no neighbouring nodes, the background noise is assumed to be a constant $nbgN$ determined by the operating environment. In more complicated scenarios, depicted in Fig. 6(b), the noise generated by the other nodes in the network must be taken into account. Let $send_k$ be a function which is 1 or 0 according

[2] The analytic formulas referred to in this section are all taken from Zuniga and Krishnamachari [12].

to whether node k is transmitting a message or not. The total background noise at receiver i interfering with the message transmitted by j is given by

$$bgN_{i,j} \quad \hat{=} \quad nbgN + \sum_{k \neq i,j} rx_{i,k} * send_k \ . \tag{3}$$

With the two quantities $bgN_{i,j}$ and $rx_{i,j}$ given at (2) and (3) respectively we can now compute the probability that i receives j's message.

Link Probabilities: The current analytic models for computing the link probabilities predict that there is signal-to-noise threshold below which there is effectively zero probability that the message can be decoded by the receiver. That threshold depends on a number of network specific parameters: the *data rate* nDR, the *noise bandwidth* nBW, the *threshold probability* nTP, the *frame length* f of the message, and the *modulation type* of the transmission. Here, we use *Frequency Shift Keying* (FSK) which describes a simple method of encoding information into the carrier wave using only two states. For FSK modulation, the threshold is computed as

$$\Delta_{i,j} \quad \hat{=} \quad -2\frac{nDR}{nBW}\log_e(2(1 - nTP^{\frac{1}{8f}})) \ . \tag{4}$$

with (4) we can finally compute the link probabilities. First we compute the threshold-free probability that j's message is received by i

$$\text{snr2prob}(SNR_{i,j}) = (1 - 0.5 * \exp(-0.5\frac{nBW}{nDR}SNR_{i,j}))^{8f} \ , \tag{5}$$

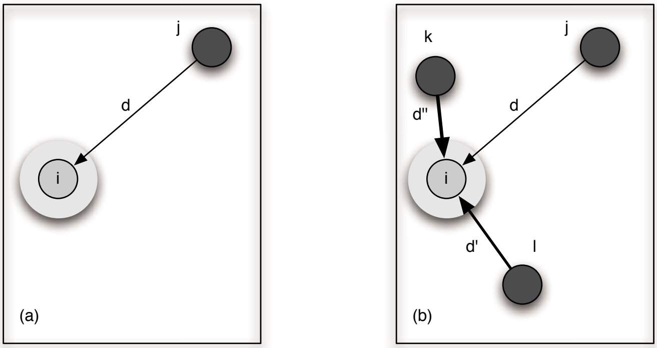
where we recall that $SNR_{i,j} \hat{=} rx_{i,j}/bgN_{i,j}$, and that $bgN_{i,j}$ is given at (3). And now taking the threshold into account, we have

$$precv_{i,j} \quad \hat{=} \quad \begin{cases} 0 & \text{if } SNR_{i,j} < \Delta_{i,j} \\ \text{snr2prob}(SNR_{i,j}) & \text{otherwise} \end{cases} \tag{6}$$

Note that since this formula depends on the mutual contribution to the noise of the surrounding nodes, this is actually a conditional probability, namely the probability that i receives j's message *given* that i does not receive the message from any of the other nodes. This together with the assumption that if $j \neq k$ then the events "i receives a message from node j" and "i receives a message from node k" are mutually disjoint, implies that we can can compute the probability P_i that any message is received by i (from whichever sender) as the sum of the individual link probabilities:

$$P_i = \sum_{j \neq i} precv_{i,j} * send_j \ , \tag{7}$$

where $send_j$ was defined above.



Sender k is closest to receiver i, so its signal is strongest; Sender j's is weakest. All link probabilities are affected by the others' activities. Here d, d' and d'' are the distances from the senders to the receiver i.

Fig. 6. Signal strength varying with distance and interference

4.1 Translating the Analytic Model to PRISM

In this section we consider a PRISM model for a small example network based on that illustrated in Fig. 2 with the detailed link probabilities modelled as probabilistic transitions. The PRISM source file appears in the appendix, and here we transcribe and explain examples representing each separate section of that model.

Four nodes are numbered 0, 1, 2, 3, with 0 and 3 corresponding to the *Source* and *Target* respectively. All nodes are either active or inactive; when a node i is active (`activei = 1`) it can listen for and/or receive a message, or send one it received previously. If a node is not active, then it is inactive (`activei = 0`), in which case it only listens. We use the variable `sendi` to denote whether node i is in possession of an uncorrupted message (`sendi = 1`) which it must forward, or not (`sendi = 0`) (either because one was never received, or because it has already been forwarded). As described above, in this simple protocol, nodes listen for a message and then forward it; once it has received and sent a message a node becomes inactive. The following PRISM code formalises this behaviour for node 3, where `recvp3` is the link probability which depends on the state of the surrounding nodes, and whose calculation we describe below.

```
module node3
    active3:[0..1] init 1;
    send3:  [0..1] init 0;

[tick] send3=0&active3=1 -> recvp3:(send3'=1)&(active3'=1)+
                                (1-recvp3):(send3'=0)&(active3'=1);
[tick] send3=1&active3=1 -> send3'=0&active3'=0;
[tick] active3=0 -> send3'=0&active3'=0;
endmodule
```

Note that this is a synchronous implementation of the abstract network described above in Sec. 3, which though easier to understand, if implemented directly would have a significant impact on space and model checking times, as well as the feasibility of automating the generation of PRISM models. In this synchronous style the nodes all behave in lockstep, synchronising on the action `tick`. The difference in behaviour between whether one node or several nodes broadcast at the same time is all accounted for in the link probabilities (rather than explicitly by including separate labelled guarded commands for each as it was explained in Sec. 3).

Next, to incorporate the link probabilities from (6) and (7) in a PRISM model, we need to compute the various quantities such as the transmission powers, the signal-to-noise ratios and thresholds for each node, taking into account their actual pairwise separations. Due to the limitations on the available arithmetical functions implemented in the current distributed version of PRISM, we precomputed $rx_{i,j}$ from (2), the power at the receiver i from message broadcast by j. These values are denoted in the PRISM model by `linRxSignal_i_j`, and appear as constant declarations for each pair of nodes. For example between nodes numbered 1 and 3, the reception power between pairs of nodes is:

```
const double linRxSignal_1_3 = 3.04330129123453E-8;
const double linRxSignal_3_1 = 3.04330129123453E-8;
```

Next the signal-to-noise ratio $SNR(i, j) = rx_{i,j}/bgN_{i,j}$ between pairs of nodes can be calculated from the above figures for reception power, given by equations (1), (2) and (3). As examples we give those quantities for $SNR(3, 1)$.

```
formula snr_3_1 = (linRxSignal_3_1*send3)/
     (linRxSignal_0_1*send0 + linRxSignal_2_1*send2 + 1.0E-10);
```

Next the conditional link probabilities $precv_{i,j}$ at (7) are calculated from the precomputed thresholds, and combined in a single PRISM formula, with $precv_{3,2}$ given as an example,

```
formula Preceive_3_2 = func(max,0,(snr_3_2>=12.357925578002547)?
      func(pow,(1-0.5*func(pow,2.71828,-0.781*snr_3_2)), 8 * 25):0);
```

Finally the total link probabilities P_i are computed as the sum, as at (7), where we take the precaution of ensuring that the sum does not exceed 1 (which sometimes happens due to rounding errors).

```
formula recvp0 = func(min,1,Preceive_1_0+Preceive_2_0+Preceive_3_0);
formula recvp1 = func(min,1,Preceive_0_1+Preceive_2_1+Preceive_3_1);
formula recvp2 = func(min,1,Preceive_0_2+Preceive_1_2+Preceive_3_2);
formula recvp3 = func(min,1,Preceive_0_3+Preceive_1_3+Preceive_2_3);
```

5 Performance Analysis with PRISM

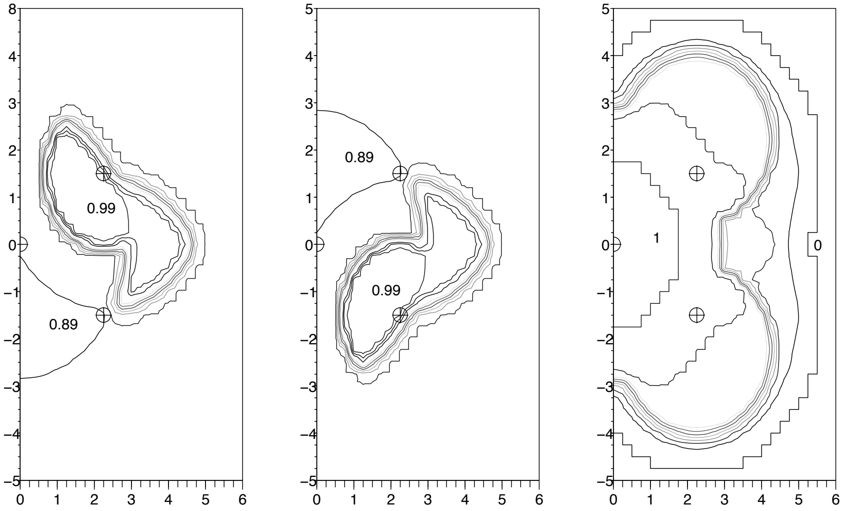
In this section we illustrate how our PRISM model captures the unreliability of wireless links caused by distance and interference between neighbouring nodes. Observe that once the background noise has been set, the only variables are the node specific parameters (such as power and signal strength) and the distances between nodes.

In this experiment the three nodes 0, 1 and 2 were placed so that 1 and 2 had a high chance of receiving the message broadcast by 0, and the effect on performance of nodes 3's position investigated. Thus for various positions of node 3 we computed the separate probabilities that nodes 1, 2 and 3 obtained the message. Specifically we used the PRISM model checker to compute the probability that the following temporal logic properties for strong until were satisfied:

```
Pmin=? [send1 = 0 U send1 = 1]
Pmin=? [send2 = 0 U send2 = 1]
Pmin=? [send3 = 0 U send3 = 1]
```

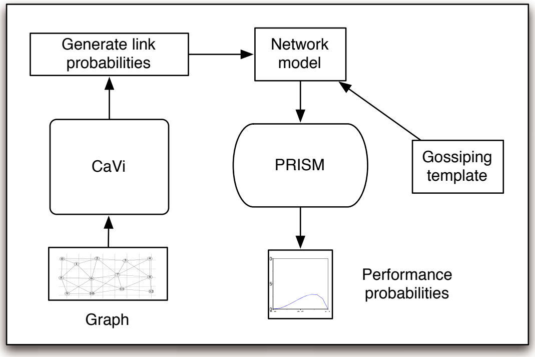
namely the chance that eventually `sendi = 1` for each of the nodes $i = 1, 2, 3$. The results appear as three contour plots in Fig. 7.

The right-hand plot, for node 3 illustrates in particular the effects of interference and distance. The 0 contour indicates that the distance is the major factor, with node 3 being far too far from any of the other nodes to receive any message. The cluster of contours around nodes 1 and 2 however shows clearly the impact of interference: if node 3 is at position $(x, y) \hat{=} (3.5, -2)$ say, it has a high chance of receiving the message from node 2, even it is too far to hear node 0 directly. As node 3 moves up vertically (increasing its y component), it becomes closer to node 1, so that interference between nodes 1 and 2 becomes the influential



The three placements of $\oplus$ in each diagram indicate the static positions of nodes $0, 1$ and 2. The contour plots show the probability that the respective nodes eventually receive a message. The contour lines for nodes 1 and 2 show the iso-probabilites in steps of 0.01, and for node 3 in steps of 0.1.

Fig. 7. Analysis of interference



This diagram illustrates how CaVi interfaces with PRISM for analysing gossiping protocols. The user develops a graphical representation of the network, from which CaVi can compute the link probabilities. These are combined with the flooding template to produce an input file for PRISM describing the model. The PRISM tool can then be used to analyse performance.

Fig. 8. CaVi and PRISM

factor, and the probability of ever receiving the message falls rapidly. Finally the high chance of receiving the message when located *directly between* nodes 1 and 2 is due to node 3 receiving node 0's broadcast directly.

The plots for nodes 1 and 2 are symetrical, and from them we notice that the movement of node 3 has only a small effect on either of them eventually receiving the message, as the most likely scenario is that they receive the message directly from node 0. In the case that node 3 is placed closer to node 0 than either of the other two, node 3 actually acts as the intermediary, increasing the chance that node 1 say receives the message, in the case that it didn't receive it directly from the source.

6 CaVi: A Graphical Specification Tool

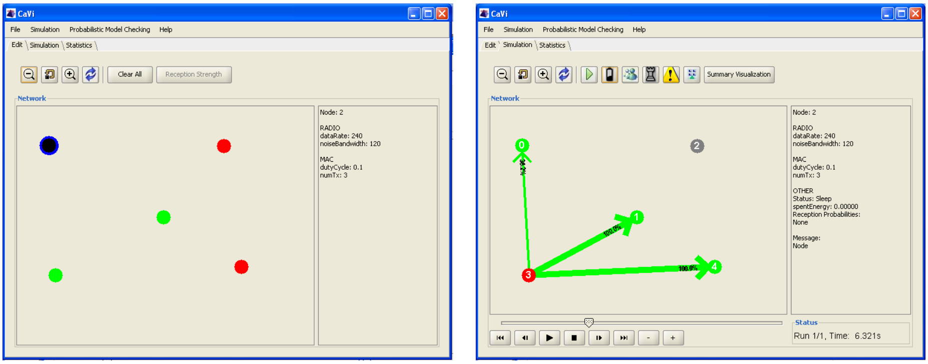
As we have seen, it is possible to formalise interference by using a probabilistic abstraction, but the detailed calculations required to introduce them into a PRISM model for example are too intricate to do by hand. To overcome this problem we have designed and implemented CaVi, a graphical specification tool which can automate the procedure of preparing a formal PRISM model with link probabilities computed directly from a graphical representation of a network. This eases considerably the task of specification in situations when the nodes all execute identical code.

The main feature of CaVi is its graphical interface, with which a user can design a specific network layout. Nodes may be created in a "drag-and-drop" fashion, and the properties of individual nodes (such as the power and signal strength) may be tuned as necessary via individual node menus of parameters.

During development a user can visualise the worst- and best-case link probabilities, calculated from equation (6).

In Fig. 9 we illustrate two examples of how the graphical interface may be used in the design and analysis. The figure shows two panes, with the left being the pane where designers may create and edit a network, and the pane on the right is for visualising the results of simulation experiments. In the left-hand pane, for example, a user may indicate which node is the receiving node (in this case the central node), and the others are assumed to be senders. Colours then differentiate between nodes whose messages will be almost certainly lost (red), have a good chance of succeeding (green), or merely a variable chance (yellow).

The pane on the right indicates how the events may be viewed as a result of a simulation experiment. Simulation allows the user to "step through" an example execution sequence in a dynamic experiment. In this case the display shows which of the nodes is sending, to whom they are connected, and with what strength. In this snapshot the bottom left node is transmitting, and is connected to all but the top right node. The thickness of the arrows indicate the strength of the connection, with the vertical connection being somewhat weaker than the other two.



Node: 2

RADIO
dataRate: 240
noiseBandwidth: 120

MAC
dutyCycle: 0.1
numTx: 3

Node: 2

RADIO
dataRate: 240
noiseBandwidth: 120

MAC
dutyCycle: 0.1
numTx: 3

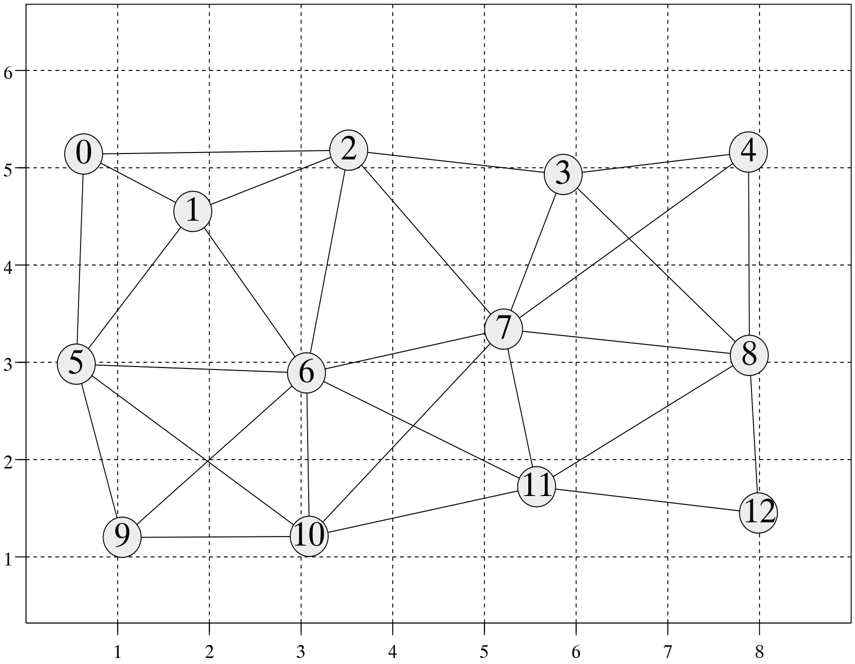
OTHER
Status: Sleep
spentEnergy: 0.00000
Reception Probabilities:
None

Message:
Node

Run 1/1, Time: 6.321s

Fig. 9. CaVi:Visualising network performance indicators

Once the network is specified, the graphical representation forms the basis for formal models which take account of the effect of the topology in terms of the link probabilities. Using a template for the per-node behaviour of the protocol, the functions used to compute the actual probabilities are printed to a PRISM model file, together with an instantiation of a pre-prepared template for each numbered node. Currently we only have a template for gossiping- and flooding-style protocols, although the aim would be to expand that to other common patterns. An example of the automatically-generated PRISM model for the four node network is provided in the appendix. This model can then be fed into PRISM for detailed performance evaluation. The diagram at Fig. 8 illustrates the relation between CaVi and PRISM.



Node 0 is the node from which the message originates; the other nodes follow a generic gossiping protocol. Lines connecting nodes indicate active connections in the network.

Fig. 10. Graphical specification of a 13-node network

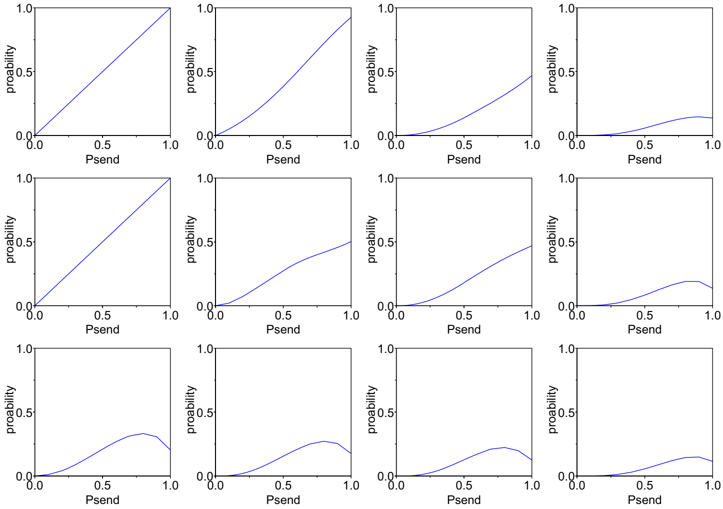
CaVi can also generate an input file for the Castalia wireless simulator, which we discuss briefly in Sec. 7.2 below. Other features of CaVi include some visualisation techniques of network performance indicators, and simulation runs, discussed in detail elsewhere [4].

7 Putting It all Together: Randomised Gossiping

In this section we show how to use the CaVi tool for investigating a PRISM model for a probabilistic gossiping protocol.

We consider the problem of designing a flooding protocol to maximise the probability that the message is distributed over a network. It is known that using a simple flooding protocol such as described in Sec. 5, where nodes send as soon as they receive the message suffers from some serious performance issues, due to the high likelihood of interference. To mitigate this problem, variations of this basic scheme have been proposed in which nodes only forward a received message with probability p. The aim is to choose the value of p to optimise the probability of the message being received by all nodes.

Using the CaVi tool, we first specify the network topology by placing the nodes in the arrangement given in Fig. 10. Here node 0 is assumed to be the



Each graph represents the probability that node i eventually receives the message. The top row of graphs correspond to nodes 1, 2, 3, 4 Fig. 10; the second row to 5, 6, 7, 8, and the third row to 9, 10, 11, and 12.
For each graph the horizontal axis is the probability `Psend`, and the vertical axis is the probability of eventually receiving the message.

Fig. 11. Per-node probabilities of eventually receiving the message

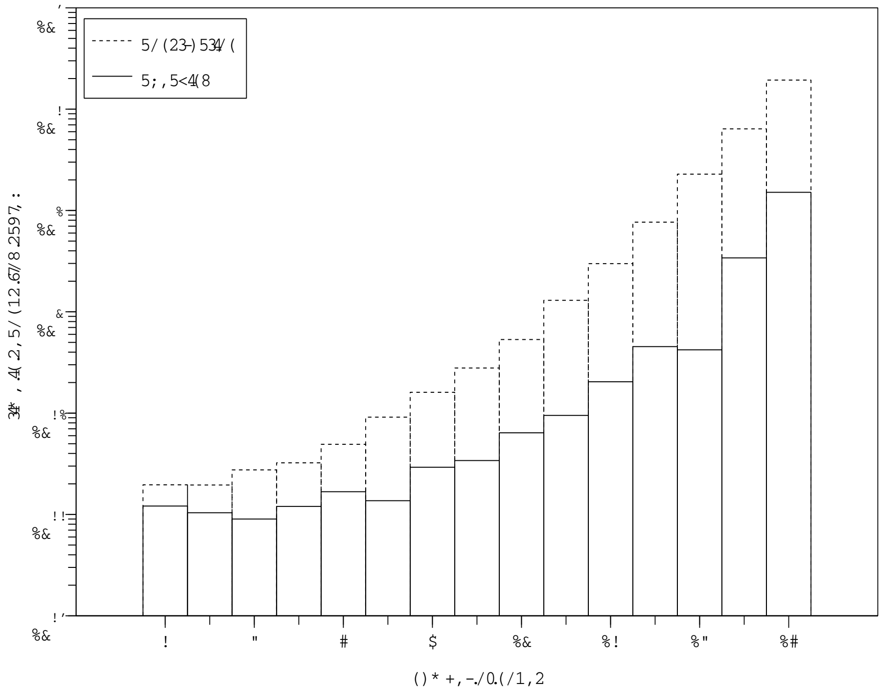
originator of the message to be distributed. The other nodes' behaviour is given as for the example in Sec. 5, but with an additional parameter, which is the probability of sending. In the PRISM model that is given by `Psend` and in this study it is the same for all nodes throughout the network.

For example, node 1's behaviour is described by the PRISM model below, where now the chance of a message being forwarded is combined with the link probability, so that the overall probability that a message is received is `Psend*recvpi`, and the chance of it not being received is `(1-Psend)*recvpi`.

```
module node1
   active1:[0..1] init 1;
   send1:  [0..1] init 0;

[tick] send1=0&active1=1 -> Psend*recvp1:(send1'=1)&(active1'=1)+
                        (1-Psend)*recvp1:(send1'=0)&(active1'=0)+
                         (1-recvp1):(send1'=0)&(active1'=1);
[tick] send1=1&active1=1 -> send1'=0&active1'=0;
[tick] active1=0 -> send1'=0&active1'=0;
endmodule
```

As before the probabilities for receiving `recvpi` are calculated automatically by the CaVi tool and combined with a node template to create automatically a PRISM model for gossiping in the network with layout at Fig. 10. Next we used

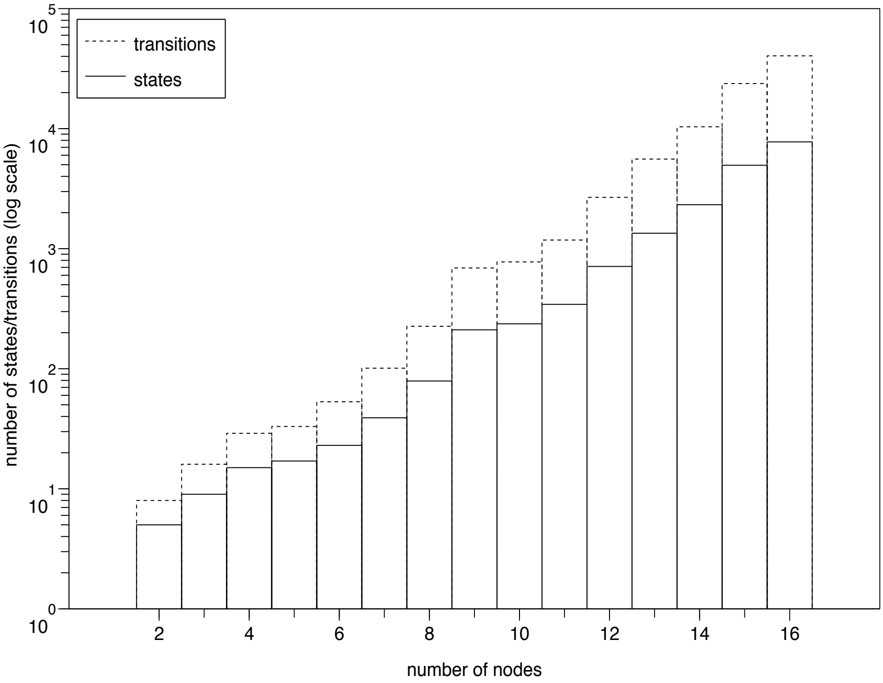


This shows the time in seconds (with a log scale) for the time to build and model check the generated gossiping models in PRISM. The dashed bars are the model building times, and the solid bars the model checking times.

Fig. 12. Model checking and model building times

PRISM to compute the per-node probabilities of eventually receiving the message. The results appear at Fig. 11 where we have plotted a separate graph for each node, with the vertical axes being the probability that the message is eventually received by that node, and the horizontal axis is Psend, which varies between 0 and 1.

The results show that the nodes 1, 2 and 5 clustered close to the originator node 0 have a probability of eventually receiving proportional to Psend, since they receive the message (if at all) directly from node 0, and since there can be no interference when only node 0 broadcasts. The nodes 3, 4, 7, 8, 11 and 12, all positioned too far away to receive it first-hand however have a non-linear relationship with Psend. When Psend is too low, then they have a very slim chance of receiving the message at all, since they are relying on a chain of nodes to forward their received message, and in this case the forwarding probability Psend is very low for each. On the other hand these nodes also have a low chance of ever receiving the message when Psend is high, because although the intermediate nodes will send with high probability, their messages also have a high chance of being destroyed by interference. The network-wide optimal value for Psend appears to be around 0.8.



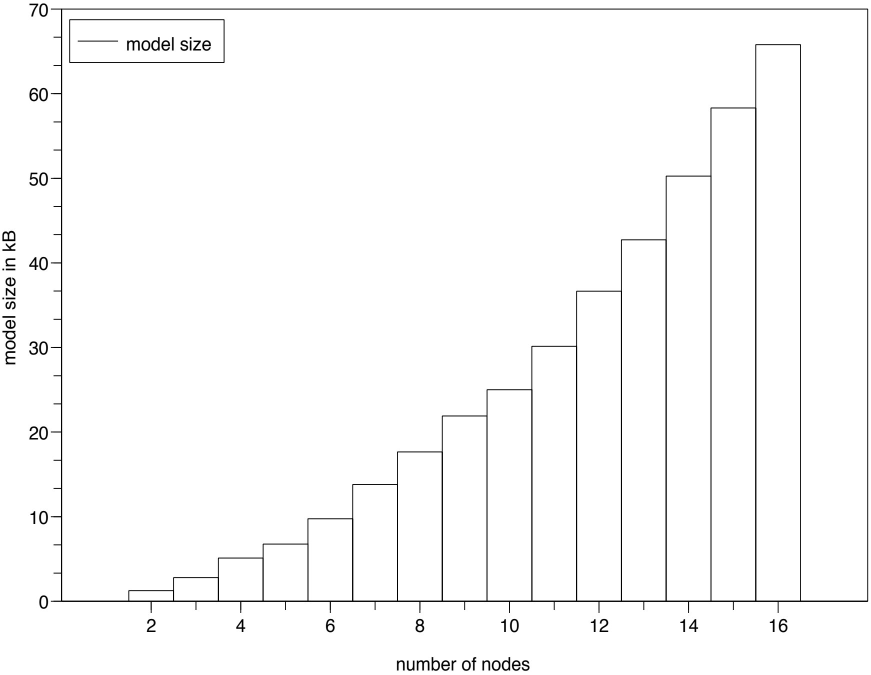
This shows the number of states and transitions (with a log scale) of the constructed gossiping models, varying with the number of nodes in the network.

Fig. 13. The number of states and transitions

7.1 Discussion

The use of CaVi to include the network topology characteristics to generate the formal model is an important step. Whilst the formulae for computing the probabilities are uniform, the task of preparing them by hand would be too time consuming and prone to error.

The templates for gossiping we use for the subsequent generation of the network models combine receiving and forwarding in a single probabilistic transition, leading to very compact internal PRISM representations of the constructed system. Fig. 13 and Fig. 14 for example give some idea as to the growth rate of the number of states and transitions as the number of network nodes increases. Though the growth is still exponential for substantially-sized networks (16 nodes) the actual size is still well within the capability of the PRISM. Similarly Fig. 12 shows the actual time spent by PRISM to construct and perform the model checking to produce the plots in Fig. 11. Interestingly the time to construct these models is an order of magnitude greater than the time to do the model checking, and this is largely due to the time spent parsing the function definitions for the calculation of the link probabilities.



This shows how the size of the model (in kilobytes, with a log scale) varies with the number of nodes in the network.

Fig. 14. Memory requirements

7.2 Model Checking and Simulation

Castalia [1] is a recently-developed simulator for wireless networks, whose novelty is that it incorporates an accurate model for wireless communication. It takes as input a file containing parameters describing the power, signal strength, "geographical" position of each node; once these have been specified, the simulator executes by effectively stepping through a sequence of possible states. Where the behaviour depends on the result of a random event, the simulator generates a random number to resolve the choice. Thus the simulator recreates, as far as possible, the results that would be obtained from testing the physical system. Statements about the performance of the system are based on statistical analysis of a large collection of many simulation runs. The errors in this kind of analysis come from the statistical analysis as well as from inaccuracies in quantifying the random events in the simulation model.

The advantages of simulation however are that since it only records the result of an actual run, it is able to cope with large networks and thus can still give a performance forecast of a network made up of many nodes.

On the other hand a very large number of simulation are runs required to obtain the same accuracy as can be obtained with model checking [11] making the overall enterprise very costly in time.

Currently we are able to generate both PRISM or Castalia models from the graphical input to CaVi, and thus we have established it as a single graphical interface between model checking and simulation. We envisage that one use of such an interface would be the ability to visualise the results obtained from both in a uniform way. Such a “bridging language” would allow “counterexamples” computed via model checking to be validated in the simulator, for example, although how best to combine model checking and simulation most effectively is still a topic for research.

8 Conclusions and Future Work

In this paper we have described a prototype tool which supports a uniform modelling approach optimised for specifying wireless protocols. Its main features include the capabilities to take account of the topology and other parameters of the network which, experiments have shown, have a major impact on the integrity of the communication. The CaVi tool allows the specification of a network via a graphical interface, and the automated generation of formats for simulation and model checking. Detailed performance indicators may be visualised during specification of the network, as well as the results of subsequent simulation and model checking experiments.

The principal difference between CaVi and other specification tools is the link it provides between simulation and formal model checking. To simplify the details related to the topology in the formal specification task, we use a translation directly to link probabilities. Those probabilities are calculated according to a validated analytic formula.

Currently we only supply templates for gossiping and flooding protocols; whilst we do not envisage a translation from a CaVi model of an arbitrary protocol to PRISM, we would aim rather to provide a library of templates for certain classes of protocol whose precise behaviour can be defined by a number of parameters, in the same way that models are defined in Castalia.

We have not explored fully the uses of our formal models, for example whether it could be used to investigate the extent of fault tolerance that needs to be built into an unreliable network. That remains an interesting topic for future research.

Acknowledgements: We thank Viet Cuong Nguyen and Michael Ma for help with the implementation of CaVi.

References

1. Boulis, A.: Castalia: A simulator for wireless sensor networks, `http://castalia.npc.nicta.com.au`
2. Aziz, A., Singhal, V., Balarinand, F., Brayton, R.K., Sangiovanni-Vincentelli, A.L.: It usually works: The temporal logic of stochastic systems. In: Wolper, P. (ed.) CAV 1995. LNCS, vol. 939, pp. 155–165. Springer, Heidelberg (1995)

3. Cavin, D., Sasson, Y., Schiper, A.: On the accuracy of manet simulators. In: Proceedings of the second ACM international workshop on Principles of mobile computing, pp. 38–43. ACM Press, New York (2002)
4. Fehnker, A., Fruth, M., McIver, A.K.: Cavi: A graphical specification tool for wireless network protocols. In: The Proceedings of Quantitative Evaluation of Systems, 2008 (to appear) (2008)
5. Fehnker, A., Gao, P.: Formal verification and simulation for performance analysis for probabilistic broadcast protocols. In: Kunz, T., Ravi, S.S. (eds.) ADHOC-NOW 2006. LNCS, vol. 4104, pp. 128–141. Springer, Heidelberg (2006)
6. Fehnker, A., McIver, A.: Formal analysis of wireless protocols. In: Proc. 2nd International Symposium in Leveraging applications in formal methods, verification and validation, pp. 263–270. IEEE Computer society digital library, Los Alamitos (2007)
7. Kotz, K., Newport, C., Gray, R.S., Liu, J., Yuan, Y., Elliott, C.: Experimental evaluation of wireless simulation assumptions. In: Proceedings of the 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems, pp. 78–82. ACM Press, New York (2004)
8. The network simulator ns 2, `http://www.isi.edu/nsnam/ns/`
9. OPNET, `http://www.opnet.com/`
10. PRISM. Probabilistic symbolic model checker, `http://www.prismmodelchecker.org/`
11. Younes, H.L.S., Kwiatkowska, M., Norman, G., Parker, D.: Numerical vs. Statistical probabilistic model checking: An empirical study. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 46–60. Springer, Heidelberg (2004)
12. Zuniga, M., Krishnamachari, B.: Analyzing the transitional region in low power wireless links. In: First IEEE International Conference on Sensor and Ad hoc Communications and Networks (SECON), pp. 517–526. IEEE, Los Alamitos (2004)

A Automatically-Generated PRISM Model for a Four-Node Gossiping Network

```
probabilistic

const double PsendingNode0 = 1.0;
const double PsendingNode1 = 1.0;
const double PsendingNode2 = 1.0;
const double PsendingNode3 = 1.0;

const double linRxSignal_0_1 = 1.5112050684404692E-9;
const double linRxSignal_0_2 = 1.5112050684404692E-9;
const double linRxSignal_0_3 = 1.0345994570907724E-8;
const double linRxSignal_1_0 = 1.5112050684404692E-9;
const double linRxSignal_1_2 = 1.6211305024389717E-9;
const double linRxSignal_1_3 = 3.04330129123453E-8;
const double linRxSignal_2_0 = 1.5112050684404692E-9;
const double linRxSignal_2_1 = 1.6211305024389717E-9;
const double linRxSignal_2_3 = 3.21510813957935E-8;
const double linRxSignal_3_0 = 1.0345994570907724E-8;
const double linRxSignal_3_1 = 3.04330129123453E-8;
const double linRxSignal_3_2 = 3.21510813957935E-8;

formula snr_0_1 = (linRxSignal_0_1*send0)/(linRxSignal_2_1*send2
+ linRxSignal_3_1*send3 + 1.0E-10);
formula snr_0_2 = (linRxSignal_0_2*send0)/(linRxSignal_1_2*send1
+ linRxSignal_3_2*send3 + 1.0E-10);
formula snr_0_3 = (linRxSignal_0_3*send0)/(linRxSignal_1_3*send1
+ linRxSignal_2_3*send2 + 1.0E-10);
formula snr_1_0 = (linRxSignal_1_0*send1)/(linRxSignal_2_0*send2
+ linRxSignal_3_0*send3 + 1.0E-10);
formula snr_1_2 = (linRxSignal_1_2*send1)/(linRxSignal_0_2*send0
+ linRxSignal_3_2*send3 + 1.0E-10);
formula snr_1_3 = (linRxSignal_1_3*send1)/(linRxSignal_0_3*send0
+ linRxSignal_2_3*send2 + 1.0E-10);
formula snr_2_0 = (linRxSignal_2_0*send2)/(linRxSignal_1_0*send1
+ linRxSignal_3_0*send3 + 1.0E-10);
formula snr_2_1 = (linRxSignal_2_1*send2)/(linRxSignal_0_1*send0
+ linRxSignal_3_1*send3 + 1.0E-10);
formula snr_2_3 = (linRxSignal_2_3*send2)/(linRxSignal_0_3*send0
+ linRxSignal_1_3*send1 + 1.0E-10);
formula snr_3_0 = (linRxSignal_3_0*send3)/(linRxSignal_1_0*send1
+ linRxSignal_2_0*send2 + 1.0E-10);
formula snr_3_1 = (linRxSignal_3_1*send3)/(linRxSignal_0_1*send0
+ linRxSignal_2_1*send2 + 1.0E-10);
formula snr_3_2 = (linRxSignal_3_2*send3)/(linRxSignal_0_2*send0
+ linRxSignal_1_2*send1 + 1.0E-10);

formula Preceive_0_1 = func(max,0,
(snr_0_1>=12.357925578002547)?func(pow,(1-0.5*func(pow,
2.71828,-0.781*snr_0_1)), 8 * 25):0);
formula Preceive_0_2 = func(max,0,
(snr_0_2>=12.357925578002547)?func(pow,(1-0.5*func(pow,
2.71828,-0.781*snr_0_2)), 8 * 25):0);
formula Preceive_0_3 = func(max,0,
(snr_0_3>=12.357925578002547)?func(pow,(1-0.5*func(pow,
2.71828,-0.781*snr_0_3)), 8 * 25):0);
formula Preceive_1_0 = func(max,0,
(snr_1_0>=12.357925578002547)?func(pow,(1-0.5*func(pow,
2.71828,-0.781*snr_1_0)), 8 * 25):0);
formula Preceive_1_2 = func(max,0,
(snr_1_2>=12.357925578002547)?func(pow,(1-0.5*func(pow,
2.71828,-0.781*snr_1_2)), 8 * 25):0);
formula Preceive_1_3 = func(max,0,
(snr_1_3>=12.357925578002547)?func(pow,(1-0.5*func(pow,
2.71828,-0.781*snr_1_3)), 8 * 25):0);
formula Preceive_2_0 = func(max,0,
(snr_2_0>=12.357925578002547)?func(pow,(1-0.5*func(pow,
2.71828,-0.781*snr_2_0)), 8 * 25):0);
```

```
formula Preceive_2_1 = func(max,0,
(snr_2_1>=12.357925578002547)?func(pow,(1-0.5*func(pow,
2.71828,-0.781*snr_2_1)), 8 * 25):0);
formula Preceive_2_3 = func(max,0,
(snr_2_3>=12.357925578002547)?func(pow,(1-0.5*func(pow,
2.71828,-0.781*snr_2_3)), 8 * 25):0);
formula Preceive_3_0 = func(max,0,
(snr_3_0>=12.357925578002547)?func(pow,(1-0.5*func(pow,
2.71828,-0.781*snr_3_0)), 8 * 25):0);
formula Preceive_3_1 = func(max,0,
(snr_3_1>=12.357925578002547)?func(pow,(1-0.5*func(pow,
2.71828,-0.781*snr_3_1)), 8 * 25):0);
formula Preceive_3_2 = func(max,0,
(snr_3_2>=12.357925578002547)?func(pow,(1-0.5*func(pow,
2.71828,-0.781*snr_3_2)), 8 * 25):0);

formula recvp0 = func(min,1,Preceive_1_0+Preceive_2_0
+Preceive_3_0);
formula recvp1 = func(min,1,Preceive_0_1+Preceive_2_1
+Preceive_3_1);
formula recvp2 = func(min,1,Preceive_0_2+Preceive_1_2
+Preceive_3_2);
formula recvp3 = func(min,1,Preceive_0_3+Preceive_1_3
+Preceive_2_3);

module node0
    active0:[0..1] init 1;
    send0:  [0..1] init 0;

[tick] send0=0&active0=1 -> PsendingNode0:(send0’=1)&(active0’=1)
+(1-PsendingNode0):(send0’=0)&(active0’=0);
[tick] send0=1&active0=1 -> send0’=0&active0’=0;
[tick] active0=0 -> send0’=0&active0’=0;
endmodule

module node1
    active1:[0..1] init 1;
    send1:  [0..1] init 0;

[tick] send1=0&active1=1 -> PsendingNode1*recvp1:(send1’=1)&
(active1’=1)+(1-PsendingNode1)*recvp1:(send1’=0)&(active1’=0)+(1-
recvp1):(send1’=0)&(active1’=1);
[tick] send1=1&active1=1 -> send1’=0&active1’=0;
[tick] active1=0 -> send1’=0&active1’=0;
endmodule

module node2
    active2:[0..1] init 1;
    send2:  [0..1] init 0;
```

```
[tick] send2=0&active2=1 -> PsendingNode2*recvp2:(send2'=1)&
(active2'=1)+(1-PsendingNode2)*recvp2:(send2'=0)&(active2'=0)+(1-
recvp2):(send2'=0)&(active2'=1);
[tick] send2=1&active2=1 -> send2'=0&active2'=0;
[tick] active2=0 -> send2'=0&active2'=0;
endmodule

module node3
    active3:[0..1] init 1;  send3:  [0..1] init 0;

[tick] send3=0&active3=1 -> PsendingNode3*recvp3:(send3'=1)&
(active3'=1)+(1-PsendingNode3)*recvp3:(send3'=0)&(active3'=0)+(1-
recvp3):(send3'=0)&(active3'=1);
[tick] send3=1&active3=1 -> send3'=0&active3'=0;
[tick] active3=0 -> send3'=0&active3'=0;
endmodule
```

Reasoning about System-Degradation and Fault-Recovery with Deontic Logic

Pablo F. Castro and T.S.E. Maibaum

McMaster University
Department of Computing & Software
Hamilton, Canada
castropf@mcmaster.ca, tom@maibaum.org

Abstract. In this paper we outline the main characteristics of a deontic logic, which we claim is useful for the modeling of and reasoning about fault-tolerance and related concepts. Towards this goal, we describe a temporal extension of this formalism together with some of its properties. We use two different examples to show how some fault-tolerance concepts (like fault-recovery and system degradation) can be expressed using deontic constructs. The second example demonstrates how contrary-to-duty reasoning (when a secondary obligation arises from the violation of a primary obligation) is applied in fault-tolerant scenarios.

Keywords: Fault-Tolerance, Formal Specification, Deontic Logics, Software Design.

1 Introduction

Fault-tolerance has emerged as an important research field in recent years; the increasing complexity of software code in current applications has implied that techniques such as program verification (e.g., Hoare logic) are very expensive to apply in practice, in part because the total elimination of errors is a hard task in large programs. This implies that designers have to find other techniques to develop critical software, which could be used together with formal verification.

Producing fault-tolerant programs (software which is able to recover from errors) is an interesting option. Although the main techniques for fault-tolerance have been proposed for the implementation phase, in the past few years some techniques and formalisms have been proposed for the design phase. We wish to propose techniques for use in the more abstract design stage and we intend to take some steps towards this goal; in this paper we introduce a propositional deontic logic (a detailed introduction is given in [1] and [2]) to specify and to reason about fault-tolerance at the design level, and then we add some features to the logic to use it in some examples with fault-tolerance features (we introduce a temporal extension of the logic).

We give two examples of application of this formal system; in the first example we show how the basic constructs of the logic can be used to specify systems where violations arise, and therefore notions such as *fault-recovery* and *system*

M. Butler et al. (Eds.): Fault Tolerance, LNCS 5454, pp. 25–43, 2009.
© Springer-Verlag Berlin Heidelberg 2009

degradation should be formalized; we show in what manner deontic logic can be used for this. On the other hand, the second example is more complex, and we introduce it showing that the notion of *violation* (and some properties related to it) can be embedded in the logical system, which makes the logic more appealing for use in fault-tolerance. In addition, contrary-to-duty formulae (which are inherent in fault-tolerance) impose some relationships between the different violations in the model, as we illustrate later on.

Although deontic logics (or DL for short) were created to formalize moral and ethical reasoning, they have been proposed as a suitable formalism for dealing with fault-tolerance by several authors (for example: [3], [4], [5] and [6]). This logic has the main benefit of allowing us to distinguish between qualitatively different scenarios: normative (following the rules, expected, normal) and non-normative (violating the rules, unexpected, abnormal) situations. In these logics we have some new predicates: P (*permission*), O (*obligation*) and F (*forbidden*) which allow us to add norms in the logic. We find different definitions of these predicates in the literature, and there is no standard definition (and therefore properties) of permission, obligation and forbidden. (A good introduction to deontic logic is given in [7].) The logic that we introduce below can be classified as an "*ought-to-do*" deontic logic, because the deontic operators are applied to actions (i.e., we impose norms on actions). Meanwhile, in other deontic logics the norms are on predicates (e.g, *it is forbidden that the housed is painted in black*).

As noted in several sources ([8], [9], [3] and [10]), ought-to-do logics seem to be more suitable for use in specifying computing systems. However, it is hard to find a suitable version of deontic logic to apply in practice. Some formalisms have been described for use in computer science (for example:[8] or [11]), but most of them are not designed to be used in the context of fault-tolerance. (Some work has been done about databases and fault-tolerance in [6].) In addition, the notion of time has been useful for reasoning about program properties, so we mix both deontic notions and temporal frameworks; and the result is the logic described in the next section, which is very expressive, allowing us to express several properties, like those related to fault recovery.

Contrary-to-duty statements (a set of predicates where a secondary obligation arises from the violation of a primary one) have been studied in the deontic community (see [12], [13] and [14]), partly because we can obtain "*paradoxical*" (contrary to our intuition) properties from these statements, for example: *you should not tell the secret to Reagan or to Gorbachov; if you tell the secret to Gorbachov, you should tell the secret to Reagan. You tell the secret to Gorbachov.* (This is called the Reagan-Gorvachov paradox in the literature [15].) In several deontic logics, we can obtain a logical contradiction from these statements, which, according to our intuition, lacks sense. We argue in section 4 that these kinds of reasoning are common in fault-tolerance, and we ground this claim with an example, and then we describe an extension of the logic to deal with contrary-to-duty reasoning.

The paper is organized as follows. In section 2 we present a brief description of our deontic logic. In sections 3 and 4 we describe two examples and an extension of the logic described ealier. Finally, we present some conclusions and further work.

2 A Temporal Deontic Logic

The logic presented in this section takes some features from the *dynamic deontic logic* described by Meyer in [8], and the *modal action logic* proposed by Maibaum and Khosla in [10]. In the language, we have a set of atomic (or primitive) actions:

$$\Delta_0 = \{\alpha, \beta, \gamma, ...\}$$

and a set of atomic propositions:

$$\Phi_0 = \{\varphi, \psi, \vartheta, ...\}$$

More complex actions can be constructed from the atomic ones using the following operators: $\sqcup, \sqcap, -$, that is: non-deterministic choice, concurrent execution and action complement. In addition, we consider two special actions: $\emptyset$ and $\mathbf{U}$. The former is an impossible action (an action that cannot be executed), while the latter is universal choice (the non-deterministic choice between the enabled actions). The complement operator is particularly useful to specify wrong behavior, for example, to say that if a given system is obliged to perform an action and it executes another action (this action is in the complement), then we have an error. We must be very careful with the complement because it can induce undecidability in the logic (for example, if we combine it with the iteration operator, see [9]).

The intuition behind each construct in the logic is as follows:

- $\alpha =_{act} \beta$: *actions α and β are equal.*
- $[\alpha]\varphi$: *after any possible execution of α, φ is true.*
- $[\alpha \sqcup \beta]\varphi$: *after the non-deterministic execution of α or β, φ is true.*
- $[\alpha \sqcap \beta]\varphi$: *after the parallel execution of α and β, φ is true.*
- $[\mathbf{U}]\varphi$: *after the non-deterministic choice of any possible action, φ is true.*
- $[\emptyset]\varphi$: *after executing an impossible action, φ becomes true.*
- $[\overline{\alpha}]\varphi$: *after executing an action other than α, φ is true.*
- $\mathsf{P}(\alpha)$: *every way of executing α is allowed.*
- $\mathsf{P_w}(\alpha)$: *some way of executing α is allowed.*

The deontic part of the logic is given by the permission predicates. Note that we consider two different versions of permission, namely strong permission $\mathsf{P}(-)$ and weak permission $\mathsf{P_w}(-)$. Both are useful, in particular since we can define an obligation operator using them:

$$\mathsf{O}(\alpha) \overset{\mathsf{def}}{\iff} \mathsf{P}(\alpha) \wedge \mathsf{P_w}(\overline{\alpha})$$

That is, an action is obliged if it is strongly permitted (i.e., every way of doing it is allowed) and the remaining actions are not weakly permitted (you are not allowed to execute any of them in any context).

This definition of obligation allows us to avoid several deontic paradoxes (some of the most well-known paradoxes in deontic logic are described in [12]). An example is the so-called Ross's paradox: *if we are obliged to send a letter then we are obliged to send it or burn it.* This is a paradox in the sense that we do not expect this sentence to be valid in natural language. The formalization of this paradox is as follows: $\mathsf{O}(send) \rightarrow \mathsf{O}(send \sqcup burn)$. The reader can verify later that this formula is not valid in our framework.

The semantics of our logic is defined by a labelled transition system, $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \mathcal{P} \rangle$, where:

- $\mathcal{W}$ is a (non empty) set of worlds.
- $\mathcal{E}$ is set of events (each event corresponds to a set of actions that may occur during system execution).
- $\mathcal{R}$ is a $\mathcal{E}$-labelled relation between worlds.
- $\mathcal{I}$ is an interpretation which tells us which propositions are true in which world; in addition, it maps an action to a set of events (the events that this action participates in during any of its executions).
- the relation $\mathcal{P} \subseteq \mathcal{W} \times \mathcal{E}$ tells us which event is allowed in a given world.

Some restriction must be imposed on the models to make it possible to axiomatize the logic; we have introduced these technical details in [2]. The most important of these restrictions says that maximal parallel execution of actions must produce an unique event.

The relation $\vDash$ can be defined in a standard way; we have two novel rules for the two versions of permission:

- $w, M \vDash p \overset{\mathsf{def}}{\Longleftrightarrow} w \in \mathcal{I}(p)$
- $w, M \vDash \alpha =_{act} \beta \overset{\mathsf{def}}{\Longleftrightarrow} \mathcal{I}(\alpha) = \mathcal{I}(\beta)$
- $w, M \vDash \neg\varphi \overset{\mathsf{def}}{\Longleftrightarrow}$ not $w \vDash \varphi$.
- $w, M \vDash \varphi \rightarrow \psi \overset{\mathsf{def}}{\Longleftrightarrow} w \vDash \neg\varphi$ or $w \vDash \psi$ or both.
- $w, M \vDash [\alpha]\phi \overset{\mathsf{def}}{\Longleftrightarrow}$ for every $e \in \mathcal{I}(\alpha)$ and $w' \in \mathcal{W}$ if $w \xrightarrow{e} w'$, then $w', M \vDash \phi$.
- $w, M \vDash \mathsf{P}(\alpha) \overset{\mathsf{def}}{\Longleftrightarrow}$ for all $e \in \mathcal{I}(\alpha)$, $\mathcal{P}(w, e)$ holds.
- $w, M \vDash \mathsf{P_w}(\alpha) \overset{\mathsf{def}}{\Longleftrightarrow}$ there exists some $e \in \mathcal{I}(\alpha)$ such that $\mathcal{P}(w, e)$.

Here M denotes a model and w a world of that model. These definitions are a formalization of the intuition explained above. Note that using modalities we can define the composition of actions, that is:

$$[\alpha; \beta]\varphi \overset{\mathsf{def}}{\Longleftrightarrow} [\alpha]([\beta]\varphi)$$

However, we introduce it only as notation. (We can introduce this operator into the language, but it complicates in several ways the semantics, in particular the semantics of the action complement.)

We can explain intuitively the deontic operators with some diagrams. Consider the following model M in figure 1. The dotted arrow means that this transition is not allowed to be performed. e_1, e_2 and e_3 represent possible events during the

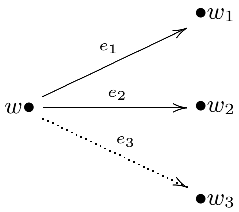


Fig. 1. Example of a model

execution of the system; we can suppose that they are generated by two actions: α and β. Suppose that α produces (during its execution) events e_1 and e_2, and action β produces event e_3. Here we have $w, M \vDash \mathsf{P}(\alpha)$, because every way of executing it is allowed, and also we have $w, M \vDash \mathsf{P_w}(\alpha)$, because α is allowed to be executed in at least one way. On the other hand, we have $w, M \vDash \neg\mathsf{P}(\beta)$ and also $w, M \vDash \neg\mathsf{P_w}(\beta)$. Finally, since $w, M \vDash \mathsf{P}(\alpha)$ and $w, M \vDash \neg\mathsf{P_w}(\overline{\alpha})$ we obtain $w, M \vDash \mathsf{O}(\alpha)$.

Some interesting properties of the modal and deontic operators are the following:

P1. $[\alpha \sqcup \alpha']\varphi \leftrightarrow [\alpha]\varphi \wedge [\alpha']\varphi$
P2. $[\alpha]\varphi \rightarrow [\alpha \sqcap \alpha']\varphi$
P3. $\mathsf{P}(\emptyset)$
P4. $\mathsf{P}(\alpha \sqcup \beta) \leftrightarrow \mathsf{P}(\alpha) \wedge \mathsf{P}(\beta)$
P5. $\mathsf{P}(\alpha) \vee \mathsf{P}(\beta) \rightarrow \mathsf{P}(\alpha \sqcap \beta)$
P6. $\neg\mathsf{P_w}(\emptyset)$
P7. $\mathsf{P_w}(\alpha \sqcup \beta) \leftrightarrow \mathsf{P_w}(\alpha) \vee \mathsf{P_w}(\beta)$
P8. $\mathsf{P_w}(\alpha \sqcap \beta) \rightarrow \mathsf{P_w}(\alpha) \wedge \mathsf{P_w}(\beta)$

P1 says that, *if after executing α or β, φ is true, then φ is true after executing α and after executing β*. P2 says that parallel composition preserves postcondition properties. P3 says that every way of executing the impossible action is allowed (because there is no ways of executing it!). P4 and P5 are similar to P1 and P2 but for strong permission. P6, P7 and P8 are the dual properties for the weak permission. In particular, P6 says that the impossible action is not weakly permitted, i.e., there is no (allowed) way of executing it. It is in this sense that $\emptyset$ is the impossible action. This explains the seemingly paradoxical nature of P3: every way of executing the impossible action is allowed (but there is no way!).

Note that we do not have, as in dynamic logic, the iteration as an operation over actions. Even though it is desirable, it will bring us undecidability. Instead, we prefer to enrich our logic with temporal operators in a branching time style (precisely, similar to Computational Tree Logic [16]). We consider the following temporal formulae:

- $\mathsf{AN}\varphi$ (*in all possible executions φ is true at the next moment*).
- $\mathsf{AG}\varphi$ (*in all executions φ is always true*),

- $\mathsf{A}(\varphi_1 \,\mathcal{U}\, \varphi_2)$ (*for every possible execution* φ_1 *is true until* φ_2 *becomes true*)
- $\mathsf{E}(\varphi_1 \,\mathcal{U}\, \varphi_2)$ (*there exists some execution where* φ_1 *is true until* φ_2 *becomes true*).

As usual, using these operators we can define their dual versions. It is interesting to note that iteration and the temporal operators are related; with iteration we can define: $[\mathbf{U}^*]\varphi \overset{\mathsf{def}}{=} \mathsf{AG}\varphi$. But the temporal formulae do not make the logic undecidable because the temporal operators cannot be mixed with the modal ones.

In addition, we consider the operator $\mathsf{Done}(\alpha)$, which means *the last action executed was* α. Using it, together with the temporal operators, we can reason about the executions of our models. Some useful formulae can be expressed using the $\mathsf{Done}()$ operator; some examples are:

- $\mathsf{ANDone}(\alpha)$, *the next action to be executed will be* α.
- $\mathsf{Done}(\alpha) \rightarrow \mathsf{Done}(\beta)$, *the execution of* α *implies the execution of* β
- $\mathsf{Done}(\alpha) \rightarrow \mathsf{O}(\beta)$, *if you performed* α *then you are (now) obliged to perform* β.
- $\mathsf{A}(\mathsf{Done}(\alpha_1 \sqcup ... \sqcup \alpha_n) \,\mathcal{U}\, \mathsf{Done}(\beta))$, *on every path you perform some* α_i *at each step until you perform* β.

Some of these formulae are important to express error-recovery, as we illustrate in the next section.

The $\mathsf{Done}(-)$ operator has some interesting properties:

Done1. $\mathsf{Done}(\alpha \sqcup \beta) \rightarrow \mathsf{Done}(\alpha) \vee \mathsf{Done}(\beta)$
Done2. $\mathsf{Done}(\alpha \sqcap \beta) \leftrightarrow \mathsf{Done}(\alpha) \wedge \mathsf{Done}(\beta)$
Done3. $\mathsf{Done}(\alpha \sqcup \beta) \wedge \mathsf{Done}(\overline{\alpha}) \rightarrow \mathsf{Done}(\beta)$
Done4. $[\alpha]\varphi \wedge [\beta]\mathsf{Done}(\alpha) \rightarrow [\beta]\varphi$

Property **Done1** says that if a choice between two actions was executed then one of them was executed. **Done2** means that if we execute the parallel composition of two actions then we have to perform both actions. **Done3** allows us to discover which action of a choice was executed. And the last property is a kind of subsumption property: *if, after executing* α φ *is true, and after doing* β, α *was also done, then after* β φ *is also true.*

The semantics of the temporal operators can be defined using traces (as usual). Suppose that $\pi = s_0 \overset{e_0}{\rightarrow} s_1 \overset{e_1}{\rightarrow} ...$ is an infinite trace on a given model (note that we can extend the finite traces to infinite ones, as is usually done in temporal logics). We says that $\pi' \preceq \pi$ if π' is an initial segment of π. Then we define the formal semantics of the temporal operators as follows:

- $\pi, i, M \vDash \mathsf{Done}(\alpha) \overset{\mathsf{def}}{\iff} i > 0$ and $e_{i-1} \in \mathcal{I}(\alpha)$.
- $\pi, i, M \vDash \mathsf{AN}\varphi \overset{\mathsf{def}}{\iff} \forall \pi'$ such that $\pi[0,i] \preceq \pi'$, we have that $\pi', i+1, M \vDash \varphi$.
- $\pi, i, M \vDash \mathsf{AG}\varphi \overset{\mathsf{def}}{\iff} \forall \pi'$ such that $\pi[0,i] \preceq \pi'$, we have that $\forall j \geq i : \pi', j, M \vDash \varphi$.
- $\pi, i, M \vDash \mathsf{A}(\varphi_1 \,\mathcal{U}\, \varphi_2) \overset{\mathsf{def}}{\iff} \forall \pi'$ such that $\pi[0,i] \preceq \pi'$, we have that $\exists j \geq i : \pi', j, M \vDash \varphi_2$ and $\forall i \leq k < j : \pi', k, M \vDash \varphi_1$.

- $\pi, i, M \vDash \mathsf{E}(\varphi_1 \; \mathcal{U} \; \varphi_2) \overset{\mathsf{def}}{\Longleftrightarrow} \exists \pi'$ such that $\pi[0,i] \preceq \pi'$, we have that $\exists j \geq i$: $\pi', j, M \vDash \varphi_2$ and $\forall i \leq k < j : \pi', k, M \vDash \varphi_1$.

Note that the relation $\vDash$ is now defined with respect to a sequence, an instant and a model, that is, $\pi, i, M \vDash \varphi$ means that the formulae φ is true at instant i of the execution π of the model M.

It is important to mention that this formal system is suitable to be used with a tableaux deduction system (see [17]), which will enable us to do automatic deduction in relation to our specifications.

3 A Practical Example

We will use a small example to illustrate why the deontic operators are useful to model fault-tolerance:

Example 1. In a factory which produces some kind of object, the process of making an object is as follows: we have two mechanical hands (A and B), one press and one drill; the hand A puts an element in the press and the hand B takes the pressed element and puts it in the drill. If the hand A fails and does not put some element in the press, then the hand B should put the element in the press and then it should continue doing its work. And vice-versa (if hand B fails). If both hands fail, an alarm sounds and the system is shut down.

The interesting point in the example is how a violation (when a mechanical hand fails) can be overcome using the other hand (taking advantage of the redundancy in the system); of course, using only one hand for the whole process implies a slower process of production, and therefore the entire process is more expensive.

Note that here we have an important difference between prescription and description of behavior: *the hand A should put an element in the press.* We need to model this as a prescription of behavior; that is, *what the system is obliged to do in a given situation.* One of the main advantages of deontic logic is that it allows us to distinguish between the description and prescription of a system (as established in [10]). For example, if we proceed to say that the hand A puts an element in the press (in a descriptional way):

$$\neg el2press \rightarrow \mathsf{ANDone}(A.putselpress)$$

which means that if there is no element in the press then the hand A puts one in it (note that $\mathsf{ANDone}(\alpha)$ could be thought of as a *do* operator). On the other hand, the deontic version:

$$\neg el2press \rightarrow \mathsf{O}(A.putselpresser)$$

says that, if there is no element in the press, then the hand A *should* put one in the press. The difference is that, in the second case, the obligation could be violated. Moreover, the violation becomes a state property in terms of which characterisation of faults, prescription of recovery and analysis can be defined.

Having these facts in mind, we can give a part of the specification:

A1 $\neg\mathsf{Done}(\mathbf{U}) \rightarrow \neg el2press \wedge \neg el2drill \wedge \neg v_1 \wedge \neg v_2$
A2 $(\neg el2press \rightarrow \mathsf{O}(A.putselpress)) \wedge (v_2 \wedge elpressed \rightarrow \mathsf{O}(A.putseldrill))$
A3 $(\neg el2drill \wedge elpressed \rightarrow \mathsf{O}(B.putseldrill)) \wedge (v_1 \rightarrow \mathsf{O}(B.putselpress))$
A4 $\neg v_1 \wedge \mathsf{O}(A.putselpress \sqcup A.putseldrill) \rightarrow [\overline{A.putselpress \sqcup A.putseldrill}]v_1$
A5 $\neg v_2 \wedge \mathsf{O}(B.putselpress \sqcup B.putseldrill) \rightarrow [\overline{B.putselpress \sqcup B.putseldrill}]v_2$
A6 $\neg v_1 \rightarrow [A.putselpress \sqcup A.putseldrill]\neg v_1$
A7 $\neg v_2 \rightarrow [B.putselpress \sqcup B.putseldrill]\neg v_2$
A8 $(v_1 \rightarrow [\overline{A.fix}]v_1) \wedge (v_1 \rightarrow [A.fix]\neg v_1)$
A9 $(v_2 \rightarrow [\overline{B.fix}]v_1) \wedge (v_2 \rightarrow [B.fix]\neg v_2)$
A10 $v_1 \wedge v_2 \rightarrow \mathsf{ANDone}(alarm)$
A11 $[alarm]\mathsf{AF}(\mathsf{Done}(A.fix \sqcap B.fix))$

Some explanation will be useful about the axioms. We have only shown the deontic axioms; some other axioms should be added (for example frame axioms and pre/post condition axioms for individual actions). Axiom **A1** establishes the initial condition in the system: at the beginning (when no action has occurred) there is no element to press, and no element to drill, and no violations. **A2** says that if there is no element in the press, then the hand A should put an element there; in addition, it says that if the hand B is not working, then A has to put pressed elements in the drill. **A3** says that if there is no element to drill and there exists a pressed element, then the hand B should put that element in the drill. Axiom **A4** expresses when a violation of type v_1 is committed: if there is no violation v_1 and hand A is obliged to put an element in the press, but the hand does not do it, then v_1 becomes true. **A5** is the specification of violation v_2: it happens when the hand B does not fulfill its obligation. **A6** and **A7** model when normal states are preserved. **A8** and **A9** express when we can recover from violation, that is, when some hand is repaired. Finally, **A10** and **A11** tell us when the worst situation is achieved, that is, when both hands are in violation; then an alarm is initiated and the hands are repaired.

It is interesting to analyze the different faults that we can have in the system; we can see that there exists an order relation between them. The situation is illustrated in figure 2. The ideal scenario is when $\neg v_1 \wedge \neg v_2$ is true, that is, when no hand is faulty. From here, the system can suffer a degradation and then it goes to violation 1 (v_1 is true) or violation 2 (v_2 is true); both situations of violation are incomparable, in the sense that none of them implies the other. Then, the system can be degraded again and both violations hold; in this case, both hands are not working correctly and there is no other option than repairing both hands. Otherwise, the entire process of production will be effected. It is important to note that, though in violation 1 (or violation 2) the system can work correctly (because of the redundancy of the hands), though the process of production is slower (only one hand will do all the work).

On the other hand, from violations $v_1 \wedge v_2$, $v_1 \wedge \neg v_2$ or $\neg v_1 \wedge v_2$ the system can upgrade itself (going to a better state) when the faulty hand is repaired. The structure in figure 1 can be thought of as a *lattice of violations* which characterise the possible violations in the system, and the relationships between them.

Note that the hand A can try to put an element in the drill. Indeed, if hand B is working correctly, this scenario is obviously not desirable. We can use the

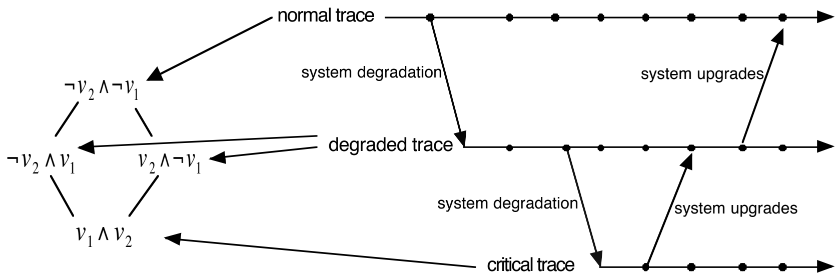


Fig. 2. ordering violations

forbidden operator to avoid this. The forbidden operator can be defined using the weak permission or the strong permission; depending on the choice, we get different results. We define it using the weak permission, as follows:

$$\mathsf{F}(\alpha) \overset{\mathsf{def}}{\Longleftrightarrow} \neg \mathsf{P}_{\mathsf{w}}(\alpha)$$

That is, an action is forbidden if it is not allowed to be performed. Using this operator we can include the following formulae:

- $\neg v_1 \rightarrow \mathsf{F}(A.putseldrill)$
- $\neg v_2 \rightarrow \mathsf{F}(B.putselpress)$

Using these new formulae we can define new violations in the case that prohibitions are violated, and therefore, the corresponding recovery actions can be introduced.

Some properties can be proved from the specification. In particular, some interesting properties to prove are:

$$\mathsf{AG}(\neg v_1 \wedge \neg v_2) \wedge \neg el2drill \wedge \mathsf{AF}el2press \rightarrow \mathsf{AF}el2drill$$

if there is no violation, and eventually we have an element to press, then we will have an element to drill.

$$\mathsf{AG}(v_1 \wedge \neg v_2) \wedge \neg el2drill \wedge \mathsf{AF}el2press \rightarrow \mathsf{AF}el2drill$$

if there is a violation of type v_1 (but no violation of type v_2), then the pressed elements will be brought to the drill, that is, the system continues working, in a degraded way.

$$\mathsf{AG}(v_1 \wedge v_2) \rightarrow \mathsf{AF}(\mathsf{EG}(\neg el2drill \wedge \neg el2press))$$

if both hands are not working correctly, then there exists the possibility that the elements will not be transported to the press or to the drill.

Of course, a lot of interesting different properties can be proposed, and proven. We have described another example of an application in [1]. The point to make here is the way in which system violation and fault recovery are specified; we can mix modal and deontic operators to specify these system properties. And the expressiveness that temporal operators give us allow us to prove important properties about the specification.

We note that the logic described is decidable and, therefore, techniques such as model checking could be used to validate specifications and to prove properties of corresponding programs.

4 Contrary-to-Duty Reasoning and Fault-Tolerance

As the reader may observe in the example given in section 3, we have to specify when violations occur. A good question is whether we can do this directly in the logic, observing that if an obligation is not fulfilled then a violation follows; the problem with this point of view is that we have usually different violations in our specification, and when each violation occurs should be a design decision. Another problem is that, as argued in [18], there are cases where a red (forbidden) transition does not yield a violation (or a faulty state); the example presented in the referenced work is when we have two different components (or "agents") and the component 1 is allowed to perform an action a which yields a green state (i.e., without violations), and the component 2 is forbidden to perform an action which yields the same state; in this situation, the status of the action (if it is allowed or not) does not follow from the status of the resulting state. In other words, red transition can yield green (normal) states.

However, there is a key observation that we can make:

- *If we are not in a violation and we perform an allowed action, we will not produce a violation.*

In other words, allowed actions preserve absence of violations. In several approaches to deontic action logic ([8] and [9]), the deontic predicates are reduced to modalities; for example in [8] we have:

$$\mathsf{F}(\alpha) \equiv [\alpha]\mathsf{V}$$

That is, an action is forbidden if and only if it produces a violation (here V is a predicate which indicates when a violation occurs). As we argued before, we want to separate the prescription and description of systems. And these kinds of definitions introduce a strong relationship between the two. In particular, we observe that (following this definition) an allowed action (that is: $\neg\mathsf{F}(\alpha)$) implies that there is a way to execute the action so that it terminates and does not cause a violation. We reject this (in the context of computing systems) because whether an action must finish or not should be specified in the descriptional part of the specification.

For example, consider a scenario where a component is allowed to perform an action, but it cannot perform that action because it is waiting for some resource; we do not want to impose that there is some way of executing the action. Of course, that we should keep the permission over time, it is the task of the specifier (or software engineer) to ensure. Summarizing, we want to establish a minimal relationship between deontic predicates and modalities, providing more freedom for the designer. This basic relationship can be summarized in the following semantical condition on the models.

C1 $w, M \vDash \neg \mathsf{V}$ and $\mathcal{P}(e, w')$ and $w \xrightarrow{e} w'$ implies $w', M \vDash \neg \mathsf{V}$

The condition characterises the idea that allowed actions do not introduce violations (but perhaps they may carry on violations); this requirement is called the *GGG* (green-green-green) condition in [18].

Some intuition about these facts is illustrated in figure 3, where dashed lines denote forbidden events in the model. In that example, if we perform event e_1 then we do not get a violation, because we perform an allowed action from a state which is free of "errors"; but if we perform event e_2, we get a violation. On the other hand, if we perform an allowed action from a state with violations, we can get a violation (as illustrated in figure 4). In other words, whether an allowed action preserves a violation or not must be specified by the designer. Those actions which take an error state to a state free of errors are called *recovery actions.* Note that, if in our specification we say (perhaps indirectly) that an allowed action introduces a violation, then this action can only be executed in error states (otherwise it will be inconsistent with condition **C1**).

Some remarks are needed regarding the predicate V, which indicates when a violation is produced; in complex applications, we can divide V into several violations (as was done in the example presented in section 3) setting:

$$\mathsf{V} = v_1 \vee ... \vee v_n$$

i.e., V is the disjunction of all the possible violations. Note that with this definition a recovery action for a violation v_i can produces a new violation v_j; however, it cannot produce the same violation, otherwise it will never recover from the error! Following this, an important property to prove in our specifications is that recovery actions do not produce again the same violation (otherwise our design will be inconsistent).

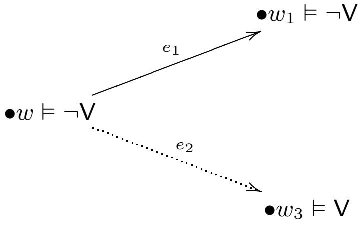

$\bullet w_1 \vDash \neg \mathsf{V}$
e_1
$\bullet w \vDash \neg \mathsf{V}$
e_2
$\bullet w_3 \vDash \mathsf{V}$

Fig. 3. Example of model with violations

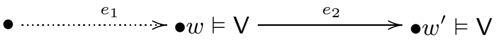


Fig. 4. An example of allowed action which carries forward violations

Condition **C1** can be established by means of predicates, as follows:

$$\neg \mathsf{V} \wedge \mathsf{P}(\alpha) \rightarrow [\alpha]\neg \mathsf{V}$$

We can think of this formula as an extra axiom in the logic, which is sometimes useful to derive properties; an example is given below. We can refine this condition, establishing that permitted actions do not introduce new violations; e.g., suppose that we are in a state where some of the violation predicates are true, then executing a permitted action will not introduce new violations, perhaps the system will not recover from the violations already present in the actual state, but this action will not take us to a worse state. This refined version of $\mathsf{C1}$ (called $\mathsf{C1}'$) is expressed by the following set of formulae:

$$\neg v_i \wedge \mathsf{P}(\alpha) \rightarrow [\alpha]\neg v_i$$

Note that we have one formula for each predicate v_i. Obviously, condition **C1′** implies condition **C1**.

On the other hand, *contrary-to-duty* structures are a set of predicates where, from a violation of a primary obligation, a secondary obligation arises. These kinds of formulae have been problematic in deontic logic because sometimes from, intuitively correct, contrary-to-duty predicates we can deduce falsehood, which is paradoxical in some sense (see [12] for a detailed discussion of contrary-to-duty paradoxes). For example, consider the following predicates (a statement of the *gentle murderer* paradox, see [12] for details):

– It is forbidden to kill.
– If you kill, then you have to kill gently.
– You kill.

In standard deontic logic ([7]), these predicates are inconsistent. In the logic described above we can formalize it as follows:

– $\mathsf{F}(\mathit{kill})$
– $\mathsf{Done}(\mathit{kill}) \rightarrow \mathsf{O}(\mathit{kill\_gently})$
– $\mathsf{Done}(\mathit{kill})$

Using modus ponens and the third and second sentences we get: $\mathsf{O}(\mathit{kill\_gently})$ and from the first sentence we have: $\mathsf{F}(\mathit{kill})$, which is equivalent to $\neg\mathsf{P}_\mathsf{w}(\mathit{kill})$. And using the definition of obligation, we also obtain $\mathsf{P}(\mathit{kill\_gently})$. Considering that $\mathit{kill\_gently} \sqsubseteq \mathit{kill}$ (*killing gently is one of the ways to kill*), we have that $\mathsf{P}(\mathit{kill\_gently})$ and $\neg\mathsf{P}_\mathsf{w}(\mathit{kill\_gently})$, and the only way to have this situation in a model is when $\mathit{kill\_gently} =_{act} \emptyset$, i.e., killing in a gentle way is impossible. This is an unexpected property derived from the specification, and in some sense it is paradoxical. One way to solve this problem (proposed in [12] for *dynamic deontic logic*) is to have a convenient collection of different permissions: $\mathsf{P}^1, ..., \mathsf{P}^n$ and $\mathsf{P}^1_w, ..., \mathsf{P}^n_w$, every pair i of weak and strong permission related as explained in section 2. Using these new predicates we can formalize the problem as follows:

- $\mathsf{F}^1(\mathit{kill})$
- $\mathsf{Done}(\mathit{kill}) \rightarrow \mathsf{O}^2(\mathit{kill\_gently})$
- $\mathsf{Done}(\mathit{kill})$

where the different violations in this scenario can be specified by:

- $\mathsf{F}^1(\mathit{kill}) \rightarrow [\mathit{kill}]v_1$
- $\mathsf{F}^2(\mathit{kill\_gently}) \rightarrow [\mathit{kill\_gently}]v_2$
- $\neg v_2 \rightarrow [\overline{\mathit{kill\_gently}}]\neg v_2$

That is, killing gently produces violation v_2 (and also v_1 since it is a way of killing), and killing ungently produces $v_1 \wedge \neg v_2$, which can be thought of as a worse violation, one which perhaps we cannot recover from. As we point out above, contrary-to-duty predicates introduce a sort of additional dimension in the structure of the violations, each v_i could be divided in several violations, and therefore giving us a matrix of two dimensions (or maybe n-dimensions when we have nested contrary-to-duty reasoning). We illustrate this with an example later on.

Let us introduce the changes in the logic to manage contrary-to-duty structures; in dynamic deontic logic, the introduction of different permissions is easier because we only need to add different violation markers for each new permission. For our logic we must add some structure in our semantic models. The new semantics of the logic is defined by a labelled transition system, $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \mathcal{P}^1, ..., \mathcal{P}^n \rangle$, where:

- $\mathcal{W}$ is a (non empty) set of worlds.
- $\mathcal{E}$ is set of events (the set of events that occurs during system execution).
- $\mathcal{R}$ is a $\mathcal{E}$-labelled relation between worlds.
- $\mathcal{I}$ is an interpretation which tells us which propositions are true in which world; in addition, it maps an action to a set of events (the events that this action participates in during its execution).
- each relationship $\mathcal{P}^i \subseteq \mathcal{W} \times \mathcal{E}$ tells us which event is allowed in a given world for permission number i.

Intuitively, $\mathcal{P}^1, ..., \mathcal{P}^n$ are relations that are the interpretation of the different permissions in the logic. In the language of the logic we have to consider the predicates:

$$\mathsf{P}^1(-), ..., \mathsf{P}^n(-), \mathsf{P}^1_w(-), ..., \mathsf{P}^n_w(-)$$

The number of (strong and weak) permissions may change depending on the scenario where the logic will be used. Using these predicates, we define n different obligation predicates O^i as follows:

$$\mathsf{O}^i(\alpha) \Leftrightarrow \mathsf{P}^i(\alpha) \wedge \neg \mathsf{P}^i_w(\overline{\alpha})$$

In the same way, conditions **C1** and **C1′** can be reformulated in this new setting. Let us present an example to illustrate this new logic in practice.

Example 2. Consider a microprocessor which is part of a critical system (perhaps in a space station, where it is not easy to replace it); we have two coolers to keep the temperature of the processor low, and also we have a sensor to measure the temperature. The processor could be in a normal state (that is, working correctly) or on stand by; the latter could occur when the processor is too hot, maybe because the coolers are not working. It is forbidden that the processor is on stand by because this can produce some incorrect behavior in the system.

We see that we have standard violations (when the coolers are not working), and also a contrary-to-duty scenario: the processor is forbidden to be on stand by, but if the temperature is too high (because of the bad behavior of some cooler), then we should put the processor on stand by. The vocabulary of the example is given by the following set of actions and predicates with their intuitive meaning:

- $c_1.start$, turn on cooler 1.
- $c_2.start$, turn on cooler 2.
- $c_1.stop$, cooler 1 stops working.
- $c_2.stop$, cooler 2 stops working.
- $p.sb$, the processor goes into stand by.
- $p.up$, the processor wakes up.
- $s.getshigh$, the sensor detects high temperature.
- $s.getslow$, the sensor detects low temperature.

and predicates:

- $p.on$, the processor is working.
- $s.high$, the sensor is detecting high temperature.
- v_1, a violation is produced because cooler 1 should be working and it is off.
- v_2, similar than v_1 but produced by cooler 2.
- v_3, a violation is produced because the processor is on stand by.

The following are some of the axioms of the specification:

Ax1 $\neg\mathsf{Done}(\mathbf{U}) \to \neg v_1 \wedge \neg v_2 \wedge \neg v_3 \wedge p.on \wedge \neg s.high \wedge \neg c_1.on \wedge \neg c_2.on$

At the beginning (of time) there are no violations, the processor is working, the sensor is low, and the coolers are off.

Ax2 $\mathsf{F}^1(p.sb)$

It is forbidden that the processor goes into stand by.

Ax3 $\neg s.high \to \mathsf{P}^i(\overline{p.sb})$ (for $i = 1, 2$)

If the sensor is low, then every action, different from putting the processor in stand by, is allowed.

Ax4 $s.high \to \mathsf{O}^1(c_1.on \sqcap c_2.on)$

If the sensor is detecting high temperature, then the two coolers should be on.

Ax5 $s.high \wedge v_1 \wedge v_2 \rightarrow \mathsf{O}^2(p.sb)$

If the sensor is detecting a high temperature and both coolers are not working, then the processor ought to go into stand by.

Ax6 $(\neg v_i \wedge \mathsf{F}^1(\overline{c_1.on}) \rightarrow [\overline{c_1.on}](v_1 \wedge \neg v_i))$
$\wedge(v_i \wedge \mathsf{F}^1(\overline{c_1.on}) \rightarrow [\overline{c_1.on}](v_1 \wedge v_i))$ (for $i = 2, 3$)
Ax7 $(\neg v_i \wedge \mathsf{F}^1(\overline{c_2.on}) \rightarrow [\overline{c_2.on}](v_2 \wedge \neg v_i))$
$\wedge(v_i \wedge \mathsf{F}^1(\overline{c_2.on}) \rightarrow [\overline{c_2.on}](v_2 \wedge v_i))$ (for $i = 1, 3$)
Ax8 $(\neg v_i \wedge \mathsf{F}^2(ps.sb) \rightarrow [ps.sb](v_3 \wedge \neg v_i))$
$\wedge(v_i \wedge \mathsf{F}^2(ps.sb) \rightarrow [ps.sb](v_3 \wedge v_i))$ (for $i = 1, 2$)
Ax9 $(v_1 \rightarrow [\overline{c_1.on}]v_1) \wedge ([c_1.on]\neg v_1)$
Ax10 $v_2 \rightarrow [\overline{c_2.on}]v_2 \wedge ([c_2.on]\neg v_2)$
Ax11 $v_3 \rightarrow [\overline{p.up}]v_3 \wedge ([p.up]\neg v_3)$

Formulae **Ax6**, **Ax7** and **Ax8** define what forbidden actions cause each violation, **Ax9**, **Ax10** and **Ax11** define the recovery actions for each violation; although this example is simple, in more complicated examples the designer has to take care that recovery actions should not cause violations.

In addition we require that if both coolers are off and the sensor is high, then the processor ought to be on stand by until the sensor is low.

Ax12 $\mathsf{AO}^2(p.sb)\,\mathcal{U}\; s.low$

We add the restriction that if both coolers are on, then it is not possible to have a high temperature in the processor (the system is well designed in this sense).

Ax13 $c_1.on \wedge c_2.on \rightarrow s.low$

We need axioms to describe the effects of the action *p.up*.

Ax14 $[p.up]p.on$
Ax15 $\neg p.on \rightarrow [\overline{p.up}]\neg p.on$

Similar axioms must be added for the other actions; axiom **Ax15** says that no other action different from *p.up* turns on the processor.

An interesting point about this description is that having two different kinds of deontic predicates adds more structure in the violation lattice; the different violations that may occur in this specification are shown in figure 5. In this illustration we can see the different violations that can arise in the example; every node denotes a set of violations which may become true at a certain point in the execution of the system. At the beginning we have no violations (the empty set); after that we can go into a violation v_1 (when the cooler 1 is not working and the temperature is high), or to a violation v_2 when we have the same situation but for cooler 2; when both coolers are not working we get the two violations. Now, when we put the processor on stand by we have a violation v_3 which gives us the second dimension in the picture; this violation is needed in some situations to prevent the processor from burning out. If we add a formula saying that having the temperature high for at least three consecutive clock

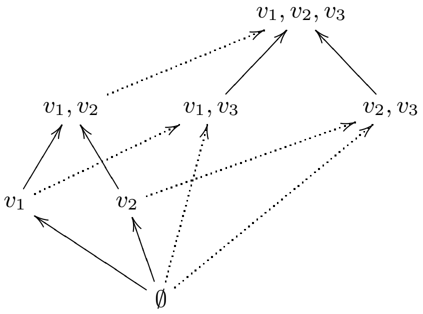


Fig. 5. Possible violations of example 2

cycles, when the processor is working, it will burn out, then we can deduce that a violation of an obligation $\mathsf{O}^2(p.sb)$ (taking the red transition three consecutive times) will yield a system crash.

These kinds of specifications seem well suited for the application of model checking to programs; for example, programs described with the nC+ ([18]) language can be checked against these specifications (to determine if the program satisfies the specification axioms), and then we can know if some deontic constraints are violated (if some red transition is taken) by the program, and what is the severity of these violations. For instance, in the example it is not the same to violate obligation O^1 as to violate obligation O^2, the second one being more dangerous (it may yield a system crash).

In the appendix we shown how this logic can be used to prove properties about specifications; for example, we provide the proof of $v_3 \rightarrow \neg p.on$; that is when we are in violation 3, the processor is on stand by. Note that this property is not obvious from the specification, and we need the *GGG* property to deduce it from the specification. (Note that in the specification we only indicate when this violation arises and what is the recovery action for this violation; the fact that no other action causes this violation comes from the deontic part of the specification.)

We prove some of the properties of this specification in the appendix (where we also describe an axiomatic system presented in earlier papers).

5 Conclusions

We have shown, using some examples, how deontic action logics can be used to express some properties of fault-tolerant systems. Though the examples are simple, they illustrate non-trivial, complex scenarios of failure and recovery, demonstrating that these ideas themselves are non trivial. For this purpose we have developed our own version of deontic logic, which has some useful metalogical properties (like compactness and decidability).

As we demonstrate in the examples, it is possible to formalize the notion of violation and normal state, which give us the possibility of analyzing how the system is degraded and repaired through time, and therefore some interesting and useful properties can be proved. The utilization of deontic operators allows us to differentiate between model description and prescription in a natural way. We have presented another (more complex) example in [1], and we proved several properties about it; from these examples, it seems possible to conclude that we can apply the underlying logic calculus in practice; in the appendix of this paper we show a simple proof to give a taste of the logical machinery in practice.

However, we need to do research about practical decision methods for the proposed logic. Our final goal is to provide automatic tools which allow designers to analyze models (and programs) in a practical way. Towards this goal, it is also interesting to research how we can modularize the deontic specifications, in such a way that different components have different deontic contracts (obligations and permissions) and then the system specification could be derived from the individual ones.

As shown in section 4, another interesting branch of investigation seems to be *contrary to duty reasoning*, in particular how this kind of reasoning is applied in fault-tolerance. Contrary to duty structures are sets of sentences, where there is a primary obligation and a secondary obligation, which arises from the violation of the primary one. These kinds of formulae are hard to reason about, as is shown everywhere in the deontic literature; indeed, several paradoxes are contrary to duty structures. In this paper we have presented an example which shows how contrary-to-duty structures can arise in fault-tolerant systems; as shown in this example, these kinds of formulae add extra structure to the violation lattice. For dealing with such complexities, we have introduced a modification of the logic, in such a way that we can introduce different permissions (and therefore obligations); intuitively this means that we will have more than two colors in our models (we can think in a broader set of reds). This adds more expressivity in the logic, but it also complicates the semantic structures.

References

1. Castro, P., Maibaum, T.: An ought-to-do deontic logic for reasoning about fault-tolerance: The diarrheic philosophers. In: 5th IEEE International Conference on Software Engineering and Formal Method. IEEE, Los Alamitos (2007)
2. Castro, P., Maibaum, T.: A complete and compact deontic action logic. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) ICTAC 2007. LNCS, vol. 4711, pp. 109–123. Springer, Heidelberg (2007)
3. Maibaum, T.: Temporal reasoning over deontic specifications. In: Sons, J.W. (ed.) Deontic Logic in Computer Science (1993)
4. Magee, J., Maibaum, T.: Towards specification, modelling and analysis of fault tolerance in self managed systems. In: Proceeding of the 2006 international workshop on self-adaptation and self-managing systems (2006)
5. Kent, S., Maibaum, T., Quirk, W.: Formally specifying temporal constraints and error recovery. In: Proceedings of IEEE International Symposium on Requirements Engineering, pp. 208–215 (1993)
6. Carmo, J., Jones, A.J.I.: Deontic database contraints, violation and recovery. Studia Logica 57(1), 139–165 (1996)

7. Chellas, B.F.: Modal Logic: An Introduction. Cambridge University Press, Cambridge (1999)
8. Meyer, J.: A different approach to deontic logic: Deontic logic viewed as variant of dynamic logic. Notre Dame Journal of Formal Logic 29 (1988)
9. Broersen, J.: Modal Action Logics for Reasoning about Reactive Systems. PhD thesis, Vrije University (2003)
10. Khosla, S., Maibaum, T.: The prescription and description of state-based systems. In: Banieqnal, B., Pnueli, H.A. (eds.) Temporal Logic in Computation. Springer, Heidelberg (1985)
11. Dignum, F., Kuiper, R.: Combining dynamic deontic logic and temporal logic for the specification of deadlines. In: Proceedings of the thirtieth HICSS (1997)
12. Meyer, J., Wieringa, R., Dignum, F.: The paradoxes of deontic logic revisited: A computer science perspective. Technical Report UU-CS-1994-38, Utrecht University (1994)
13. Sergot, M.J., Prakken, H.: Contrary-to-duty obligations. In: DEON 1994. Proc. Second International Workshop on Deontic Logic in Computer Science (1994)
14. Meyer, J., Wieringa, R.: Deontic logic: A concise overview. In: First International Workshop on Deontic Logic (DEON 1991) (1991)
15. Belzer, M.: Legal reasoning in 3-d. In: ICAIL, pp. 155–163 (1987)
16. Emerson, E., Halpern, J.: Decision procedures and expressiveness in the temporal logic of branching time. In: 14th Annual Symposiun on Theory of Computing (STOC) (1982)
17. Fitting, M.: First-Order Logic and Automated Theorem Proving. Springer, Heidelberg (1990)
18. Sergot, M.J., Craven, R.: The deontic component of action language $n\mathcal{C}+$. In: Goble, L., Meyer, J.-J.C. (eds.) DEON 2006. LNCS, vol. 4048, pp. 222–237. Springer, Heidelberg (2006)

Appendix

In this section we show how we can prove properties using the logical machinery presented above. First, the temporal part of the logic is, basically, a Computation Tree Logic, and therefore we have the following valid properties:

Temp1. $\mathsf{AG}\varphi \leftrightarrow \varphi \wedge \mathsf{ANAG}\varphi$
Temp2. $\mathsf{EN}\varphi \leftrightarrow \neg\mathsf{AN}\neg\varphi$
Temp3. $\mathsf{E}(\varphi\,\mathcal{U}\,\psi) \leftrightarrow \psi \vee (\varphi \wedge \mathsf{ENE}(\varphi\,\mathcal{U}\,\psi))$
Temp4. $\mathsf{A}(\varphi\,\mathcal{U}\,\psi) \leftrightarrow \psi \vee (\varphi \wedge \mathsf{ANA}(\varphi\,\mathcal{U}\,\psi))$
Temp5. $[\alpha]\mathsf{Done}(\alpha)$
Temp6. $[\overline{\alpha}]\neg\mathsf{Done}(\alpha)$
Temp7. $\neg\mathsf{Done}(\emptyset)$
Temp8. $\neg\mathsf{Done}(\mathbf{U}) \rightarrow \neg\mathsf{Done}(\alpha)$

Properties **Temp1-Temp4** are classic valid formulae of computational tree logics. **Temp5** to **Temp8** define the basic properties of the done predicate. **Temp5** says that after doing α, $\mathsf{Done}(\alpha)$ is true. **Temp6** says that after doing something different from α, the predicate $\mathsf{Done}(\alpha)$ is false. Finally, **Temp7** expresses that we cannot do an impossible action, and **Temp8** says that at the beginning we have not performed any action.

We have also a number of deduction rules, the most important being the following:

TempRule1. $\{\neg\mathsf{Done}(\mathbf{U}) \to \varphi, \varphi \to \mathsf{AN}\varphi\} \vdash \varphi$

This rule allows us to use a kind of inductive reasoning: if we prove that a property holds at the beginning, and if we suppose that the property is true, and we prove that it is true at the next instant, we can conclude that this property is true always.

Recall the properties $\mathsf{P1} - \mathsf{P8}$ introduced in section 2, there are two further important properties which relate strong permission and weak permission:

$\mathsf{P9}$. $(\bigwedge_{[\alpha]_{BA} \wedge \alpha \sqsubseteq \alpha'} (P_w(\alpha) \vee (\alpha =_{act} \emptyset))) \to \mathsf{P}(\alpha')$
$\mathsf{P10}$. $\mathsf{P}(\alpha) \wedge \alpha \neq_{act} \emptyset \to \mathsf{P_w}(\alpha)$

$\mathsf{P9}$ says that strong permission implies weak permission for "*non-impossible*" actions. $\mathsf{P10}$ says that *if every way of executing an action α is weakly permitted, then α is strongly permitted.*

Let us prove the property $v_3 \to \neg p.on$ using these properties.

1a. $\neg\mathsf{Done}(\mathbf{U}) \to \neg v_3$	Ax1 & PL
2a. $\neg\mathsf{Done}(\mathbf{U}) \to (v_3 \to \neg p.on)$	PL, 1a

This proves the first case of the induction, the other case is as follows:

1b. $\neg v_3 \wedge \neg s.high \to \mathsf{P}^i(\overline{p.sb})$	PL, Ax3
2b. $\neg v_3 \wedge \mathsf{P}^i(\overline{p.sb}) \to [\overline{p.sb}]\neg v_3$	PL, **C1'**
3b. $\neg v_3 \wedge \neg s.high \to [\overline{p.sb}]\neg v_3$	PL, 1b, 2b
4b. $\neg v_3 \wedge s.high \to \mathsf{O}^1(c_1.on \sqcap c_2.on)$	PL, Ax4
5b. $\mathsf{O}^1(c_1.on \sqcap c_2.on) \to \mathsf{F}^1(\overline{c_1.on}) \wedge \mathsf{F}^1(\overline{c_2.on}) \wedge \mathsf{P}^1(c_1.on \sqcap c_2.on)$	PL, Def.O
6b. $\neg v_3 \wedge s.high \to [\overline{c_1.on} \sqcup \overline{c_2.on}]\neg v_3$	PL, $\mathsf{P1}$, 4b, 5b, Ax6, Ax7
7b. $\neg v_3 \wedge s.high \to [c_1.on \sqcap c_2.on]\neg v_3$	PL, 4b, 5b, **C1'**
8b. $\neg v_3 \wedge s.high \to [\mathbf{U}]\neg v_3$	PL, $\mathsf{P1}$, BA, 6b, 7b
9b. $[p.sb]\neg p.on$	Ax15
10b. $\neg v_3 \to [p.sb](v_3 \to \neg p.on)$	ML, 9b
11b. $\neg v_3 \to [\overline{p.sb}](v_3 \to \neg p.on)$	PL, $\mathsf{P1}$, BA, 8b, 3b
12b. $\neg v_3 \to [\mathbf{U}](v_3 \to \neg p.on)$	PL, $\mathsf{P1}$, 10b, 11b
13b. $\neg p.on \to [\overline{p.up}]\neg p.on$	Ax15
14b. $v_3 \to [p.up]\neg v_3$	Ax11
15b. $v_3 \wedge (v_3 \to \neg p.on) \to [U](v_3 \to \neg p.on)$	PL, BA, $\mathsf{P1}$, 13b, 14b
16b. $(v_3 \to \neg p.on) \to [\mathbf{U}](v_3 \to \neg p.on)$	PL, 12b, 15b

The acronyms "BA", "ML" and "PL" refer to that a property of boolean algebras, modal logic or propositional logic, respectively, are used at that point of the proof. It is important to remark that we use the condition **C1'** during the proof (line $\mathsf{2b}$); this condition seems to be useful for proving properties of deontic specifications, since we can deduce preservation properties from it (i.e., when an action preserves the absence of errors).

Temporal Verification of Fault-Tolerant Protocols

Michael Fisher, Boris Konev, and Alexei Lisitsa

Department of Computer Science, University of Liverpool, Liverpool, United Kingdom
{MFisher,Konev,A.Lisitsa}@liverpool.ac.uk

1 Introduction

The automated verification of concurrent and distributed systems is a vibrant and successful area within Computer Science. Over the last 30 years, *temporal logic* [10,20] has been shown to provide a clear, concise and intuitive description of many such systems, and automata-theoretic techniques such as *model checking* [7,14] have been shown to be very useful in practical verification. Recently, the verification of *infinite-state* systems, particularly parameterised systems comprising *arbitrary* numbers of identical processes, has become increasingly important [5]. Practical problems of an open, distributed nature often fit into this model, for example robot swarms of arbitrary sizes.

However, once we move beyond finite-state systems, which we do when we consider systems with arbitrary numbers of components, problems can occur. Although temporal logic still retains its ability to express such complex systems, verification techniques such as model checking must be modified. *Abstraction* techniques are typically used to reduce an infinite-state problem down to a finite-state variant suitable for application of standard model checking techniques. However, it is clear that such abstraction techniques are not always easy to apply and that more sophisticated verification approaches must be developed.

In assessing the reliability of such infinite-state systems, formal verification is clearly desirable and, consequently, several new approaches have been developed:

1. *model checking for parameterised and infinite state-systems* [1,2];
2. *constraint based verification using counting abstractions* [9,11];
3. *verification based on interactive theorem proving* [21,22], *including that for temporal logic* [4,23];
   and
4. *deductive verification in first-order* decidable *temporal logics* [12,8].

The last of these approaches is particularly appealing, often being both complete (unlike (1)) and decidable (unlike (2)), able to verify both safety *and* liveness properties, and adaptable to more sophisticated systems involving asynchronous processes or communication delays. It is also (unlike (3)) fully mechanisable and does not require human interaction during the proof.

Now we come to the problem of verifying fault tolerance in protocols involving an arbitrary number of processes. What if some of the processes develop faults? Will the

M. Butler et al. (Eds.): Fault Tolerance, LNCS 5454, pp. 44–56, 2009.
© Springer-Verlag Berlin Heidelberg 2009

protocol still work? And how many processes must fail before the protocol fails? Rather than specifying *exactly* how many processes will fail, which reduces the problem to a simpler version, we wish to say that there is *some* number of faulty processes, and that failure can occur at any time. Again we can capture this using temporal logics. If we allow there to be an infinite number of failures, then the specification and verification problem again becomes easier; however, such scenarios appear unrealistic. In many cases, correctness of the protocols depends heavily on the assumption of a known number of failures.

So, we are left with the core problem: *can we develop deductive temporal techniques for the verification of parameterised systems where a* ***finite****, but unknown, number of failures can occur?* This question is exactly what we address here.

We proceed as follows. Section 2 gives a brief review of *first-order temporal logic* (FOTL) and its properties. In Section 3, we propose two mechanisms for adapting deductive techniques for FOTL to the problem of finite numbers of failures in infinite-state systems, and in Section 4 we outline a case study. Finally, in Section 5, we provide concluding remarks.

2 Monodic First-Order Temporal Logics

First-order (linear time) temporal logic (FOTL) is a very powerful and expressive formalism in which the specification of many algorithms, protocols and computational systems can be given at a natural level of abstraction [20]. Unfortunately, this power also means that, over many natural time flows, this logic is highly undecidable (not even recursively enumerable). Even with incomplete proof systems, or with proof systems complete only for restricted fragments, FOTL is interesting for the case of parameterised verification: one proof may certify correctness of an algorithm for infinitely many possible inputs, or correctness of a system with infinitely many states.

FOTL is an extension of classical first-order logic by temporal operators for a discrete linear model of time (isomorphic to $\mathbb{N}$, being the most commonly used model of time). Formulae of this logic are interpreted over structures that associate with each element n of $\mathbb{N}$, representing a moment in time, a first-order structure $\mathfrak{M}_n = (D, I_n)$ with the same non-empty domain D.

The *truth* relation $\mathfrak{M}_n \models^{\mathfrak{a}} \phi$ in the structure $\mathfrak{M}$ and a variable assignment $\mathfrak{a}$ is defined inductively in the usual way under for the following (sample) temporal operators:

$\mathfrak{M}_n \models^{\mathfrak{a}} \bigcirc\phi$ iff $\mathfrak{M}_{n+1} \models^{\mathfrak{a}} \phi$;

$\mathfrak{M}_n \models^{\mathfrak{a}} \Diamond\phi$ iff there exists $m \geq n$ such that $\mathfrak{M}_m \models^{\mathfrak{a}} \phi$;

$\mathfrak{M}_n \models^{\mathfrak{a}} \Box\phi$ iff for all $m \geq n$, $\mathfrak{M}_m \models^{\mathfrak{a}} \phi$;

$\mathfrak{M}_n \models^{\mathfrak{a}} \phi\,\mathsf{U}\,\psi$ iff there exists $m \geq n$ such that $\mathfrak{M}_m \models^{\mathfrak{a}} \psi$ and for all $n \leq i < m$ $\mathfrak{M}_i \models \phi$;

$\mathfrak{M}_n \models^{\mathfrak{a}} \CIRCLE\phi$ iff $n > 0$ and $\mathfrak{M}_{n-1} \models^{\mathfrak{a}} \phi$;

$\mathfrak{M}_n \models^{\mathfrak{a}} \blacksquare\phi$ iff for all $m \leq n$,$\mathfrak{M}_m \models^{\mathfrak{a}} \phi$;

$\mathfrak{M}_n \models^{\mathfrak{a}} \blacklozenge\phi$ iff there exists $0 \leq m < n$ such that $\mathfrak{M}_m \models^{\mathfrak{a}} \phi$;

$\mathfrak{M}_n \models^{\mathfrak{a}} \phi\,\mathcal{S}\,\psi$ iff there exists $m \leq n$ such that $\mathfrak{M}_m \models^{\mathfrak{a}} \psi$ and for all $m < i \leq n$ $\mathfrak{M}_i \models \phi$.

The non-temporal aspects have semantics as follows:

$$\begin{array}{lll}
\mathfrak{M}_n \models^{\mathfrak{a}} \top & & \\
\mathfrak{M}_n \models^{\mathfrak{a}} P(t_1,\ldots,t_n) & \text{iff } (I_n(\mathfrak{a}(t_1)),\ldots,I_n(\mathfrak{a}(t_n))) \in I_n(P) \\
\mathfrak{M}_n \models^{\mathfrak{a}} \neg\varphi & \text{iff not } \mathfrak{M}_n \models^{\mathfrak{a}} \varphi \\
\mathfrak{M}_n \models^{\mathfrak{a}} \varphi \vee \psi & \text{iff } \mathfrak{M}_n \models^{\mathfrak{a}} \varphi \text{ or } \mathfrak{M}_n \models^{\mathfrak{a}} \psi \\
\mathfrak{M}_n \models^{\mathfrak{a}} \exists x\varphi & \text{iff } \mathfrak{M}_n \models^{\mathfrak{b}} \varphi \text{ for some assignment } \mathfrak{b} \text{ that may differ from} \\
 & \quad \mathfrak{a} \text{ only in } x \text{ and such that } \mathfrak{b}(x) \in D \\
\mathfrak{M}_n \models^{\mathfrak{a}} \forall x\varphi & \text{iff } \mathfrak{M}_n \models^{\mathfrak{b}} \varphi \text{ for every assignment } \mathfrak{b} \text{ that may differ from} \\
 & \quad \mathfrak{a} \text{ only in } x \text{ and such that } \mathfrak{b}(x) \in D
\end{array}$$

$\mathfrak{M}$ is a *model* for a formula ϕ (or ϕ is *true* in $\mathfrak{M}$) if there exists an assignment $\mathfrak{a}$ such that $\mathfrak{M}_0 \models^{\mathfrak{a}} \phi$. A formula is *satisfiable* if it has a model. A formula is *valid* if it is satisfiable in any temporal structure under any assignment. The set of valid formulae of this logic is not recursively enumerable. Thus, there was a need for an approach that could tackle the temporal verification of parameterised systems in a *complete* and *decidable* way. This was achieved for a wide class of parameterised systems using *monodic temporal logic* [15].

Definition 1. *A* FOTL *formula is said to be* **monodic** *if, and only if, any subformula with its main connective being a temporal operator has at most one free variable.*

Thus, ϕ is called *monodic* if any subformula of ϕ of the form $\bigcirc\psi$, $\square\psi$, $\Diamond\psi$, $\blacklozenge\psi$, etc., contains at most one free variable. For example, the formulae $\forall x.\ \square\exists y.\ P(x,y)$ and $\forall x.\ \square P(x,c)$ are monodic, while $\forall x,y.\ (P(x,y) \Rightarrow \square P(x,y))$ is *not* monodic.

The monodic fragment of FOTL has appealing properties: it is axiomatisable [24] and many of its sub-fragments, such as the two-variable or monadic cases, are decidable. This fragment has a wide range of applications, for example in spatio-temporal logics [13] and temporal description logics [3]. A practical approach to proving monodic temporal formulae is to use *fine-grained temporal resolution* [17], which has been implemented in the theorem prover TeMP [16]. It was also used for deductive verification of parameterised systems [12]. One can see that in many cases temporal specifications fit into the even narrower, and decidable, monodic *monadic* fragment. (A formula is monadic if all its predicates are *unary*.)

3 Incorporating Finiteness

When modelling parameterised systems in temporal logic, informally, elements of the domain correspond to processes, and predicates to states of such processes [12]. For example $idle(x)$ means that a process x is in the idle state, $\Diamond\forall y.\ agreement(y)$ means that, eventually, all processes will be in agreement, while $\exists z.\ \square inactive(z)$ means that there is at least one process that is always inactive. (See [12] for further details.)

For many protocols, especially when fault tolerance is concerned, it is essential that the number of processes is finite. The straightforward generalisation to infinite numbers of processes makes many protocols incorrect. Although decidability of monodic fragments holds also for the case of semantics where only temporal structures over *finite* *domains* are allowed [15], the proof is model-theoretic and no practical procedure is known.

We here examine two approaches that allow us to handle the problem of finiteness within temporal specification:

- first, in 3.1 we consider proof principles which can be used to establish correctness of some parameterised protocols;
- then in 3.2 we prove that, for a wide class of protocols, decision procedures that do not assume the finiteness of a domain can still be used.

3.1 Formalising Finiteness Principles

The language of FOTL is very powerful and one might ask if a form of finiteness can be defined inside the logic. We have found the following principles (which are valid over finite domains, though not in general) useful when analysing the proofs of correctness of various protocols and algorithms specified in FOTL (recall: $\blacklozenge\varphi$ means φ *was* true in the past):

Fin_1 (**deadline axiom**): $\Diamond(\forall x.\ (\Diamond P(x) \rightarrow \blacklozenge P(x)))$

Fin_2 (**finite clock axiom**): $[\forall x.\ \Box(P(x) \rightarrow \bigcirc\Box\neg P(x)] \Rightarrow [\Diamond\Box(\forall x.\ \neg P(x))]$

Fin_3 (**stabilisation axiom**):

$$[\Box(\forall x.\ (P(x) \rightarrow \bigcirc P(x))] \Rightarrow [\Diamond\Box(\forall x.\ (\bigcirc P(x) \rightarrow P(x))]$$

Actually the Fin_1 principle is a (more applicable) variant of the intuitively clearer principle $[\forall x.\ \Diamond P(x)] \Rightarrow [\Diamond\forall x.\ \blacklozenge P(x)]$ which is also valid over finite domains.

These principles have the following informal motivation. The deadline axiom principle, Fin_1, states that there is a moment after which "nothing new is possible"; that is, if, after the deadline, $P(x)$ becomes true for some domain element $a \in D$, there already was a moment in the past such that $P(x)$ was true on a at that moment. The final clock axiom, Fin_2, states that if the moments when the predicate $P(x)$ becomes true on some domain element are interpreted as clock ticks, the clock will eventually stop ticking. Finally, the stabilisation principle, Fin_3, states that if some domain area, where $P(x)$ is true, is growing then it will stop growing at some point. It can be easily seen that all these principles hold true in arbitrary finite domain structures.

Now, consider Fin_i for $i = 1, 2, 3$ as axiom schemes which can be added to a *reasonable* axiomatisation of FOTL (call this, Ax_{FOTL}) in order to capture, at least partially, "finite reasoning". By a 'reasonable' Ax_{FOTL}, we mean that we assume some Hilbert-style finitary axiomatic system for FOTL extending a standard, non-temporal, predicate logic axiomatisation by, at least, the axiom schemata presented in Fig. 1.

We show that all these three principles are actually equivalent modulo any reasonable Ax_{FOTL} (i.e. they can be mutually derived). The principle (an axiom scheme) F_1 is said to be derivable from F_2 if, for every instance α of F_1, we have $Ax_{\mathsf{FOTL}} + F_2 \vdash \alpha$. We will denote it simply $Ax_{\mathsf{FOTL}} + F_2 \vdash F_1$.

Theorem 1. *The principles Fin_1, Fin_2 and Fin_3 are mutually derivable.*

Future time axioms:

F0. $\vdash \Box\varphi \rightarrow \varphi$
F1. $\vdash \bigcirc\neg\varphi \leftrightarrow \neg\bigcirc\varphi$
F2. $\vdash \bigcirc(\varphi \rightarrow \psi) \rightarrow (\bigcirc\varphi \rightarrow \bigcirc\psi)$
F3. $\vdash \Box(\varphi \rightarrow \psi) \rightarrow (\Box\varphi \rightarrow \Box\psi)$
F4. $\vdash \Box\varphi \rightarrow \Box\bigcirc\varphi$
F5. $\vdash (\varphi \rightarrow \bigcirc\varphi) \rightarrow (\varphi \rightarrow \Box\varphi)$
F6. $\vdash (\varphi \,\mathsf{U}\, \psi) \leftrightarrow \psi \vee (\varphi \wedge \bigcirc(\varphi \,\mathsf{U}\, \psi))$
F7. $\vdash (\varphi \,\mathsf{U}\, \psi) \rightarrow \Diamond\psi$

Past time axioms:

P1. $\vdash \neg\bullet\neg\varphi \rightarrow \bullet\varphi$
P2. $\vdash \bullet(\varphi \rightarrow \psi) \rightarrow (\bullet\varphi \rightarrow \bullet\psi)$
P3. $\vdash \varphi\,\mathcal{S}\,\psi \leftrightarrow \psi \vee (\varphi \wedge \neg\bullet\neg(\varphi\,\mathcal{S}\,\psi))$
P4. $\vdash \bullet$**false**

Mixed axiom:

M8. $\vdash \varphi \rightarrow \bigcirc\bullet\varphi$

Interaction axioms:

I1. $\vdash \forall x.\,(\bigcirc\varphi(x)) \rightarrow \bigcirc(\forall x.\,\phi(x))$
I2. $\vdash \forall x.\,(\bullet\varphi(x)) \rightarrow \bullet(\forall x.\,\phi(x))$

Fig. 1. Ax_{FOTL}: Axioms of FOTL (all except the Interaction Axioms are taken from [18])

Proof

1. $Ax_{\mathsf{FOTL}} + Fin_1 \vdash Fin_2$.

   Assume $\forall x.\,\Box(P(x) \rightarrow \bigcirc\,\Box\neg P(x))$ $(*)$, which is the assumption of Fin_2.

   Consider then Fin_1 which is $\Diamond(\forall x.(P(x) \vee \Diamond P(x) \rightarrow \blacklozenge P(x)))$.

   In Fin_1 assume $\Diamond P(c)$ for an arbitrary c inside of $\Diamond(\ldots)$, then we have $\blacklozenge P(c)$ which together with $(*)$ gives $\neg P(c)$ and contradiction.

   That means, we have $\Diamond(\forall x.\neg(P(x) \vee \Diamond P(x)))$ which implies $\Diamond\Box(\forall x.\neg P(x))$.

2. $Ax_{\mathsf{FOTL}} + Fin_2 \vdash Fin_3$.

   Define $Q(x)$ to be $\neg P(x) \wedge \bigcirc P(x)$.

   Assume $\Box(\forall x.(P(x) \rightarrow \bigcirc P(x)))$ $(**)$.

   Then we have $\forall x.\ \Box(Q(x) \rightarrow \bigcirc\Box\neg Q(x))$ from the definition of $Q(x)$ and $(**)$.

   Applying Fin_2 we get $\Diamond\Box(\forall x.\ \neg Q(x))$ which is equivalent to $\Diamond\Box(\forall x.\neg\bigcirc P(x) \vee P(x))$ and to $\Diamond\Box(\forall x.\,(\bigcirc P(x) \rightarrow P(x)))$.

3. $Ax_{\mathsf{FOTL}} + Fin_3 \vdash Fin_1$.

   Applying to a valid formula, provable in Ax_{FOTL}, $\forall x(\blacklozenge P(x) \rightarrow \bigcirc\blacklozenge P(x))$ the principle Fin_3 we get

   $$\Diamond\Box(\forall x.(\bigcirc\blacklozenge P(x) \rightarrow \blacklozenge P(x)))\,.$$

   This implies [18] $\Diamond\Box(\forall x.\ \neg(\Diamond P(x)) \wedge \neg\blacklozenge P(x)))$.

   After propositionally equivalent transformations we get $\Diamond\Box(\forall x.\ (\Diamond P(x)) \rightarrow \blacklozenge P(x))$, which is Fin_1.

This theorem shows that all three principles are equivalent and so can be used interchangeably in the proofs. However, the differing syntactical forms may make some principles more suitable for *natural* proofs, yet may affect the efficiency of the automated proof search using these principles.

3.2 Eventually Stable Protocols

In Section 3.1 we highlighted some deduction principles capturing the finiteness of the domain. Alternatively, we can consider a family of protocols which terminate after a certain (but unknown) number of steps. For example, if every process sends only a finite number of messages, such protocol will eventually terminate. Consensus protocols [19], distributed commit protocols [6], and some other protocols fit into this class. Temporal models of specifications of such terminating protocols will eventually stabilise, that is, the interpretations I_n will be the same for sufficiently large n. We show that for these *eventually stable* specifications satisfiability over finite domains coincides with satisfiability over arbitrary domains.

Let $\mathcal{P}$ be a set of unary predicates. The *stabilisation principle w.r.t.* $\mathcal{P}$ is the formula:

$$\mathsf{Stab}_{\mathcal{P}} = \Box(\forall x \bigwedge_{P\in\mathcal{P}} [P(x) \equiv \bigcirc P(x)]).$$

Informally, if $\mathsf{Stab}_{\mathcal{P}}$ is true at some moment of time, from this moment the interpretation of predicates in $\mathcal{P}$ does not change. Let ϕ be a monodic temporal formula. Let $\mathcal{P}$ be the set of unary predicates occurring in ϕ. Then the formula

$$\phi_{\mathsf{Stab}} = \phi \wedge \Diamond\mathsf{Stab}$$

is called an *eventually stable formula*. We formulate the following proposition for monodic monadic formulae; it can be extended to other monodic classes obtained by *temporalisation by renaming* [8] of first-order classes with the finite model property.

Proposition 1. *Let ϕ be a monodic monadic formula. The eventually stable formula ϕ_{Stab} is satisfiable in a model with a finite domain if, and only if, ϕ_{Stab} is satisfiable in a model with an arbitrary domain.*

This proposition implies that if a protocol is such that it can be faithfully represented by an eventually stable formula, correctness of such protocol can be established by a procedure that does *not* assume the finiteness of the domain.

Proof . For simplicity, we prove the proposition for formulae in Divided Separated Normal Form (DSNF) [8] only. The proof can be extended to the general case by the consideration of sub-formulae of ϕ.

A *monodic temporal problem* P *in divided separated normal form (DSNF)* is a quadruple $\langle \mathcal{U}, \mathcal{I}, \mathcal{S}, \mathcal{E} \rangle$, where:

1. the universal part $\mathcal{U}$ and the initial part $\mathcal{I}$ are finite sets of first-order formulae;
2. the step part $\mathcal{S}$ is a finite set of clauses of the form $p \Rightarrow \bigcirc q$, where p and q are propositions, and $P(x) \Rightarrow \bigcirc Q(x)$, where P and Q are unary predicate symbols and x is a variable; and
3. the eventuality part $\mathcal{E}$ is a finite set of formulae of the form $\Diamond L(x)$ (a *non-ground* eventuality clause) and $\Diamond l$ (a *ground eventuality* clause), where l is a propositional literal and $L(x)$ is a unary non-ground literal with variable x as its only argument.

With each monodic temporal problem $\langle \mathcal{U}, \mathcal{I}, \mathcal{S}, \mathcal{E} \rangle$ we associate the FOTL formula $\mathcal{I} \wedge \Box\mathcal{U} \wedge \Box\forall x\mathcal{S} \wedge \Box\forall x\mathcal{E}$. When we talk about particular properties of a temporal problem (e.g., satisfiability, validity, logical consequences, etc) we refer to properties of this associated formula. Every monodic temporal formula can be transformed into divided separated normal form (DSNF) in a satisfiability equivalence preserving way with only linear growth in size [17].

Let $\mathsf{P} = \langle \mathcal{U}, \mathcal{I}, \mathcal{S}, \mathcal{E} \rangle$ be a monodic temporal problem in DSNF. We only have to show that if $\mathsf{P}_{\mathsf{Stab}}$ has a model, $\mathfrak{M} = \mathfrak{M}_0, \mathfrak{M}_1, \ldots$, with an infinite domain, it also has a model with a finite one. Let N be such that $\mathfrak{M}_N \models \mathsf{Stab}$. Consider now the temporal structure $\mathfrak{M}' = \mathfrak{M}_0, \mathfrak{M}_1, \ldots \mathfrak{M}_{N-1}, \mathfrak{M}_N, \mathfrak{M}_N, \mathfrak{M}_N, \ldots$ (i.e. from moment N the structure does not change). It can be seen that $\mathfrak{M}'$ is a model for P.

For every predicate, P, occurring in P, we introduce $N+1$ new predicates $P^0, P^1, \ldots, P^N$ of the same arity. Let ϕ be a first-order formula in the language of P. We denote by $[\phi]^i$, $0 \leq i \leq N$, the result of substitution of all occurrences of predicates in ϕ with their i-th counterparts; (e.g., $P(x_1, x_2)$ is replaced with $P^i(x_1, x_2)$).

- Let $\phi_{\mathcal{I}} = \bigwedge\{[\phi]^0 \mid \phi \text{ is in } \mathcal{I}\}$
- Let $\phi_{\mathcal{U}} = \bigwedge\{\bigwedge\limits_{i=0}^{N} [\phi]^i \mid \phi \text{ is in } \mathcal{U}\}$.
- Let $\phi_{\mathcal{S}} = \bigwedge\{\bigwedge\limits_{i=0}^{N-1} \big(\forall x(P^i(x) \Rightarrow Q^{i+1}(x)\big) \mid P(x) \Rightarrow \bigcirc Q(x) \text{ is in } \mathcal{S}\} \wedge$ $\bigwedge\{\forall x(P^N(x) \Rightarrow Q^N(x) \mid P(x) \Rightarrow \bigcirc Q(x) \text{ is in } \mathcal{S}\}$
- Let $\phi_{\mathcal{E}} = \bigwedge\{[\forall x L(x)]^N \mid L(x) \text{ is in } \mathcal{E}\}$

Let $\phi_{FO} = \phi_{\mathcal{I}} \wedge \phi_{\mathcal{U}} \wedge \phi_{\mathcal{S}} \wedge \phi_{\mathcal{E}}$. Note that ϕ_{FO} does not contain any temporal operators. Consider now a first-order structure $\mathfrak{N}$ with the same domain D as $\mathfrak{M}$, interpreting constants in the same way, and such that $\mathfrak{N} \models P^i(a_1, \ldots, a_n)$, for some $a_1, \ldots, a_n \in D$, if, and only if, $\mathfrak{M}_i \models P(a_1, \ldots, a_n)$. It can be seen that that $\mathfrak{N} \models \phi_{FO}$. Since P is a monodic monadic problem, ϕ_{FO} is a monadic first-order formula, which has a model with a finite domain. Reversing the process, one can construct a model for P with a finite domain. ❑

4 Case Study: FloodSet Protocol

Next, we provide an example of how both methods described in Section 3 (explicit finiteness principles, and stabilisation principle for protocols with finite change) can be used for the proof of correctness of a protocol specified in monodic FOTL.

The setting is as follows. There are n processes, each having an *input bit* and an *output bit*. The processes work synchronously, run the same algorithm and use *broadcast* for communication. Any message sent by a non-faulty process is instantaneously delivered to all other processes. Some processes may fail and, from that point onward, such processes do not send any further messages. Note, however, that the messages sent by a process *in the moment of failure* may be delivered to *an arbitrary subset* of the processes. Crucially, there is a *finite* bound, f, on the number of processes that may fail.

The goal of the algorithm is to eventually reach an agreement, i.e. to produce an output bit, which would be the same for all non-faulty processes. It is required also that if all processes have the same input bit, that bit should be produced as an output bit.

This is a variant of *FloodSet algorithm with alternative decision rule* (in terms of [19], p.105) designed for solution of the Consensus problem in the presence of crash (or fail-stop) failures, and the basic elements of the protocol (adapted from [19][1]) are as follows.

- In the first round of computations, every process broadcasts its input bit.
- In every later round, a process broadcasts any value *the first time it sees it.*
- In every round the (tentative) output bit is set to the minimum value seen so far.

The correctness criterion for this protocol is that, eventually (actually, no later than in $f + 2$ rounds) the output bits of all non-faulty processes will be the same.

Claim. The above FloodSet algorithm and its correctness conditions can be specified (naturally) within monodic monadic temporal logic without equality, and its correctness can be proved in monodic monadic temporal logic, using the above **finite clock axiom**.

We give a larger specification below, but first note the keys points concerning this:

1. Each process (s) must be categorised as one of the above types:
   $\square(\forall x(Normal(x) \mid Failure(x) \mid Faulty(x)))$
   here the symbol $\mid$ means that exactly one of the predicates $Normal(x)$, $Failure(x)$, and $Faulty(x)$ is true.
2. If we see a '0' (the process has this already, or receives a message with this value) then we output '0':
   $\square(\forall x(\neg Faulty(x) \wedge Seen(x,0) \rightarrow \bigcirc Output(x) = 0))$
3. If we have not seen a '0' but *have* seen a '1', then we output '1':
   $\square(\forall x. (\neg Faulty(x) \wedge \neg Seen(x,0) \wedge Seen(x,1) \rightarrow \bigcirc Output(x) = 1))$

[1] In [19], every process *knows* the bound f in advance and stops the execution of the protocol after $f + 2$ rounds, producing the appropriate output bit. We consider the version where the processes do not know f in advance and produce a *tentative output bit* at every round.

4. The condition to be verified, namely that eventually all (non faulty) processes agree on the bit ‘0’, or eventually all agree on the bit ‘1’:

$$\Diamond((\forall x. \neg \mathit{Faulty}(x) \Rightarrow \mathit{Output}(x) = 0) \ \lor \ (\forall x. \neg \mathit{Faulty}(x) \Rightarrow \mathit{Output}(x) = 1))$$

We do not include the whole proof here, but will reproduce sample formulae to give the reader a flavour of the specification and proof.

4.1 Specification

A FOTL specification of the above $\mathit{FloodSet}$ algorithm φ is given as a conjunction of the following formulae, divided for convenience, into four groups as follows

1. **RULES:**
   $\Box(1\mathit{stRound} \to \bigcirc \neg 1\mathit{stRound})$
   $\Box(\forall x. (\mathit{Failure}(x) \to \bigcirc \mathit{Faulty}(x)))$
   $\Box(\forall x. (1\mathit{stRound} \land \mathit{Normal}(x) \to \bigcirc(\mathit{Send}(x, \mathit{Input}(x)) \land \mathit{Seen}(x, \mathit{Input}(x)))))$
   $\Box(\forall x. (1\mathit{stRound} \land \mathit{Failure}(x) \to \bigcirc(\mathit{Send\_Failure}(x, \mathit{Input}(x)) \land \mathit{Seen}(x, \mathit{Input}(x)))))$
   $\Box(\forall x. \forall y. (\neg 1\mathit{stRound} \land \mathit{Normal}(x) \land \mathit{Received}(x, y) \land \neg \mathit{Seen}(x, y) \to \bigcirc \mathit{Seen}(x, y) \land \mathit{Send}(x, y)))$
   $\Box(\forall x. \forall y. (\neg 1\mathit{stRound} \land \mathit{Failure}(x) \land \mathit{Received}(x, y) \land \neg \mathit{Seen}(x, y) \to \bigcirc(\mathit{Seen}(x, y) \land \mathit{Send\_Failure}(x, y))))$
   $\Box(\forall x. \forall y. (\mathit{Faulty}(x) \to \bigcirc(\neg \mathit{Send}(x, y) \land \neg \mathit{Send\_Failure}(x, y))))$
   $\Box(\forall x. \forall y. (\neg 1\mathit{stRound} \land \neg \mathit{Faulty}(x) \land (\neg \mathit{Received}(x, y) \lor \mathit{Seen}(x, y)) \to \bigcirc(\neg \mathit{Send}(x, y) \land \neg \mathit{Send\_Failure}(x, y))))$
   $\Box(\forall x. (\neg \mathit{Faulty}(x) \land \mathit{Seen}(x, 0) \to \bigcirc \mathit{Output}(x) = 0))$
   $\Box(\forall x. (\neg \mathit{Faulty}(x) \land \neg \mathit{Seen}(x, 0) \land \mathit{Seen}(x, 1) \to \bigcirc \mathit{Output}(x) = 1))$

2. **FRAME CONDITIONS:**
   $\Box(\neg 1\mathit{stRound} \to \bigcirc \neg 1\mathit{stRound})$
   $\Box(\forall x. (\mathit{Faulty}(x) \to \bigcirc \mathit{Faulty}(x)))$
   $\Box(\forall x. (\neg \mathit{Faulty}(x) \land \neg \mathit{Failure}(x) \to \bigcirc \neg \mathit{Faulty}(x)))$
   $\Box(\forall x. \forall y. (\mathit{Seen}(x, y) \to \bigcirc \mathit{Seen}(x, y)))$

3. **CONSTRAINTS:**
   $\Box(\forall x. \forall m. (\mathit{Send}(x, m) \to \forall y. \mathit{Received}(y, m)))$
   $\Box(\forall x. \forall m. (\mathit{Received}(x, m) \to \exists y(\mathit{Send}(y, m) \lor \mathit{Send\_Failure}(y, m))))$
   $\Box(\forall x. \forall m. \neg(\mathit{Send}(x, m) \land \mathit{Send\_Failure}(x, m)))$
   $\Box(\forall x. (\mathit{Normal}(x) \mid \mathit{Failure}(x) \mid \mathit{Faulty}(x)))$

   $\Box(\forall x. (\mathit{Output}(x) = 0 \lor \mathit{Output}(x) = 1))$
   $\Box(\forall x. (\mathit{Input}(x) = 0 \lor \mathit{Input}(x) = 1))$
   $\Box(\forall x. \forall y. (\mathit{Send}(x, y) \lor \mathit{Received}(x, y) \lor \mathit{Seen}(x, y)) \to (y = 0 \lor y = 1))$

4. **INITIAL CONDITIONS:**
   $\Box(\textbf{start} \Rightarrow 1\mathit{stRound})$
   $\Box(\textbf{start} \Rightarrow \forall x.\ \mathit{Normal}(x))$
   $\Box(\textbf{start} \Rightarrow \forall x. \forall y.\ \neg \mathit{Seen}(x, y))$
   $\Box(\textbf{start} \Rightarrow \forall x.\ (\mathit{Input}(x) = 0 \lor \mathit{Input}(x) = 1))$

Note. One can get rid of all equalities in this example by using finiteness of the set of values, which are supposed to be second argument of $Seen(\_,\_)$, $Send(\_,\_)$ and $Send\_Failure(\_,\_)$.

Notice that the temporal specification uses, among others, the predicates $Normal(\_)$ to denote normal operating processes, $Failure(\_)$ to denote processes experiencing failure (at some point of time), $Faulty(\_)$ for the processes already failed. There are also predicates such as $Seen(\_,\_)$ specifying the effect of communications. Having these, it is straightforward to write down the temporal formulae describing the above protocol and correctness condition (i.e. (4) above). In the proof of correctness below, the **finite clock axiom** has to be instantiated to the $Failure(x)$ predicate (i.e. replace P by $Failure$ in Fin_2).

4.2 Refutation

In this section we will consider the actual proof concerning the correctness of the above specification with respect to the conditions we have presented. We will not present the full proof, but will provide an outline indicating how the major steps occur.

First of all, the clausal temporal resolution approach is a refutation procedure and so we add the negation of the required condition (i.e. $\neg\psi$) and attempt to derive a contradiction. We note that $\neg\psi$ is

$$\Box((\exists x \neg Faulty(x) \wedge Output(x) \neq 0)$$

$$\wedge$$

$$(\exists x \neg Faulty(x) \wedge Output(x) \neq 1))$$

We translate formulae such as '$Output(x) \neq 0$' to '$\neg\neg Output(x)$' since the only values allowed are '0' and '1'. Consequently the two temporal formulae derived from $\neg\psi$ are:

$C1$: $\Box(\exists x \neg Faulty(x) \wedge Output(x))$
$C2$: $\Box(\exists x \neg Faulty(x) \wedge \neg Output(x))$

Frame conditions and the finite clock axiom applied for the *Failure* predicate give

$C3$: $\Diamond\Box(\forall x. \neg Failure(x))$

From $C1$ and $C3$ we have

$C4$: $\Diamond\Box(\forall x. \neg Failure(x) \wedge \exists x(\neg Faulty(x) \wedge Output(x)))$

From $C4$ and constraints we now have

$C5$: $\Diamond\Box(\forall x. \neg Failure(x) \wedge \exists x(Normal(x) \wedge Output(x)))$

By rules concerning *Output* and $C5$ we get

$C6$: $\Diamond\Box(\exists x Normal(x) \wedge \bullet Seen(x, 0))$

Next, let us note a useful variant of the induction axiom called the *minimal element principle*:

$$\forall\bar{x}([\Diamond\varphi(\bar{x})] \rightarrow [\Diamond(\varphi(\bar{x}) \wedge \bullet\blacksquare\neg\varphi(\bar{x}))])$$

By the minimum element principle

$C7$: $\Diamond\Box(\exists x Normal(x) \wedge \blacklozenge(Seen(x,0) \wedge \bullet\blacksquare\neg Seen(x,0)))$

By rules from $C7$

$C8$: $\Diamond\Box(\exists x Normal(x) \wedge \blacklozenge(\bullet Received(x,0)))$

By rules from $C8$

$C9$: $\Diamond\Box(\exists x Normal(x) \wedge \blacklozenge(\bullet(Received(x,0) \wedge \neg Seen(x,0))))$

By rules from $C9$

$C10$: $\Diamond\Box(\exists x Normal(x) \wedge \blacklozenge(Normal(x) \wedge Received(x,0) \wedge \neg Seen(x,0))))$

By rules from $C10$

$C11$: $\Diamond\Box(\exists x Normal(x) \wedge \blacklozenge(Send(x,0)))$

By rules from $C11$

$C12$: $\Diamond\Box(\blacklozenge\forall x.\, Seen(x,0)))$

From $C12$

$C13$: $\Diamond\Box(\forall x.\, Seen(x,0))$

From $C2$ and rules

$C14$: $\Box(\exists y \neg Seen(y,0))$

Finally, from $C13$ and $C14$ we get a contradiction. □

4.3 Eventual Stabilisation of FloodSet Protocol

One may also verify the *FloodSet* protocol using the eventual stabilisation principle from Section 3.2. To establish the applicability of the principle one may use the following arguments: every process can broadcast at most twice, and taking into account finiteness of both the numbers of processes and of failures, one may conclude that eventually the protocol stabilises. Note that such an analysis only allows us to conclude that the protocol stabilises, but its properties still need to be proved. Let ϕ be a temporal specification of the protocol. Taking into account the stabilisation property, the protocol is correct iff $(\phi \wedge \neg\psi)_{\mathsf{Stab}}$ is not satisfiable over finite domains. By Proposition 1, there is no difference in satisfiability over finite and general domains for such formulae and so one may use theorem proving methods developed for monadic monodic temporal logics over general models to establish this fact. In this case, the proof follow(s) exactly the form of proof presented in the previous section, with the exception that statement

$C3 : \Diamond\Box(\forall x.\, \neg Failure(x))$ is obtained in a different way. One of the conjuncts of the stabilisation principle with respect to $\phi \wedge \neg\psi$ is

$$\Diamond\Box(\forall x Failure(x) \equiv \bigcirc Failure(x)\,.$$

Together with the rule

$$\Box(\forall x.\, (Failure(x) \rightarrow \bigcirc Faulty(x)))$$

and the constraint

$$\Box(\forall x.\, (Normal(x) \mid Failure(x) \mid Faulty(x)))$$

this implies $C3$, as required.

5 Concluding Remarks

In this paper we have introduced two approaches for handling the finiteness of the domain in temporal reasoning.

The first approach uses explicit finiteness principles as axioms (or proof rules), and has potentially wider applicability, not being restricted to protocols with the stabilisation property. On the other hand, the automation of temporal proof search with finiteness principles appears to be more difficult and it is still largely an open problem.

In the approach based on the stabilisation principle, all "finiteness reasoning" is carried out at the meta-level and essentially this is used to reduce the problem formulated for finite domains to the general (not necessarily finite) case. When applicable, this method is more straightforward for implementation and potentially more efficient. Applicability, however, is restricted to the protocols which have stabilisation property (and this property should be demonstrated in advance as a pre-condition).

Finally, we briefly mention some future work. Automated proof techniques for monadic monodic FOTL have been developed [8,17] and implemented in the TeMP system [16], yet currently proof search involving the finiteness principles requires improvement. Once this has been completed, larger case studies will be tackled. The techniques themselves would also benefit from extension involving probabilistic, real-time and equational reasoning.

References

1. Abdulla, P.A., Jonsson, B., Nilsson, M., d'Orso, J., Saksena, M.: Regular Model Checking for LTL(MSO). In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 348–360. Springer, Heidelberg (2004)
2. Abdulla, P.A., Jonsson, B., Rezine, A., Saksena, M.: Proving Liveness by Backwards Reachability. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 95–109. Springer, Heidelberg (2006)
3. Artale, A., Franconi, E., Wolter, F., Zakharyaschev, M.: A Temporal Description Logic for Reasoning over Conceptual Schemas and Queries. In: Flesca, S., Greco, S., Leone, N., Ianni, G. (eds.) JELIA 2002. LNCS, vol. 2424, pp. 98–110. Springer, Heidelberg (2002)

4. Bjorner, N., Browne, A., Chang, E., Colon, M., Kapur, A., Manna, Z., Sipma, H.B., Uribe, T.E.: STeP: Deductive-Algorithmic Verification of Reactive and Real-time Systems. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 415–418. Springer, Heidelberg (1996)
5. Calder, M., Miller, A.: An Automatic Abstraction Technique for Verifying Featured, Parameterised Systems. Theoretical Computer Science (to appear)
6. Chkliaev, D., van der Stock, P., Hooman, J.: Mechanical Verification of a Non-Blocking Atomic Commitment Protocol. In: Proc. ICDCS Workshop on Distributed System Validation and Verification, pp. 96–103. IEEE, Los Alamitos (2000)
7. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
8. Degtyarev, A., Fisher, M., Konev, B.: Monodic Temporal Resolution. ACM Transactions on Computational Logic 7(1), 108–150 (2006)
9. Delzanno, G.: Constraint-based Verification of Parametrized Cache Coherence Protocols. Formal Methods in System Design 23(3), 257–301 (2003)
10. Emerson, E.A.: Temporal and Modal Logic. In: Handbook of Theoretical Computer Science, pp. 996–1072. Elsevier, Amsterdam (1990)
11. Esparza, J., Finkel, A., Mayr, R.: On the Verification of Broadcast Protocols. In: Proc. 14th IEEE Symp. Logic in Computer Science (LICS), pp. 352–359. IEEE CS Press, Los Alamitos (1999)
12. Fisher, M., Konev, B., Lisitsa, A.: Practical Infinite-state Verification with Temporal Reasoning. In: Verification of Infinite State Systems and Security. NATO Security through Science Series: Information and Communication, vol. 1. IOS Press, Amsterdam (2006)
13. Gabelaia, D., Kontchakov, R., Kurucz, A., Wolter, F., Zakharyaschev, M.: On the Computational Complexity of Spatio-Temporal Logics. In: Proc. 16th International Florida Artificial Intelligence Research Society Conference (FLAIRS), pp. 460–464. AAAI Press, Menlo Park (2003)
14. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley, Reading (2003)
15. Hodkinson, I., Wolter, F., Zakharyaschev, M.: Decidable Fragments of First-order Temporal Logics. Annals of Pure and Applied Logic 106, 85–134 (2000)
16. Hustadt, U., Konev, B., Riazanov, A., Voronkov, A.: TeMP: A Temporal Monodic Prover. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS, vol. 3097, pp. 326–330. Springer, Heidelberg (2004)
17. Konev, B., Degtyarev, A., Dixon, C., Fisher, M., Hustadt, U.: Mechanising First-order Temporal Resolution. Information and Computation 199(1-2), 55–86 (2005)
18. Lichtenstein, O., Pnueli, A.: Propositional Temporal Logics: Decidability and Completeness. International Journal of the IGPL 8, 55–85
19. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
20. Manna, Z., Pnueli, A.: Temporal Logic of Reactive and Concurrent Systems. Springer, Heidelberg (1992)
21. Owre, S., Rushby, J., Shankar, N., von Henke, F.: Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. IEEE Transactions on Software Engineering 21, 107–122
22. Röckl, C.: Proving write invalidate cache coherence with bisimulations in Isabelle/HOL. In: In Proc. of FBT 2000, Shaker, pp. 69–78 (2000)
23. Pnueli, A., Arons, T.: TLPVS: A PVS-based LTL verification system. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 598–625. Springer, Heidelberg (2004)
24. Wolter, F., Zakharyaschev, M.: Axiomatizing the Monodic Fragment of First-order Temporal Logic. Annals of Pure and Applied Logic 118(1-2), 133–145 (2002)

Design and Verification of Fault-Tolerant Components*

Miaomiao Zhang[1], Zhiming Liu[2,**], Charles Morisset[2], and Anders P. Ravn[3]

[1] School of Software Engineering, Tongji University, China
miaomiao@mail.tongji.edu.cn

[2] International Institute of Software Technology,
United Nations University, Macau
{Z.Liu,morisset}@iist.unu.edu

[3] Department of Computer Science, Aalborg University
apr@cs.aau.dk

Abstract. We present a systematic approach to design and verification of fault-tolerant components with real-time properties as found in embedded systems. A state machine model of the correct component is augmented with internal transitions that represent hypothesized faults. Also, constraints on the occurrence or timing of faults are included in this model. This model of a faulty component is then extended with fault detection and recovery mechanisms, again in the form of state machines. Desired properties of the component are model checked for each of the successive models. The models can be made relatively detailed such that they can serve directly as blueprints for engineering, and yet be amenable to exhaustive verification. The approach is illustrated with a design of a triple modular fault-tolerant system that is a real case we received from our collaborators in the aerospace field. We use UPPAAL to model and check this design. Model checking uses concrete parameters, so we extend the result with parametric analysis using abstractions of the automata in a rigorous verification.

Keywords: Fault-tolerance, real-time embedded systems, abstraction, model checking.

1 Introduction

Fault-tolerance is often required in components for embedded real-time systems which have to be highly dependable [14], and although fault-tolerance has been studied since the beginning of digital computing and there is a well-established terminology for the area and established mechanisms, see e.g. [3] for an overview, yet it is still difficult to implement them correctly, because the algorithms are complex and often time dependent. Furthermore, assumptions about the kind and frequency of faults are not stated explicitly, and often development involves both software engineers and digital systems engineers.

* Research supported by project No. 60603037, the National Natural Science Foundation of China and the HTTS project funded by Macau Science and Technology Development Foundation.

** Correspondence requests to: Z. Liu, UNU-IIST, P.O. Box 3058, Macao SAR, China, E-mail: z.liu@iist.unu.edu

M. Butler et al. (Eds.): Fault Tolerance, LNCS 5454, pp. 57–84, 2009.

© Springer-Verlag Berlin Heidelberg 2009

The solution we present in this paper is essentially to combine the formalization of the algorithms as transition systems [15,16] with the modelling and verification power of mature model checking [4]. Convincing support for the latter idea appears also in other verifications of fault-tolerant embedded systems [18,11,5].

The overall procedure we suggest to developers of fault-tolerant components is the following:

1. Develop a model in the form of a network of state machines of the correct component and check that it has the desired properties.
2. Model relevant faults and introduce them as internal transitions to *error states* in the previous model and produce a *fault-affected model*. Check systematically that this fault-affected model fails to have the desired properties; this is a check that the fault hypotheses are correctly modelled.
3. Introduce into the model the mechanisms for *fault detection*, *error recovery and masking* and check that the desired properties are valid for this design both in the presence and absence of faults. In this step, one may have to specify constraints on timing parameters in order to make the verification come through.
4. For certain parameter values, properties are satisfied, and for other values they fail. Therefore, we are interested in deriving the precise constraints that ensure correctness. However, current model checking tools does not support parameter analysis, so we need to instantiate the parameters for some values to check whether or not the properties are fulfilled. We improve on this in two ways: (1) we find constraints on the parameters that ensure satisfaction of these properties; (2) we abstract the model using simulation relations which are manually proved to be correct, and we check the properties on the abstracted models.

The end result is a set of models with parameter constraints that may serve as input to further detailed design of the component, because state machines models are well-understood by both digital systems designers and programmers.

1.1 Example Problem

In the remainder of this paper we illustrate the approach on a design of a fault-tolerant component that appears in many systems. The fundamental component is a computation unit (CU) that ideally does not suffer from hardware faults and correctly implements a computation. However, in reality, computers or programs may fail, either because of hardware failures, or faults in the design or in the implementation of an algorithm. Obviously, a system with a single faulty CU may fail, when a fault occurs. Consequently, if no fault-tolerant actions are implemented, a fault may cause a system failure that violates the overall system requirements. It is clear that in a system for critical missions, for instance in aerospace application, using fault-tolerant components to avoid failures is important since maintenance and fail safe behaviour is almost impossible to implement once the mission is on its way.

As a solution we adopt a classical fault-tolerant mechanism that uses multiple versions of the CU, which preferably are designed and implemented independently. The necessary redundancy is thus in space, preserving timing properties of the combined system. Redundancy in the time domain, for instance using recovery block mechanisms, do not preserve timing properties as easily.

Although the principles of the mechanism are well known, it is far from easy to model the faults and the assumptions one makes for the concrete design. Abstraction techniques have to be applied so that the model of the design can be verified with the model checking tool. The faults that we are asked to consider are transient, and therefore we can design a restart (or recovery) mechanism for a CU when it fails. This involves detecting the occurrence of a fault in a CU, and when it fails, there must be a component to trigger the restart. We are told by the domain engineers that they are concerned with three kinds of faults that may occur in a CU.

The first kind of faults cause the CU to enter a deadlock state in which the CU does not take any action. For this, we equip each CU with a watchdog that is activated periodically (kicked) during normal operation. When a fault occurs, the CU deadlocks and stops kicking its watchdog. As a consequence, the watchdog timer grows until it exceeds a predefined value, (we say that the watchdog overflows), and that shall trigger a restart of the CU.

A fault of the second kind causes the CU to output incorrect data. To detect such a value fault, we introduce a component called a *voter*. With the assumption that at any time only a minority of the CUs fail (in the concrete case with three CUs at most one of them can be in an error state), the voter can detect which CU has failed and trigger a restart. Furthermore, the voter can also *mask* the incorrect output from the failed CU.

In the third case, a CU fails by entering a livelock state. In this state, the CU fails to output any result; but it keeps kicking its watchdog periodically. Consequently, the watchdog timer does not overflow the CU failure cannot be detected by the watchdog. However, the fault detection in this case can be done by the voter that reads an empty value from the failed CU. Thus we convert an omission fault into a value fault. Actuallty the same technique would apply for deadlock. However, the engineers want to distinguish the cases.

To avoid that the voter becomes a single point of failure, which would reduce the overall fault tolerance of the component, we can use more voters. We therefore need to design a component, called the *arbiter*, to detect an error in a voter and select the output from the voter that has not failed. With this design, the arbiter should also take responsibility to trigger restart of a failed CUs. To illustrate the idea of this design, we use two voters and show how the arbiter can trigger the restart of a failed CU without considering the error detection in a voter.

Admittedly, in this paper, we only consider CU faults and do not design the concrete switching mechanism for voters when one of them fails. Also, the arbiter is a single point of failure, but it is an extremely simple piece of hardware, thus we and the engineers accept this risk.

1.2 Overview

The component described informally above is in the following designed and verified using UPPAAL [4] that is available at www.uppaal.com. It is an integrated tool environment for formal specification, validation and verification of real time systems modeled as networks of timed automata [2]. The language for the new version of UPPAAL 4.0 includes a subset of the c programming language, and the syntax of timed automata for specifying timing constraints. The c code embedded in the automaton can be easily read and translated by the engineers to for instance the Verilog language for hardware implementation. Due to these extensions, UPPAAL is sufficiently expressive for the description of critical parts of system specifications.

The remaining sections are organized as follows: Section 2 describes the system with a single `CU`; it also defines the desired properties of the system. We then introduce the assumed faults into the automaton of the `CU` to get a *fault-affected* model. Section 3 presents the design of the fault-tolerant system. In Section 4, we use a network of timed automata in UPPAAL to model the fault-affected behavior of the system and then, using different instantiations of the system parameters, we verify the correctness properties. In order to demonstrate that the system tolerates the assumed faults for all instantiations of its parameters, we carry out a parametric analysis in Section 6. Finally, Section 7 concludes the paper and points out future work.

2 Modelling Faults and Fault-Tolerance

In this section, we recall the definition of the modelling language for timed automata, and use it to model the behaviors of a fault-free component. We then extend it to model faults of the system. We further show the techniques that are used to check if a fault-affected system actually fails. Hence it is ensured the fault hypotheses are correctly modelled.

2.1 Timed Automata

Timed automata were introduced in [2] as formal models for real-time systems. Here we only give a brief description of timed automata and their behaviour. Readers are referred to [2,4] for more details. We denote by $\mathbb{R}^+$ and $\mathbb{N}$ the sets of nonnegative real numbers and natural numbers, respectively.

We write $\mathcal{C} \subseteq \mathbb{R}^+$ for the finite set of clocks, denoted as x, y in the following, and Σ for the finite alphabet of actions, denoted as a, b.

A clock constraint is a conjunctive formula of atomic constraints of the form $x \sim n$ or $x - y \sim n$, where $\sim \in \{\leq, <, =, >, \geq\}$ and $n \in \mathbb{N}$. We write $\mathcal{B}(\mathcal{C})$ for the set of clock constraints, and an element g may be used as a location invariant or a transition guard in the timed automata.

Definition 1. *A timed automaton A is a tuple $\langle L, l_0, I, E\rangle$, where*

- *L is a finite set of locations,*
- *$l_0 \in L$ is the initial location,*
- *$I : L \to \mathcal{B}(\mathcal{C})$ assigns invariants to locations,*
- *$E \subseteq L \times 2^{\mathcal{B}(\mathcal{C})} \times \Sigma \times 2^{\mathcal{C}} \times L$ is a set of transitions. A discrete transition $\langle l, g, a, r, l'\rangle$ moves from location l to location l' with action a, when the clock constraint g is satisfied; r is a set of clocks to be reset, i.e., assignments of the form $x = n$, where $n \in \mathbb{N}$.*

In UPPAAL, we restrict location invariants to constraints that are downward closed, i.e. $\sim \in \{\leq, <\}$. Only upper bounds on clocks are allowed in invariants.

In the semantics, A can also take time steps, where the location is not changed, but all clocks are advanced with a $\delta \in \mathbb{R}^+$. Admissible time steps have to satisfy the location invariant.

UPPAAL allows declaration of ordinary state variables, that act like program variables, and may be used in guard expressions and are updated in actions. The notation is similar to the one used for `c` programs.

The semantics of a network of timed automata is given in [4], in which a synchronous interaction between two automata is specified by the *complementary labelled transitions* $\xrightarrow{c?}$ and $\xrightarrow{c!}$. The labels are defined in *channel* declarations and broadcast channels are allowed as well; the broadcaster with a transition labelled $\xrightarrow{c!}$ synchronizes with all enabled transitions of the form $\xrightarrow{c?}$. If none are enabled, no synchronization take place.

UPPALL builds a symbolic representation of a computational tree, where each node in the tree represents a set of states, and the edges represent possible transition steps in the network. It can therefore check properties of the network by analyzing the paths and the states in a path in this tree. A state property π is a Boolean expression over the variables and locations of the network. It is possible in UPPAAL to ask for properties of the following form:

- `A[]` π: for all paths and all states π holds invariantly.
- `A<>` π: for all paths there is a state where π holds, *i.e.*, π is reachable.
- `E[]` π: for some path and all states of that path π holds.
- `E<>` π: for some paths there is a state where π holds, *i.e.*, π is possible.
- $\pi_1 \longrightarrow \pi_2$: for all paths if a state satisfying π_1 is reached then later in the path there is a state satisfying π_2, in other words π_1 leads to π_2.

If a property fails to hold for all paths of a model, UPPAAL can produce a finite path which is a counterexample. Likewise, if the property specifies that some path should exist, then the verifier can produce a witness.

Example: Correct CU. A `CU` receives data from some components as its inputs and computes an output to be used as input for other components. Let `cu_input` be the input and `cu_output` the output of the `CU`. In general, the `CU` computes a function of the input and some internal state, which we shall ignore without loss of generality, that is, `cu_output` $=$ f(`cu_input`), for some function f. When a fault occurs, the `CU` may compute an incorrect value, that is `cu_output` $\neq$ f(`cu_input`), or may not produce any output. In the design, an impulse generator is used to first clear the output and then issue an edge impulse *synclk_xms* every `T` time units to force the `CU` to read its inputs, compute and place the result in `cu_output`.

To simplify the model, we assume that the value of `cu_output` ranges from -1 to 1, and 1 is the correct value, -1 is the incorrect value, while 0 is the cleared value. Initially, the value is 1.

The time spent on the computation is small and can be ignored. Fig.1 displays a system with one non-faulty `CU` and one impulse generator modelled in UPPAAL. The automaton `Impulse` clears the buffer `cu_output` and sends a *synclk_xms*! signal each

T time units that synchronizes with the automaton CU. In the automaton Impulse, a clock x is used to record the time elapsed.

This system satisfies the following two requirements: a) Every period of T time units, the CU computes a new correct value, that is put in cu_output. b) The value 1 is kept for T time units before it is cleared. This can be specified in UPPAAL as the following four properties:

P_1 : $\texttt{A[]}(\texttt{Impulse.x} >= 0 \texttt{ and } \texttt{Impulse.x} <= \texttt{T})$
P_2 : $\texttt{A[]}(\texttt{Impulse.x} > 0 \texttt{ imply } \texttt{cu\_output} == 1)$
P_3 : $\texttt{cu\_output} == 0 \longrightarrow \texttt{cu\_output} == 1$
P_4 : $\texttt{cu\_output} == 1 \longrightarrow \texttt{cu\_output} == 0$

Note that in P_2, cu_output stands for the final output of the system.

Theorem 1. *Properties P_1 – P_4 are valid for the network of the two automata in Fig.1.*

The proof is done by the verifier of UPPAAL. It answers that all the above properties are satisfied by the model.

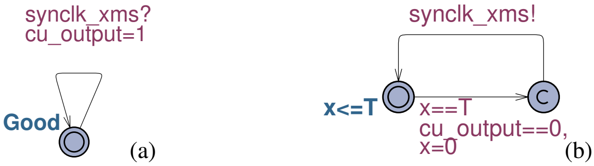


Fig. 1. Non-faulty CU system: (a) CU automaton (b) Impulse automaton

The annotation C to a location makes it ‘committed’; it means that some enabled transition is taken immediately when the location is entered without interleaving of transitions from other automata. This forces the examples to move. The weaker annotation U for urgent could be used as well in this case. It means that no time can pass in the location.

2.2 Modelling the Faults

A component S may encounter faults. These faults can be described by a set F of transitions that interfere with the execution of S by possibly changing the values of variables in S. The transitions in F are nondeterministic transitions from a valid location to an error location. Depending on the fault behaviours, there might be transitions out of the error locations that leads to *failures* or *recovery* to good states.

Definition 2 (Fault-Affected Automaton). *A fault-affected automaton is an automaton with identified faulty transitions to error locations.*

- *$L \cup ERR$ is the union of the two finite and disjoint sets of normal and error locations.*
- *$l_0 \in L$ is the initial location, it is a normal state, thus the system starts correctly.*

The remaining definitions of location invariants and transitions are as in Definition 1.

A transition from a location in $L \cup ERR$ to a location in ERR is called a *fault*. Let F bet the set of all fault transitions, then this automaton is called a F*-affected* automaton.

In general, the design of a fault-tolerant component starts from a fault-free model S, and then analyzing the possible faults to obtain a fault-affected model M. The error locations and fault transitions do not change the fault-free model. For an F-affected automaton M, we have two derived automata:

- $M \backslash F$ is the automaton obtained from M by removing all fault transitions, but still keeping the error locations,
- $M \backslash F^*$ is the automaton obtained from $M \backslash F$ by further inductively removing all transitions from the error locations, and locations that are only reachable from error locations and their outgoing transitions.

The F-affected model of S of S must satisfy the following two *healthiness conditions*

H1. $M \backslash F^* = S$, i.e. they are exactly the same both syntactically and semantically.
H2. $M \backslash F \approx S$, meaning that they are *bisimilar*.

In particular Condition **H2** implies that none of the locations in ERR is reachable in $M \backslash F$, i.e. normal transition cannot lead the system into an error state. This should be model checked by the tool.

Furthermore, the fault-affected model should not satisfy the properties of the fault-free model. If it does, the faults are insignificant for its behaviour. Thus one should model check the fault-affected model with these properties to see that they do not hold. The counterexamples produced by the model checker will contain a fault transition that identifies faults that cause the error. In general there may be several of these and they can be checked systematically be removing error locations one by one, until the fault-free system is reached.

Example Continued: a Faulty CU. In this case, the domain engineers are concerned with the following faults:

- FAULT$_0$: the `CU` enters a deadlock state, and it stops doing anything at all.
- FAULT$_1$: the `CU` enters an error state in which it computes incorrect results.
- FAULT$_2$: the `CU` enters a livelock state and executes only internal actions without outputting a result.

Following the technique described above, we define the fault-effected `CU` shown in Fig.2. In the fault-free location `Good`, the `CU` automaton nondeterministically selects which fault may occur. If FAULT$_0$ occurs, it moves to the error location `Error0` and the automaton deadlocks. The *synclk_xms* is defined as a broadcast channel, so when the `CU` stays in location `Error0`, the `Impulse` automaton can still execute the *synclk_xms*!. The location `Error1` is reached from location `Good` when FAULT$_1$ occurs. In this location, the model outputs incorrect data when the signal *synclk_xms* is issued. Similarly, if FAULT$_2$ occurs, the `CU` goes to location `Error2`, in which it fails to output data when the signal *synclk_xms* is issued.

Also, we add fault transitions from location `Error1` to `Error0` and `Error2`, because a FAULT$_1$ may be followed by one of the other faults.

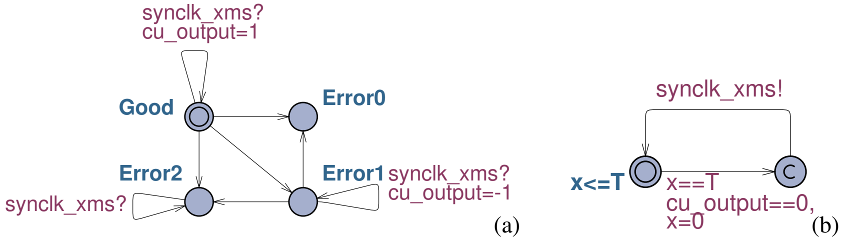


Fig. 2. Faulty CU system: (a) CU automaton b) Impulse automaton

Theorem 2. *None of the properties P_1-P_4 is satisfied by the network of the two automata in Fig.2.*

The proof is a done using UPPAAL.

2.3 Fault-Tolerance

The design of a fault-tolerant model C adds to the fault-affected model M transitions from error states to normal states (new normal states are generally required) such that the specified properties are satisfied. Thus C prevents the faults in M from leading to failures.

Theorem 2 shows the F-affected network in Fig.2 does not tolerate the faults F. Thus we need to add mechanisms as we do for our example in the following Section 3.

A fault-tolerant system C should work both when faults are absent and when they occur. In particular, the design should not assume that faults must occur. Therefore the fault-tolerant design C should satisfy the following properties:

Fault-Tolerance: If the non-faulty model S satisfies a property P then C satisfies P.
Fault monotonicity: For any subset of error locations $E \subseteq \mathit{ERR}$ and any property P of S, $C \setminus E$ satisfies P.

Fault-monotonicity means that tolerance of a fault should not be achieved by "positively using the effects of another fault" as it is not guaranteed to occur.

This point is illustrated in Fig. 3 that shows two "fault-tolerant" automata M and M'. The error locations are Error0 and Error1 that respectively correspond to the occurrence of FAULT$_0$ and FAULT$_1$.They are simplified versions of the CU automaton.

When we consider a property $P : v == 0 \longrightarrow v == 1$; we can see that the model with only the normal states satisfies it. UPPAAL will verify that P holds for M. However, if we remove the location Error0, it no longer holds! In fact, the system M without the location Error0 deadlocks in Error1. It is not fault-monotonic, because it relies on FAULT$_0$ to occur when FAULT$_1$ has occurred. A well-formed, fault-monotonic system is given by M'. This can be checked by removing error locations systematically and checking the property P.

A logical characterization of fault-tolerant refinement is given in [15,16] and these properties are carefully studied.

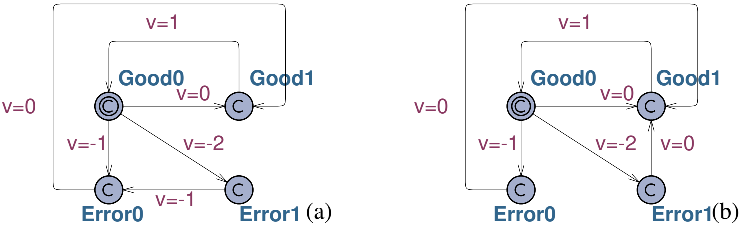


Fig. 3. Fault-affected automaton M and M': (a) Faults are not monotonic in M (b) Faults are monotonic in M'

2.4 Fault Hypotheses

In most cases, fault-tolerance can only be achieved when one makes assumptions about the global properties of the faults modelled by F, such as the maximum number of faulty CUs at a time, and the minimum time between faults. This suggests that the exact modelling of the F-affected system M includes such *fault hypotheses*. As discussed in [15,16], a global behavioural assumption on faults is in general a *safety property* that prevents certain transitions from taking place from some states and thus can be modelled in the guards of the transitions. In fact, since faults do not constrain the actions of the non-faulty model S, the hypotheses can be given by guards of the fault transitions of F. This is exemplified in Section 4, Fig. 7.

3 Design of the Triple Modular Fault-Tolerant Component

The assumed faults of the faulty CU are tolerated by the triple modular system shown in Fig.4. It consists of three CUs, each equipped with a watchdog, and then there are two voters, one arbiter and one impulse generator. The watchdogs are modeled together with their CUs.

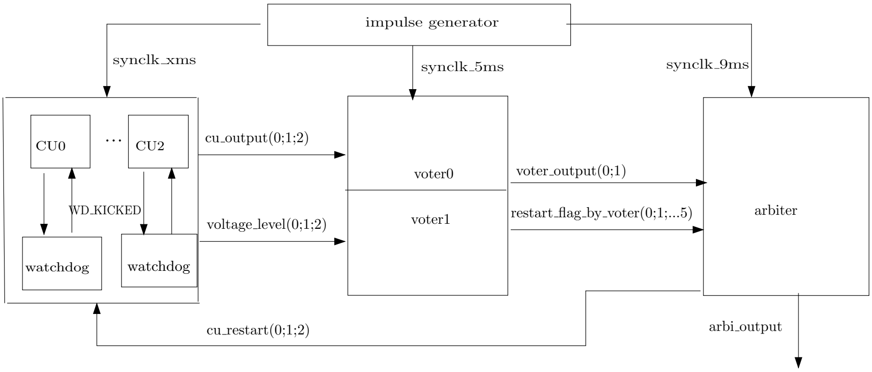


Fig. 4. The triple modular redundancy system

3.1 Impulse Generator

The impulse generator issues edge impulses to force the components to process their inputs. In a cycle, a synchronization impulse *synclk_xms* is generated first to trigger the three CUs to process their inputs simultaneously. After a period CU_PERIOD of time, a *synclk_5xms* impulse is generated to trigger the two voters to process their inputs from the CUs simultaneously. Impulse *synclk_9xms* is produced with a delay of VOTER_PERIOD to activate the arbiter to process its inputs from the voters. A *synclk_xms* impulse is produced again after a period of ARBI_PERIOD time to trigger the CUs in the next cycle. So, all the three types of impulses are generated in every period of T, where T is equal to CU_PERIOD + VOTER_PERIOD + ARBI_PERIOD.

3.2 CU and Watchdog

To make a CU recover from an error state, we introduce a restart mechanism to determine when a CU needs a restart and which component is responsible for triggering a restart from an error state.

When a CU restarts it enters a reset phase and stays there for a period of RESET_PERIOD time before it enters a startup phase, which has a duration of START_PERIOD. In the hardware design, the voltage change of a special pin of the CU signals this procedure. The voltage value stays low (0) for the time of RESET_PERIOD, before it changes to high (1), as described in Fig. 5. We use a Boolean array voltage_level to denote the pin voltage levels of the CUs. When voltage_level[i] is 0, CU$_i$ is in the reset phase, and it is either in the startup or working phase otherwise. The change of voltage_level[i] from 1 to 0 indicates a restart of CU$_i$.

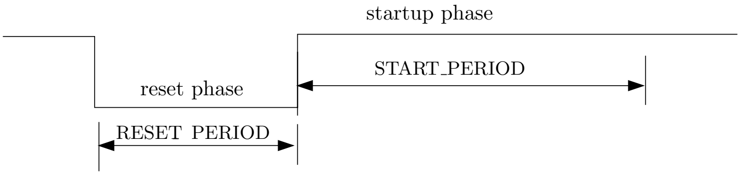


Fig. 5. CU startup procedure

To allow a CU to recover from the error state Error0, we introduce a watchdog for each CU. The timer of the watchdog of the CU starts to count when the CU enters a startup phase. In all the states, except for the state Error0, the CU kicks the watchdog after every period of T time units to set the timer value to WD_KICKED.

We decide that when the timer of the watchdog overflows, it triggers its CU to restart, and thus recover from an error due to the occurrence of FAULT$_0$. However if FAULT$_1$ or FAULT$_2$ occurs, the watchdog is still kicked normally. So the watchdogs cannot be used to detect and recover from occurrences of these faults.

3.3 Voters

To detect errors caused by FAULT$_1$ or FAULT$_2$, we design a voter. The voter receives inputs from the three CUs. Assume that at any time at most one CU is in an error state, the voter votes for the value that is agreed to by at least two CUs, and identifies whether the failed CU need a restart.

As we said before, a voter may fail. We use two voters, VOTER_0 and VOTER_1. The intention is to prevent the voter from being a single point of failure that reduces the overall dependability of the system.

When VOTER_0 and VOTER_1 receive a *synclk_5xms* impulse, they simultaneously start to process the input data to determine which CU works correctly and select the correct result. The time spent on data processing is much smaller than the constant VOTER_PERIOD. This enables a voter to complete the computation before the occurrence of the *synclk_9xms* impulse. We define voting as follows in which buffer voter_output is used to store the output of a voter.

```
if (cu_output[0] == cu_output[1] || cu_output[0]==cu_output[2])
   voter_output = cu_output[0];
if (cu_output[0] != cu_output[1] && cu_output[0] != cu_output[2])
   voter_output = cu_output[1];
```

VOTER_0 is considered the primary, in the sense that if both voters work well, then the arbiter will select the result of VOTER_0 as its output. Only when VOTER_0 goes awry, it selects the result of VOTER_1 as the final output.

Furthermore, the voter has to detect whether a CU is in the reset phase or startup phase. For this, the value of the pin voltage voltage_level[i] is read by the voter. Therefore, in addition to the voting function, each voter has the following functionalities.

Detect CU mode. When VOTER_j , where $j = 0, 1$, reads a correct output from CU_i, it assigns 1 to the Boolean variable cu_normal_by_voter$[3 \times \text{j} + \text{i}]$, which indicates that CU_i is in its *normal mode*. The CU remains in this mode until VOTER_j finds that voltage_level[i] changes from 1 to 0. At this moment VOTER_j assigns 0 to cu_normal_by_voter$[3 \times \text{j} + \text{i}]$, which means that CU_i mode is *abnormal*. The value of cu_normal_by_voter$[3 \times \text{j} + \text{i}]$ remains 0 until VOTER_j reads a correct value from CU_i.

Restart CU. VOTER_j uses the Boolean variable restart_flag_by_voter$[3 \times \text{j} + \text{i}]$ to trigger CU_i to restart. There is no need for the voter to force a CU to restart whenever it reads an incorrect value from the CU. That would render the watchdog ineffective, as it takes some time for its timer to overflow. Therefore we allow a CU to output a sequence of incorrect values before it is restarted. We introduce two positive integers n1 and n2 to control this. When CU_i is in the normal (resp. abnormal) mode, VOTER_j restarts only after having received n1 (resp. n2) incorrect values from CU_i in a row.

3.4 Arbiter

When receiving a *synclk_9xms* impulse, the arbiter acquires the outputs of each voter. Let j be the index of the voter that the arbiter trusts, and arbi_output be the output of the arbiter. In each cycle, the arbiter assigns voter_output[j] to arbi_output, and sends a restart signal cu_restart[i] to CU_i if restart_flag_by_voter$[3 \times \text{j} + \text{i}]$ equals 1. To insure that the arbiter completes the computation before the arrival of the next *synclk_xms* impulse, the computation time of the arbiter must be less than ARBI_PERIOD.

3.5 Design of the Timing Parameters

Taking the real hardware implementation into consideration, for example, a watchdog timer must not overflow when the CU is in the restart procedure, we have the following constraints.

$$\mathtt{WD\_PERIOD} > 2\mathtt{T} + \mathtt{START\_PERIOD} \tag{1}$$

$$\mathtt{WD\_PERIOD} > \mathtt{T} + \mathtt{WD\_KICKED} \tag{2}$$

A CU can restart when either its watchdog overflows or when it gets a restart signal from the arbiter. Therefore, even when the CU is in the startup phase, it can re-enter the reset phase due to a restart signal from the arbiter. This delays the CU to return to normal working mode. To avoid repeated restarts of the CU, we add the following constraints.

$$\mathtt{n2} > \lceil(\mathtt{RESET\_PERIOD} + \mathtt{START\_PERIOD})/\mathtt{T}\rceil + 2 \tag{3}$$

$$\mathtt{RESET\_PERIOD} > \mathtt{T} \tag{4}$$

Finally, a voter can detect incorrect output of a CU when FAULT_0 occurs. If we set the value of the constant WD_PERIOD too high, or the value of the constant n1 too small, the restart of the CU is triggered by the signal sent from the arbiter instead of the overflow of the watchdog. This would make the watchdog ineffective in detecting the occurrences of FAULT_0. That would prevent the engineers from distinguishing the type of faults which occur. A useful statistics, which is used to improve the software, is thus hidden. Thus, the above scenario is not desirable, and it is excluded by the following constraint:

$$\mathtt{n1} > \lfloor \mathtt{WD\_PERIOD}/\mathtt{T} \rfloor + 1 \tag{5}$$

As will show in Section 5, this constraint prevents repeated restart of a faulty CU as well.

4 Model of the Triple Modular Design

The fault tolerant model is now specified by four UPPAAL automata: Impulse, CU, Voter, and Arbiter. Before we explain the construction, we summarize additional fault hypotheses.

4.1 Fault Hypotheses

The triple modular system will only work under some assumptions about the occurrence of faults [15,16]. Thus at any time:

- at most one CU encounters a fault, and
- the minimum time between the occurrences of faults should be long enough to allow the successful recovery of a CU from an error state, and
- no faults occur in the voters and the arbiter.

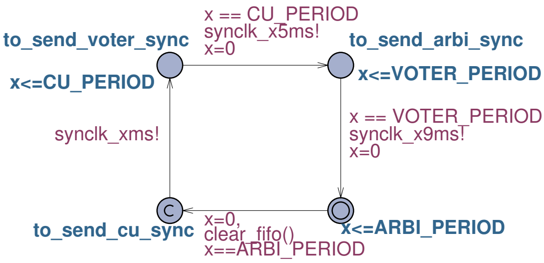


Fig. 6. Impulse automaton

Note that the arbiter will detect a voter fault, but it cannot do anything about it. Here a third voter would be needed, but the approach to model three voters would be similar to the one used here to model three CUs, so we do not include it.

These assumptions will be reflected in the model and some will be checked to hold with the model checking tool.

4.2 Impulse

Figure 6 shows the Impulse automaton which models how the impulse generator periodically produces edge impulses. The clock x records the time between sending of two edge impulses. Every ARBI_PERIOD units of time, clear_fifo() is executed to set cu_output to 0. Immediately after, *i.e.* in zero time, the automaton broadcasts a CU synchronization signal *synclk_xms*! to trigger the three CUs to process their inputs. A *synclk_5xms*! is broadcasted CU_PERIOD time units after. Finally, a *synclk_9xms*! is sent after VOTER_PERIOD time units. Then the cycle repeats.

4.3 CU

Fig.7 shows a CU$_i$ automaton, where $\mathtt{i} = 0, 1, 2$. It models the following phases:

- The restart phase.
- Decision on what to do after receiving a signal.
- Faults and faulty behaviour.

In the automaton a local clock x is used to measure the duration of the restart process and works as the timer of watchdog. Initially CU$_i$ works well and stays in location Good.

The self-loop in location Good models the scenario when a synchronization impulse *synclk_xms* occurs, the CU outputs a correct result and kicks its watchdog. This is the non-faulty model.

An auxiliary global variable cu_faulty_index indicates which CU is faulty. It is initially -1, representing that all the CUs are in location Good. When a fault occurs in CU$_i$, the value of cu_faulty_index becomes i, and it remains so until restarting completes, and the automaton enters location Good again.

Notice the guard cu_faulty_index $==$ -1 on all fault transitions from the normal. It encodes the assumption that at most one CU fails at a time.

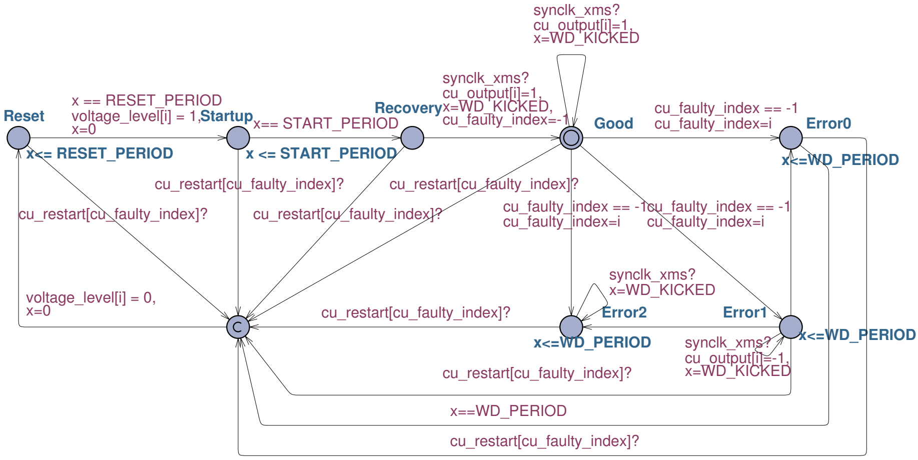


Fig. 7. CU automaton

When no fault has occurred in any other CU, a fault can non-deterministically occur in a CU and move it to an error location: Error0, Error1 or Error2.

In location Error0 the automaton stops working, it will move only when it gets a restart signal or when the the watchdog timer overflows, and then through a committed location immediately to a reset.

An occurrence of FAULT$_1$ moves the automaton to location Error1. In this location, the CU kicks its watchdog but outputs incorrect data when *synclk_xms*! is issued. When the automaton receives a cu_restart[i]? signal from the arbiter, the CU moves through a committed location immediately to a reset.

An occurrence of FAULT$_2$ moves the automaton to location Error2. When *synclk_xms*! is issued, CU kicks its watchdog, but fails to output a value. Similar to the case of FAULT$_0$ and FAULT$_1$, in this location the automaton may receive a cu_restart[i]? signal from the arbiter for a restart.

Because of the parameter constraints discussed in Subsection 3.5, and because of the signal *synclk_xms*! which is generated in a T cycle, delay transitions of none of the locations of Good, Error1 and Error2 will lead to watchdog overflow. Therefore, there are not watchdog overflow transitions in these locations.

The CU reset and startup phases are modelled by the locations Reset and Startup and the transitions between them. As shown in Fig. 5, the automaton resides in location Reset for RESET_PERIOD time units and then jumps to location Startup with voltage_level set to 1. It stays in location Startup for START_PERIOD time units before moving to location Recovery. In location Recovery, it resynchronizes on *synclk_xms*, outputs correct data, kicks the watchdog, and finally sets cu_faulty_index to -1. This means that CU$_i$ has recovered from a fault and it enters location Good.

Since the watchdog timer starts to record the time elapsed when the CU entered a startup phase, and due to the constraints on the hardware parameters, watchdog timer overflows in location Reset or Start or Recovery do not occur.

An over-approximation. In the CU automaton, we have encoded the fault hypothesis about a single CU failure at a time with the guard cu_faulty_index $== -1$. Therefore, the automaton CU$_i$ does not specify the exact duration between two consecutive faults. Instead, it only models the fact that the minimum time when the next fault can occur should be long enough to let the CU output a correct result. This allows more timing behaviours with respect to occurrences of faults in the model.

4.4 Voter

The automaton Voter$_j$ is shown in Fig. 8(a), where j $= 0, 1$. We introduce a local variable cu_error_time[i] to record the number of incorrect values that the voter read from CU$_i$. Initially the automaton stays in location Idle. When it receives a *synclk_5ms* signal, it calls two functions: fault_check() and vote(). For each CU, function fault_check() computes the following steps.

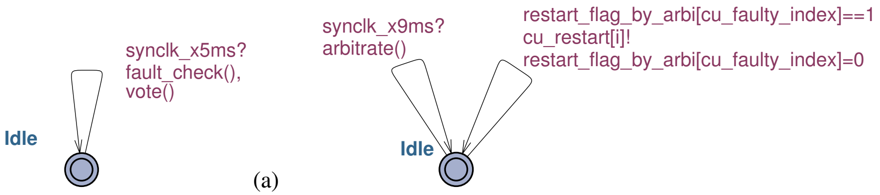


Fig. 8. (a) Voter automaton (b) Arbiter automaton

1. It checks if there is a restart of CU$_i$. A local variable lvoltage_level[i] is used to store the value of voltage_level[i] from the last cycle.
2. It checks the data read from cu_output[i] and decides if CU$_i$ needs a restart.

The function vote() completes the voting algorithm. It compares results from the three CUs, and outputs the majority value to the buffer vote_output.

```
void fault_check()
{  int i;
   for(i = 0; i < 3; i++)
      restart_flag_by_voter[i + j * 3] =0;
   for(i = 0; i < 3; i++)
   {  if (lvoltage_level[i]==1 && voltage_level[i]==0) //edge jumps
         { cu_error_time[i] = 0;
           cu_normal_by_voter[i]=0; //judge CU i is in abnormal mode
         }
      lvoltage_level[i] = voltage_level[i]; //lvoltage_level is updated
      if(cu_output[i] !=1) //data from CU i is incorrect
        { cu_error_time[i]++;
          if (cu_normal_by_voter[i] == 1) //if CU i is in normal mode
            {if (cu_error_time[i] >= n1)
             {cu_error_time[i] = n1;
              restart_flag_by_voter[i + j * 3] = 1; //CU i needs a restart
             }
            }
          else // CU i is in abnormal mode
            {if (cu_error_time[i] >= n2)
```

```
              {cu_error_time[i] = n2;
               restart_flag_by_voter[i + j * 3] = 1; //CU i needs a restart
              }
             }
          }
        else  // data from CU i is correct
          { cu_error_time[i] = 0;
            cu_normal_by_voter[i]=1; //judge CU i is in normal mode
          }
     }
 }
```

```
 void vote()
 { if (cu_output[0]==cu_output[1])
      voter_output[j]=cu_output[0];
   else if (cu_output[0]==cu_output[2])
      voter_output[j]=cu_output[0];
   else voter_output[j]=cu_output[1];
 }
```

4.5 Arbiter

The Arbiter is modelled as the automaton in Fig. 8(b), it uses a local Boolean variable `restart_flag_by_arbi[i]` to express if $\texttt{CU}_i$ needs to restart. Whenever *synclk_x9mx* is issued, one self-loop transition executes the function `arbitrate()` .

```
void arbitrate()
{ int i;
  for (i=0;i<=2;i++)
    restart_flag_by_arbi[i]=restart_flag_by_voter[i];
  arbi_output=voter_output[0];
}
```

The other self-loop transition models that if $\texttt{CU}_{cu\_faulty\_index}$ should restart, a restart signal is sent.

4.6 Checking of the Triple Modular Design in UPPAAL

We decide on the following values for the different timing constants; these values satisfy the parametric constraints in Subsection 3.5. These definitions are copied almost verbatim in the UPPAAL declaration section of our model.

```
CU_PERIOD:    0.5  ms  (time difference between synclk_xms and synclk_5xms)
VOTER_PERIOD: 0.4  ms  (time difference between synclk_5xms and synclk_9xms)
ARBI_PERIOD:  0.1  ms  (time difference between synclk_9xms and synclk_xms)
RESET_PERIOD  100  ms  (the period that CU stays in reset phase)
START_PERIOD: 300  ms  (the period that CU stays in startup phase)
WD_KICKED:    450  ms  (the value watchdog timer kicked)
WD_PERIOD     500  ms  (the maximum time watchdog timer can record)
n1:  30   (the number of incorrect data voter allows if voter judges CU normal)
n2:  550  (the number of incorrect data voter allows if voter judges CU abnormal)
```

The properties Q_1 – Q_4 below for the fault-tolerant system ensure the correctness properties P_1 – P_4 of the fault-free system in Subsection 2.1. Indeed, the final output of the fault-tolerant system is no longer `cu_output`, but `voter_output` and `arbi_output`. This correctness is based on the equivalence of any of the `CU`s.

Q_1: `A[](Impulse.x >= 0 and Impulse.x <= T)`
Q_2: `A[](voter_output[0] == 1 and arbi_output == 1)`

Q_3: $\texttt{cu\_output}[0] == 0 \to \texttt{cu\_output}[0] == 1$
Q_4: $\texttt{cu\_output}[0] == 1 \to \texttt{cu\_output}[0] == 0$

Checking these properties only takes a few seconds. The verification results in UPPAAL reveal that all the above properties are satisfied by the model of the design.

In UPPAAL, we can check the above system using different instantiations of its parameters. However, each checking can only verify a single instance of the system, whereas we would like to establish correctness for all instantiations of its parameters. This motivates the parametric analysis of the system in the following section.

5 Parametric Analysis of the Triple Modular Design

We are interested in using the parameter constraints in Subsection 3.5 to manually prove the correctness of $Q_1 - Q_4$. In terms of several steps of abstraction, we prove that the desired properties hold under the assumption that the values of the parameters meet the constraints.

5.1 The Composed System

The full system can now be described as the network of the three CU automaton, the two Voter automata, the Arbiter automaton, and the Impulse automaton, with all synchronization actions hidden. It can be computed as a product automaton subject to the synchronizations. It is shown as the automaton A in Fig 9.

The clock x is used to record the time passing between sending two edge impulses. The clock y is used to measure the waiting time of the restart process and as the timer of watchdog. The error locations Error00, Error0C, Error01 and Error02 imply that FAULT$_0$ occurs in $\texttt{CU}_{\texttt{cu\_faulty\_index}}$. While the error locations Error10, Error1C, Error11 and Error12 imply that FAULT$_1$ occurs in the CU, and the error locations Error20, Error2C, Error21 and Error22 imply that FAULT$_2$ occurs in the CU.

The synchronization *synclk_xms* yields τ_0, while the synchronization *synclk_5xms* yields τ_1, and the synchronization *synclk_9xms* yields τ_2. The urgent synchronization cu_restart yields internal action τ_3. To ensure that τ_3 is immediately executed we introduce the committed locations from DD0 to DD5, and from these locations we judge if restart_flag_by_arbi[cu_faulty_index] equals 1. In this case the actions are taken from these locations to reset the faulty CU.

To simplify the composed system, when τ_0 is taken, we use different functions to update the outputs of the three CUs. If cu_faulty_index equals -1, this implies that all the three CUs work normally, the outputs of the three CUs are described by the function good_cu_output(). Otherwise, if FAULT$_0$ or FAULT$_2$ occurs, the outputs of the three CUs are described by function error02_cu_output(); if FAULT$_1$ occurs, the outputs are described by function error1_cu_output().

```
void error02_cu_output()
{  int i;
   for(i = 0; i < 3; i++)
    {  if (i!=cu_faulty_index)
          cu_output[i] = 1;
    }
}
```

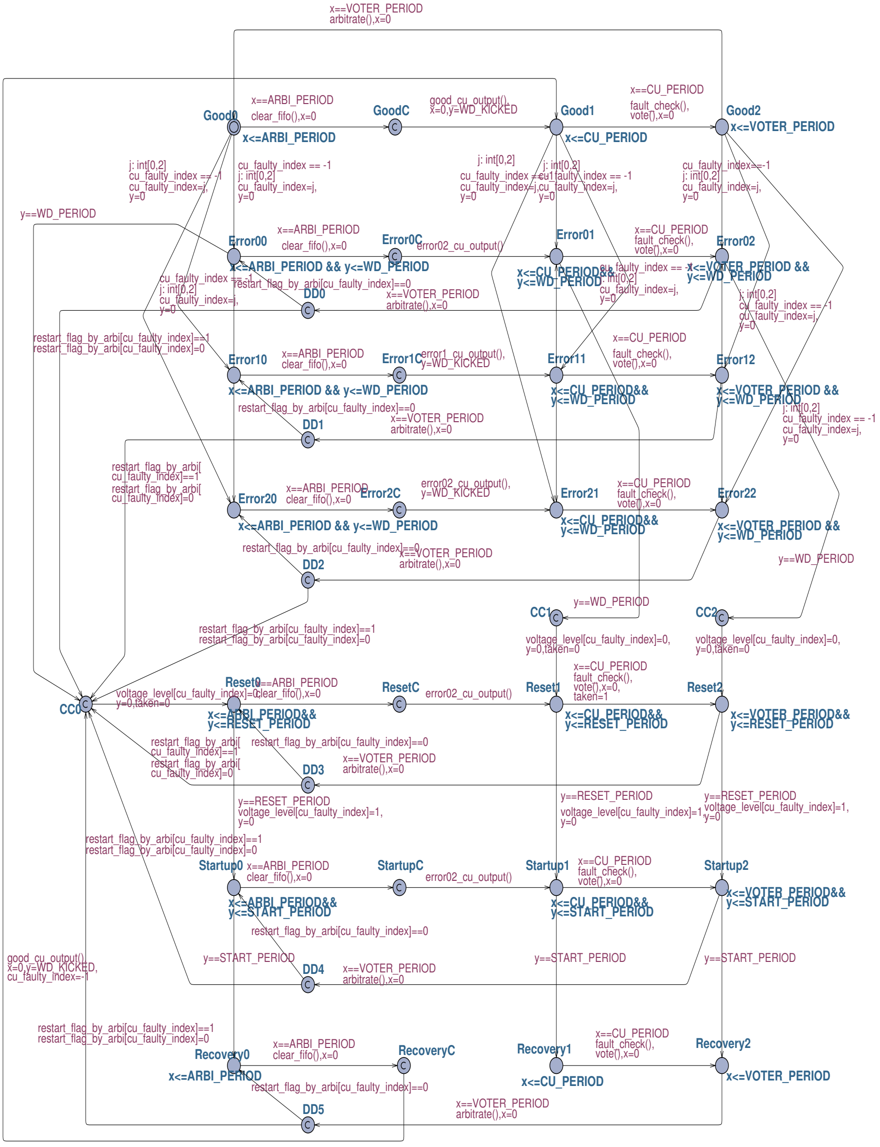


Fig. 9. The automaton A

```
void error1_cu_output() {    int i;
    for(i = 0; i < 3; i++)
    {   if (i!=cu_faulty_index)
          cu_output[i] = 1;
        else cu_output[i] =-1;
    }
}
```

```
void good_cu_output() {   int i;
    for(i = 0; i < 3; i++)
      cu_output[i] = 1;
}
```

Obviously, if A is in the location $\mathtt{CU}_{\mathtt{cu\_faulty\_index}}$.Recovery, when τ_0 occurs, all the three CUs output a correct value. Thus the outputs of the CUs are implemented by good_cu_output().

In order to ease the manual invariant proof in the following subsection, we introduce an auxiliary Boolean variable taken. It indicates whether the function fault_check() is executed when A enters the location Reset0, Reset1 or Reset2. This variable is such that:

- it is initially 0,
- when the transition from Reset1 to Reset2 is taken, the value becomes 1, and
- when the transition to location Reset0 or Reset1 or Reset2 is taken, the value become 0.

We now establish that the composed automaton A satisfies the properties $Q_1 - Q_4$ if the parametric constraints are met.

5.2 Parametric Analysis

We want now to prove that the properties Q_1, Q_2, Q_3 and Q_4 hold in A. Clearly Q_1 holds in A. In order to prove the other properties, we introduce in this section four automata A_1, A_2, A_3 and A_4 based on A, each one of them simplifying A in order to prove one property. A_1 is a timed automaton, which gather the different kind of faults in A, while preserving the error locations as well as other information. A_2 is also a timed automaton, in which using the parameter constraints, we omit the restart actions in A_1 when a faulty CU is in its restart and recovery process. A_3 is further acquired by merging several locations of A_2 in which A_2 stays to wait for the occurrence of τ_0, τ_1 and τ_2, as a single location. Finally, A_4 is obtained by removing all timing information in A_1.

In the following, we prove Q_3 by proving Q_{30} and Q_{31} respectively in A_1 and A_3.

Q_{30}: $\mathtt{cu\_faulty\_index} == -1$ and $\mathtt{cu\_output}[0] == 0 \rightarrow \mathtt{cu\_output}[0] == 1$
Q_{31}: $\mathtt{cu\_faulty\_index} == 0$ and $\mathtt{cu\_output}[0] == 0 \rightarrow \mathtt{cu\_output}[0] == 1$

The First Intermediate Automaton A_1. Intermediate automaton A_1 is displayed in Fig. 10. The timed automaton is identical to the composed system, except that the error locations are merged to one location. All the transitions leading from one of the merged locations now comes from the super location. This is therefore a correct abstraction of A according to the following definition of timed refinement (abstraction is the converse of refinement).

Definition 3. *Let B and C be two timed automata, s_0, t_0 be initial states of B and C, Q_B, Q_C be states of B and states of C. A timed simulation from B to C is a binary relation $R \subseteq Q_B \times Q_C$ written $B \sqsubseteq C$, provided for all $(s, t) \in R$, $s[V_C] = t[V_C]$, and*

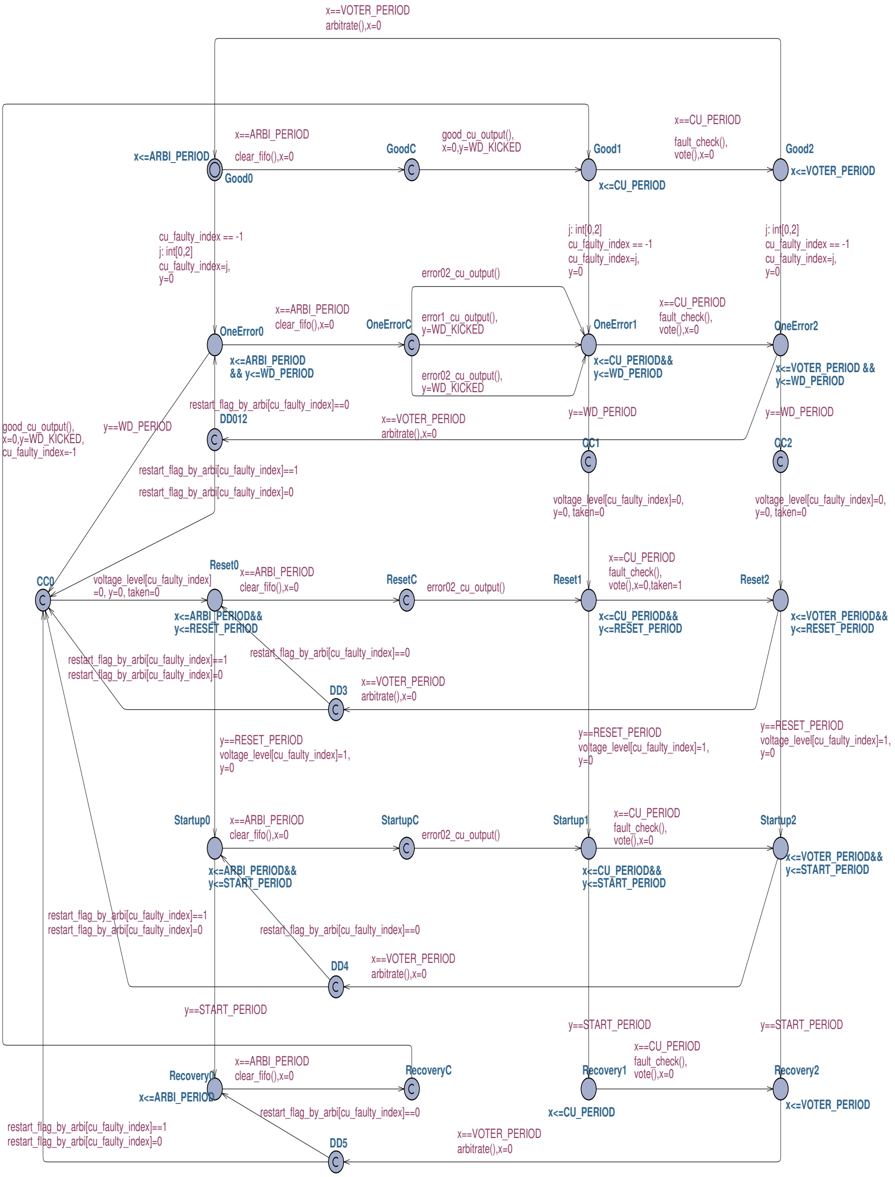


Fig. 10. Intermediate automaton A_1

- $(s_0, t_0) \in R$
- *if* $(s,t) \in R$ *and* $s \xrightarrow{a} s'$ *in* B*, then* $t \xrightarrow{a} t'$ *in* C *and* $(s',t') \in R$*,*
- *if* $(s,t) \in R$ *and* $s \xrightarrow{\epsilon(d)} s'$ *in* B*, then* $t \xrightarrow{\epsilon(d)} t'$ *in* C *and* $(s',t') \in R$

In connection with proof of refinements below, we use the following conventions: the location l appearing in the definition of relation R, stands for a state that consists of the location l as well as a valuation for state variables.

Lemma 1. $A \sqsubseteq A_1$.

Proof. The relation of Definition3 is given by $R = (s, \psi(s))$ for any s in Q^A, where

$$\forall s \in Q^A \ \psi(s) = \begin{cases} \texttt{OneError0} & \text{if } s \in \{\texttt{Error00}, \texttt{Error10}, \texttt{Error20}\} \\ \texttt{OneError1} & \text{if } s \in \{\texttt{Error01}, \texttt{Error11}, \texttt{Error21}\} \\ \texttt{OneError2} & \text{if } s \in \{\texttt{Error02}, \texttt{Error12}, \texttt{Error22}\} \\ \texttt{OneErrorC} & \text{if } s \in \{\texttt{Error0C}, \texttt{Error1C}, \texttt{Error2C}\} \\ \texttt{DD012} & \text{if } s \in \{\texttt{DD0}, \texttt{DD1}, \texttt{DD2}\} \\ s & \text{otherwise} \end{cases}$$

Let us prove that R is a simulation from A to A_1. First, since the initial states are identical, they clearly belong to R. Now, let s and s' be two states of A, t a state of A_1 and a an action such that $s \ R \ t$ and $s \xrightarrow{a} s'$. If $s \in \{\texttt{Error00}, \texttt{Error10}, \texttt{Error20}\}$, then by definition of R, $t = \texttt{OneError0}$ and two cases are possible.
$i)$ $s = \texttt{Error00}$ and A performs an action a to CC0.
In this case, $s' = \texttt{CC0}$ and $\texttt{OneError0} \xrightarrow{a} \texttt{CC0}$ in A_1, and we can conclude by definition of R.
$ii)$ $s \in \{\texttt{Error00}, \texttt{Error10}, \texttt{Error20}\}$ and A performs an action a, when the timer expires: $\texttt{x} = \texttt{ARBI\_PERIOD}$. In this case, $s' \in \{\texttt{Error0C}, \texttt{Error1C}, \texttt{Error2C}\}$ and $\texttt{OneError0} \xrightarrow{a} \texttt{OneErrorC}$ in A_1 and we can conclude by definition of R.

We proceed in the same way for the other states and since it is clear that R is also a relation for time delay transitions, we can conclude that R is simulation from A to A_1.

Lemma 2. *Q_{30} holds in A_1.*

Proof. Q_{30} can be straightforwardly proved in A_1. When all the three CUs are good, A_1 is deterministic. As function good_cu_output() is taken infinitely often, Q_{30} holds in A_1.

Lemma 3. *Q_4 holds in A_1.*

Proof. This property clearly holds as there is no infinite sequence of instantanous transitions in A_1 and the function clear_fifo() is taken each T cycle.

The Second Intermediate Automaton A_2. If we take into account the constraints defined in Section 3.5, then the transitions from DD3 to CC0, from DD4 to CC0 and from DD5 to CC0 are never enabled in A_1. Indeed we can prove the following invariants:

Lemma 4

$$\begin{aligned} &\texttt{I1} : \texttt{A[]}\ A_1\texttt{.DD3 imply restart\_flag\_by\_arbi[cu\_faulty\_index]} == 0 \\ &\texttt{I2} : \texttt{A[]}\ A_1\texttt{.DD4 imply restart\_flag\_by\_arbi[cu\_faulty\_index]} == 0 \\ &\texttt{I3} : \texttt{A[]}\ A_1\texttt{.DD5 imply restart\_flag\_by\_arbi[cu\_faulty\_index]} == 0 \end{aligned}$$

Proof. Let α be the value of clock $\mathtt{y}$ that record the time point when $\mathtt{taken}$ is changed from 0 to 1. We first need to introduce some other invariants:

$$\begin{array}{ll}
\mathtt{I4}: & \mathtt{A[]}\ (\mathtt{A_1.OneError0} \vee \mathtt{A_1.OneError1} \vee \mathtt{A_1.OneError2}) \wedge \mathtt{y} > \mathtt{T} \\
& \quad \mathtt{imply}\ \mathtt{cu\_output[cu\_faulty\_index]} \neq \mathtt{1} \\
\mathtt{I5}: & \mathtt{A[]}\ (\mathtt{A_1.Reset0} \vee \mathtt{A_1.Reset1} \vee \mathtt{A_1.Reset2}) \\
& \quad \mathtt{imply}\ (\mathtt{cu\_output[cu\_faulty\_index]} \neq \mathtt{1} \\
& \quad \wedge\ \mathtt{voltage\_level[cu\_faulty\_index]} == \mathtt{0}) \\
\mathtt{I6}: & \mathtt{A[]}\ \mathtt{A_1.Reset2} \wedge \mathtt{taken} == \mathtt{1} \wedge \mathtt{y} == \alpha \\
& \quad \mathtt{imply}\ (\mathtt{cu\_normal\_by\_voter[cu\_faulty\_index]} == \mathtt{0} \\
& \quad \wedge\ \mathtt{cu\_error\_time[cu\_faulty\_index]} == \mathtt{0}) \\
\mathtt{I7}: & \mathtt{A[]}\ \mathtt{A_1.Reset2} \wedge \mathtt{taken} == \mathtt{1} \\
& \quad \mathtt{imply}\ \mathtt{restart\_flag\_by\_voter[cu\_faulty\_index]} == \mathtt{0} \\
\mathtt{I8}: & \mathtt{A[]}\ \mathtt{A_1.DD3} \wedge\ \mathtt{taken} == \mathtt{1} \\
& \quad \mathtt{imply}\ \mathtt{restart\_flag\_by\_arbi[cu\_faulty\_index]} == \mathtt{0} \\
\mathtt{I9}: & \mathtt{A[]}\ (\mathtt{A_1.OneError0} \vee \mathtt{A_1.OneError1} \vee \mathtt{A_1.OneError2}) \\
& \quad \mathtt{imply}\ \mathtt{cu\_normal\_by\_voter[cu\_faulty\_index]} == \mathtt{1} \\
\mathtt{I10}: & \mathtt{A[]}\ \mathtt{A_1.DD3} \wedge\ \mathtt{taken} == \mathtt{0} \\
& \quad \mathtt{imply}\ \mathtt{restart\_flag\_by\_arbi[cu\_faulty\_index]} == \mathtt{0}
\end{array}$$

These invariants are verified by the inspection of the automaton, and under these assumptions, we can prove the invariants $\mathtt{I1}, \mathtt{I2}$ and $\mathtt{I3}$.

1. We easily prove $\mathtt{I1}$ by the invariants $\mathtt{I8}$ and $\mathtt{I10}$.
2. In order to prove $\mathtt{I2}$, we need the following invariants:

   $$\begin{array}{l}
   \mathtt{I11}: \mathtt{A[]A_1.Startup2} \wedge \mathtt{taken} == \mathtt{1} \\
   \quad \mathtt{imply}\ \mathtt{restart\_flag\_by\_voter[cu\_faulty\_index]} == \mathtt{0} \\
   \mathtt{I12}: \mathtt{A[]A_1.Startup2}\ \mathtt{imply}\ \mathtt{restart\_flag\_by\_voter[cu\_faulty\_index]} == \mathtt{0}
   \end{array}$$

   As $\mathtt{Error02\_cu\_output()}$ is executed each $\mathtt{T}$ cycle when A_1 is in reset or startup process, by $\mathtt{I6}$ and constraint (3), we have $\mathtt{I11}$. Due to constraint (4), $\mathtt{I12}$ is acquired. From $\mathtt{I12}$, obviously we have $\mathtt{I2}$.
3. In order to prove $\mathtt{I3}$, we need the following invariant:

   $$\begin{array}{l}
   \mathtt{I13}: \mathtt{A[]A_1.Recovery2} \wedge \mathtt{taken} == \mathtt{1} \\
   \quad \mathtt{imply}\ \mathtt{restart\_flag\_by\_voter[cu\_faulty\_index]} == \mathtt{0} \\
   \mathtt{I14}: \mathtt{A[]A_1.Recovery2}\ \mathtt{imply}\ \mathtt{restart\_flag\_by\_voter[cu\_faulty\_index]} == \mathtt{0}
   \end{array}$$

   Still as $\mathtt{Error02\_cu\_output()}$ is executed each $\mathtt{T}$ cycle when A_1 is in reset or startup process, by $\mathtt{I6}$ and constraint (3), we have $\mathtt{I13}$. Due to constraint (4), $\mathtt{I14}$ is acquired. From $\mathtt{I14}$, obviously we have $\mathtt{I3}$.

We can conclude that the invariants $\mathtt{I1}, \mathtt{I2}$ and $\mathtt{I3}$ hold for A_1.

Since the three previous invariants hold, we can clearly remove from A_1 the transitions from $\mathtt{DD3}$ to $\mathtt{CC0}$, from $\mathtt{DD4}$ to $\mathtt{CC0}$ and from $\mathtt{DD5}$ to $\mathtt{CC0}$, thus defining the automaton A_2.

Notice that following invariant clearly holds for A_2:

$$\mathtt{I15}: \mathtt{A[]}\ (\mathtt{A_2.Recovery0} \vee \mathtt{A_2.Recovery1} \vee \mathtt{A_2.Recovery2})\ \mathtt{imply}\ \mathtt{y} \leq \mathtt{T}$$

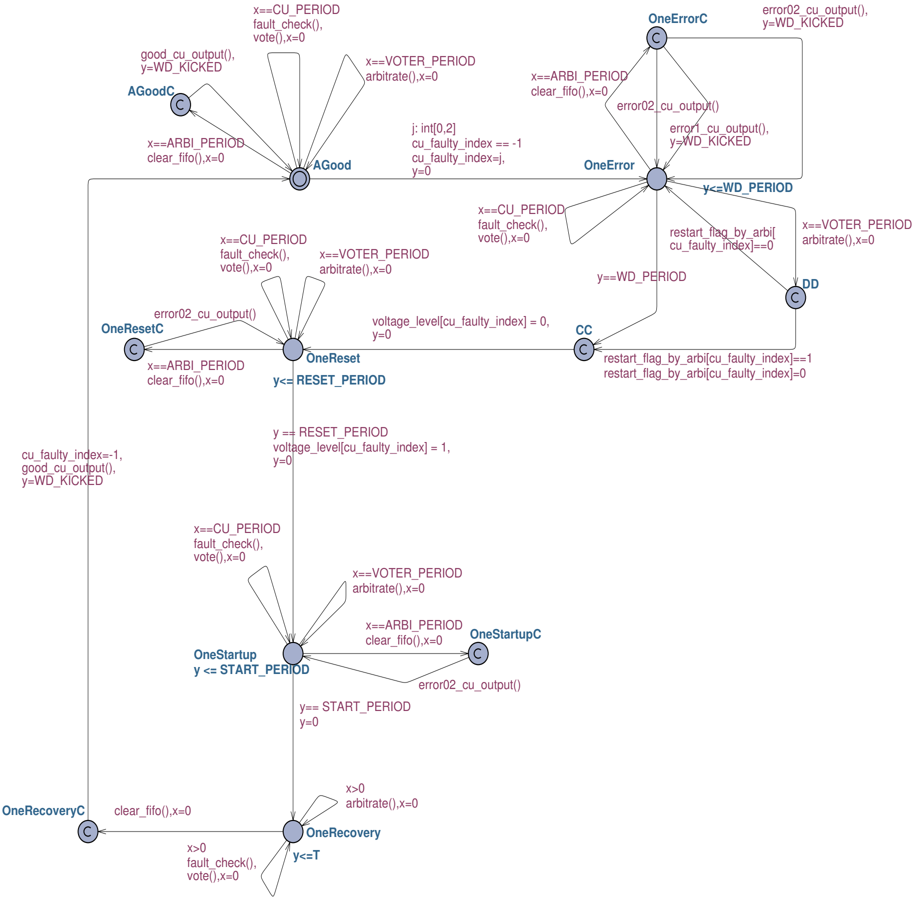


Fig. 11. Intermediate automaton A_3

The Third Intermediate Automaton A_3. From the automaton A_2 defined in the previous subsection, we define the automaton A_3 as follows:

- roughly speaking, each “row” of A_2 is merged into one super location,
- the variable taken is removed, and
- the guard from OneRecovery to OneRecoveryC is weakened.

The automaton A_3 is described in Fig. 11. Notice that from the invariant I15 previously defined and the definition of A_3, the following invariant clearly holds:

$$\texttt{I16} : \texttt{A[]}\ \texttt{A}_3\texttt{.OneRecovery imply y} \leq \texttt{T}$$

Let Q^{A_2}, Q^{A_3} be the states of automata A_2 and A_3, $\psi_1 : Q^{A_2} \rightarrow Q^{A_3}$ be the following mapping:

$$\forall s \in Q^{A_2}\ \psi_1(s) = \begin{cases} \texttt{AGood} & \text{if } s \in \{\texttt{Good0}, \texttt{Good0}, \texttt{Good2}\} \\ \texttt{OneError} & \text{if } s \in \{\texttt{OneError0}, \texttt{OneError1}, \texttt{OneError2}\} \\ \texttt{OneReset} & \text{if } s \in \{\texttt{Reset0}, \texttt{Reset1}, \texttt{Reset2}\} \\ \texttt{OneStartup} & \text{if } s \in \{\texttt{Startup0}, \texttt{Startup1}, \texttt{Startup2}\} \\ \texttt{OneRecovery} & \text{if } s \in \{\texttt{Recovery0}, \texttt{Recovery1}, \texttt{Recovery2}\} \\ \texttt{DD} & \text{if } s \in \{\texttt{DD012}, \texttt{DD3}, \texttt{DD4}, \texttt{DD5}\} \\ \texttt{CC} & \text{if } s \in \{\texttt{CC0}, \texttt{CC1}, \texttt{CC2}\} \\ \texttt{AGoodC} & \text{if } s \in \{\texttt{GoodC}\} \\ \texttt{OneStartupC} & \text{if } s \in \{\texttt{StartupC}\} \\ \texttt{OneRecoveryC} & \text{if } s \in \{\texttt{RecoveryC}\} \\ s & \text{otherwise} \end{cases}$$

Lemma 5. $A_2 \sqsubseteq A_3$. *More specifically the relation* $R = (s, \psi_1(s))$ *for any* s *in* Q^{A_2}, *defines a simulation from* A_2 *to* A_3.

Proof. Routine proof similar to the proof of Lemma 1.

Lemma 6. Q_{31} *holds in* A_3.

Proof. Let s be a state such that $s \models \texttt{cu\_faulty\_index} == \texttt{0}$ and $\texttt{cu\_output}[\texttt{0}] == \texttt{0}$ and let us prove that $\texttt{cu\_output}[\texttt{0}]$ will be eventually equal to 1. By definition of A_3, only four cases are possible for s:
$i)$ $s = \texttt{OneRecovery}$.
Here, the self-loop transitions l_1 and l_2 and the time-delay transition in OneRecovery does not set $\texttt{cu\_output}[\texttt{0}]$ to 1. However, since the maximum time A_2 stays in this location is T, and since in both transitions l_1 and l_2, we have guard $\texttt{x} > \texttt{0}$ and reset $\texttt{x} = \texttt{0}$, so each occurrence of the transition l_1 or l_2 implies the increment of the clock y. Thus for any infinite path from s, we only have a finite number of the self-loop transitions so that eventually we must reach the location OneRecoveryC, which is immediately left to output a good value of $\texttt{cu\_output}[\texttt{0}]$ by function good_cu_output() and we can conclude that $\texttt{cu\_output}[\texttt{0}] == \texttt{1}$ eventually holds.
$ii)$ $s = \texttt{OneStartup}$.
In this case, since A_2 stays in this location for a maximum time START_PERIOD, and because only after a non-zero time delay can any self-loop transition takes place, thus for any infinite path from s, we only have a finite number of the self-loop transitions and eventually the location OneRecovery is reached and we can conclude by $i)$.
$iii)$ $s = \texttt{OneReset}$.
Likewise, if A_2 stays in OneReset, the location OneStartup is eventually reached and we can conclude by $ii)$.
$iv)$ $s = \texttt{OneErrorC}$.
In the same way, if A_2 is in the state OneErrorC, the state OneReset is eventually reached and so we can conclude by $iii)$.

The Fourth Intermediate Automaton A_4. Lastly, we define the automaton A_4 by removing all the clock variables as well as all the references to these variables in locations and transitions of A_1. Since the property Q_2 does not depend on time parameters, if Q_2 holds for A_4, it also holds for A_1.

Lemma 7. *Q_2 holds for A_4.*

Proof. Done using UPPAAL.

We are now in position to prove the following proposition:

Proposition 1. *The properties Q_1, Q_2, Q_3 and Q_4 hold for A.*

Proof.

1. As we said in Section 5.2, the property Q_1 clearly holds for A.
2. By the Lemma 7, we have proved that Q_2 holds for A_4 and so for A_1, since A_4 is equal to A_1 without time parameters and since Q_2 does not depend on time parameters. Moreover, by Lemma 1 we have $A \sqsubseteq A_1$, so we can conclude that Q_2 holds for A.
3. By the Lemma 2, Q_{30} holds for A_1 and so, by Lemma 1, it also holds for A. By the Lemma 6, Q_{31} holds for A_3 and by Lemma 5, we have $A_2 \sqsubseteq A_3$ and it follows that Q_{31} also holds for A_2. Moreover, we have proved by the Lemma 4 A_2 is equivalent to A_1, since we only remove transitions which are never enabled and so Q_{31} also holds for A_1. Still by the Lemma 1, we know that Q_{31} holds for A. Moreover, by the definition of Q_{30} and Q_{31}, since they both hold for A, we can conclude that Q_3 holds for A.
4. By the Lemma 3, Q_4 holds for A_1, and by the Lemma 1, we can finally conclude that Q_4 also holds for A.

5.3 The Parameter Constraints Are Necessary

Above, we have proved that the parameter constraints in Subsection 3.5 are sufficient to prove the desired properties of the system. However, it could be the case that they are too narrow, therefore a number of counterexamples below shows that they are also necessary. Such that we can conclude that they are the best we can do, they are both sufficient and necessary.

From A it is obvious that if there is a restart action in DD3 or DD4 or DD5, then there exits a loop that consists of location Reset0. This loop may generate an infinite path and along it the output value of a faulty CU is always different from 1. This violates property Q_3. Therefore, we should avoid the restart action from DD3 or DD4 or DD5. That is, we should avoid repeated restart of a faulty CU, Constraint (3) is needed to avoid the restart phenomenon from location DD4 or DD5. When a CU enters abnormal mode, fault_check() triggers it to restart if cu_error_time equals n2. Suppose the constraint is not satisfied, then consider the case that at some time t, A enters location Startup0 from Reset0. The value of cu_normal_by_voter[0] becomes 0 when fault_check() is first executed since time t . Because fault_check() is executed each T cycle, after at most n2 × T time units from time point t, the value of restart_flag_by_arbi[cu_faulty_index] becomes 1. By the assumption, A stays in startup and can take a restart in location DD4, which violates our properties. Similarly, the restart phenomenon from DD5 exists if the constraints is not satisfied. In the inequality "+2 " is used to give more robustness against skewed timers.

Constraint (4) is also needed to avoid a restart from location DD4. To illustrate this, suppose at some time point t, A enters location Reset and the current state s satisfies the formula:

$$\begin{aligned}\texttt{lvoltage\_level}[\texttt{cu\_faulty\_index}] == 1 \wedge \texttt{voltage\_level} == 0 \\ \wedge \texttt{cu\_normal\_by\_voter}[\texttt{cu\_faulty\_index}] == 1\end{aligned}$$

Assume $\texttt{T} > \texttt{RESET\_PERIOD}$, then the automaton may move to Startup0 from Reset0 with taken being equal to 0. Since both $\texttt{voltage\_level}[\texttt{cu\_faulty\_index}]$ and $\texttt{lvoltage\_level}[\texttt{cu\_faulty\_index}]$ are set to 1, after some time, $\texttt{fault\_check}()$ judges that $\texttt{cu\_normal\_by\_voter}[\texttt{cu\_faulty\_index}]$ remains 1. If we have the constraint $\texttt{START\_PERIOD} > \texttt{n1} \times \texttt{T}$, then the $\texttt{restart\_flag\_by\_arbi}[\texttt{cu\_faulty\_index}]$ could be set to 1. So, there exits the scenario that A can transit from DD4 to CC0, which violates the desired property.

Constraint (5) is required to avoid the restart action from DD3. Consider the case that Reset2 is reached and $\texttt{cu\_normal\_by\_voter}[\texttt{cu\_faulty\_index}]$ equals 1. According to the restart mechanism, when a CU is in the normal mode, $\texttt{fault\_check}()$ triggers it to restart, after the CU having continuously produced n1 incorrect values, that is, cu_error_time equals n1. If the constraint is not met, when clock x equals VOTER_PERIOD, the execution of function arbitrate from location Reset2 to DD3 may set $\texttt{restart\_flag\_by\_arbi}[\texttt{cu\_faulty\_index}]$ to 1. Thus the restart action from DD3 is not avoided.

6 Conclusion and Future Work

We have presented a systematic approach to the development and the verification of fault-tolerant components, and illustrated it on a design of a triple modular fault-tolerant system. The verification uses the model checking tool UPPAAL. The examples were given by domain engineers in the aerospace field. Our experience shows that formal modeling and verification are applicable to verification problems in practical fault-tolerant systems. Such designs are in general hard to test for software and system engineers, and their solution require delicate techniques in modelling from experts in the area of formal verification.

The results also show that UPPAAL is able to model the system faithfully. Especially, the c-like syntax in the tool, which in our model includes the functions fault_check(), clear_fifo(), vote() and arbitrate(), will certainly be familiar to the domain engineers and can be easily translated to Verilog hardware language during the implementation.

In the modelling, we ignored the computation time for each component. One reason is that the computation time is relatively small compared with the time difference between two consecutive activations, so this does not affect the correctness properties.

The other reason is that when we use a model with multiple clocks to describe the computation time for each component, we encounter the state space explosion problem. To further reduce state space, as demonstrated by [10], which has been successfully applied in an Zero Configuration protocol, we may use dead variable reduction to abstract our model. Dead variable reduction is a well known static analysis technique, that has for instance been studied in the PhD thesis of Yorav [21]. A variable *v* is said to be dead at a location *l* if on every execution path from *l*, *v* is defined before it is used, or is never used at all. Clearly, systems that only differ in the values of dead variables are equivalent in a very strong sense (bisimilar). In our model, array `cu_output` is dead in location `Idle` in automaton `Voter`, and can be reset to zero after completing functions `fault_check`() and `vote`() upon entering this location. Another example is that the arrays `voter_output` and `restart_flag_by_voter` are dead in location `Idle` in automaton `Arbiter`, and can be reset to a default value upon occurrence of the transition to this location.

The current available version of UPPAAL does not support parametric analysis, an inductive method therefore have been used in the invariant proof to solve this problem. However since a manual, operational proof often contains small mistakes, we also would like to have a proof that is obtained in a more structured and formal way. Thus a formal proof with theorem proving, probably with constraint solving will be investigated as a further work. For practical purposes, it may be most safe to bound the parameters and then find admissible configurations by exhaustive enumeration. All sets could then be used in a model checking.

It is clear that development of the fault-affected model and the fault hypotheses is crucial for getting the intended end-product. A small variation in the suggested procedure may be very helpful: each fault transition can be labelled with a fault name. Then the fault hypotheses can be encoded as a fault generating automaton that synchronizes with these. In the general case, it will just be a self loop for each fault label, offering these randomly. This automaton may also assist in checking for fault monotonicity.

For future work, there are still several other directions that we pursue. We have neither considered the voter switching strategy, nor a faulty voter. In the current system the arbiter trusts the output of VOTER$_0$ by default. So clearly, defining the switching mechanism is a direction in which the effort on modeling and analyzing the fault-tolerant properties can also be extended.

Also, up to now, we have verified the functionality requirements of the system. In our future work, we will consider how the dependability (performance) of the system may be investigated using a model checker and probabilistic simulation for real-time probabilistic systems.

References

1. Abadi, M., Lamport, L.: The existence of refinement mapping. Theoretical Computer Science 82(2), 253–284 (1991)
2. Alur, R., Dill, D.L.: A Theory of Timed Automata. Theoretical Computer Science 126(2), 183–235 (1994)
3. Avizienis, A., Laprie, J.-C., Randell, B.: Fundamental Concepts of Dependability. In: Proceedings of the 3rd IEEE Information Survivability Workshop (ISW 2000), pp. 7–12 (2000)

4. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
5. Bernardeschi, C., Fantechi, A., Gnesi, S.: Model checking fault tolerant systems. Journal of Software Testing, Verification and Reliability (STVR) 12(4), 251–275 (2002)
6. Clarke, E.M., Grumberg, O., Long, D.E.: Model Checking and Abstraction. ACM Transations on Programming Languadge and Ststems 16(5), 1512–1542 (1992)
7. Cousot, P., Cousot, R.: On abstraction in software verification. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 37–56. Springer, Heidelberg (2002)
8. Dams, D., Gerth, R., Grumberg, O.: Abstract interpretation of reactive systems. ACM Transactions on Programming Languages and Systems (TOPLAS) 19(2), 253–291 (1997)
9. Devillers, M.C.A., Griffioen, W.O.D., Romijn, J.M.T., Vaandrager, F.W.: Verification of a Leader Election Protocol - Formal Methods Applied to IEEE 1394. Formal Methods in System Design 16(3), 307–320 (2000)
10. Gebremichael, B., Vaandrager, F.W., Zhang, M.: Analysis of the Zeroconf Protocol Using UPPAAL. In: Proceedings of the 6th Annual ACM & IEEE Conference on Embedded Software (EMSOFT 2006), pp. 242–251. ACM Press, New York (2006)
11. Gnesi, S., Lenzini, G., Martinelli, F.: Logical specification and analysis of fault tolerant systems through partial model checking. In: Proceedings of the International Workshop on Software Verification and Validation (SVV 2003). Electronic Notes in Theoretical Computer Science, vol. 118, pp. 57–70 (2003)
12. Jensen, H.E.: Abstraction-Based Verification of Distributed Systems. Phd thesis, Department of Computer Science, Aalborg University, Denmark (June 1999)
13. Jensen, H.E., Larsen, K.G., Skou, A.: Scaling up uppaal. In: Joseph, M. (ed.) FTRTFT 2000. LNCS, vol. 1926, pp. 19–30. Springer, Heidelberg (2000)
14. Johnson, B.W.: Design and analysis of fault-tolerant digital systems. Addison-Wesley Publishing, Reading (1989)
15. Liu, Z., Joseph, M.: Specification and verification of fault-tolerance timing, and scheduling. ACM Transactions on Programming Languages and Systems 21(1), 46–89 (1999)
16. Liu, Z., Joseph, M.: Verification of fault-tolerance and real time. In: FTCS 1996, pp. 220–229. IEEE Computer Society Press, Los Alamitos (1996)
17. Loiseaux, C., Graf, S., Sifakis, J., Bouajjani, A., Bensalem, S.: Property Preserving Abstractions for the Verification of Concurrent Systems. Formal Methods in System Design 6(1), 11–44 (1995)
18. Schneider, F., Easterbrook, S.M., Callahan, J.R., Holzmann, G.J.: Validating requirements for fault tolerant systems using model checking. In: Proceedings of the 3rd International Conference on Requirements Engineering, pp. 4–13. IEEE Computer Society Press, Los Alamitos (1998)
19. Simons, D.P.L., Stoelinga, M.: Mechanical verification of the IEEE 1394- a root contention protocol using Uppaal2k. International Journal on Software Tools for Technlogy Transfer, 469–485 (2001)
20. Stoelinga, M.I.A., Vaandrager, F.W.: Root contention in IEEE 1394. In: Katoen, J.-P. (ed.) AMAST-ARTS 1999, ARTS 1999, and AMAST-WS 1999. LNCS, vol. 1601, pp. 53–74. Springer, Heidelberg (1999)
21. Yorav, K.: Exploiting syntactic structure for automatic verification. PhD thesis, The Technion, Israel Insitute of Technology (2000)

Dynamically Detecting Faults via Integrity Constraints

Ian J. Hayes

School of Information Technology and Electrical Engineering,
University of Queensland, Brisbane, Australia

Abstract. Control programs for safety-critical systems are required to tolerate faults in the devices they control. In this paper we examine a systematic approach to devising code to detect faulty devices at runtime. The approach is centred around the use of *integrity constraints*, which are invariants on the state of a system's variables, including its inputs and outputs. Under normal operation integrity constraints should always hold, but they are designed to fail to hold if there is a fault. By adding variables to capture the previous state of variables or the time of significant events, additional integrity constraints can be devised to check for faults in state transitions or faults with the rate of progress of the system. We discuss techniques for devising integrity constraints as well as efficiently evaluating the constraints. When an error is detected via the failure of an integrity constraint, the integrity constraint(s) that failed can help diagnose the likely fault. The techniques are presented by way of a simple case study of controller software written in the action system style, but the approach is equally applicable to other state machine approaches such as Event-B and TLA.

Keywords: Integrity constraint; action system; fault detection; real-time programming.

1 Introduction

Real-time computer systems are increasingly being employed to control safety-critical applications in which the safety of the system depends on the computer, and for which the consequences of failure can be severe. For example, real-time systems are used in railway signaling and in fly-by-wire aircraft. When equipment is operating correctly the behaviour of the equipment in response to control commands is predictable, but a device failure may lead to behaviours that are much less predictable and may affect the operation of other devices. Equipment can malfunction, connections can be broken, sensors and actuators can fail, human operators can make mistakes, the computer running the software can fail, and the software itself may be faulty. The challenge is to design systems and software that can detect and/or tolerate such faults. There are different levels of tolerance to faults: a system may continue to perform its desired function using alternative means; it may degrade and not perform all functions; or it may revert to a safe state until the fault is rectified.

Our work builds on standard techniques for developing safety-critical systems. In the early stages of system design it is conventional to perform a preliminary hazard analysis process, utilising standard techniques, such as hazard and operability analysis, to identify system hazards and assess them for severity and frequency of occurrence [1,2].

M. Butler et al. (Eds.): Fault Tolerance, LNCS 5454, pp. 85–103, 2009.
© Springer-Verlag Berlin Heidelberg 2009

For individual components of the system, including hardware, software, and human operators, the possible failure modes of each component are recorded.

In current fault-tolerant systems, checks to detect faults are commonly added to the code, but in an ad hoc manner that is explicitly programmed into the code (making it complicated). In this paper we focus on a systematic approach to fault detection and diagnosis that is centred on *integrity constraints*, which are conditions that hold in normal operation, but may fail to hold in the event of a fault. Our goals are to simplify the task of the system developer and improve the dependability of the generated system.

System modelling. There are a number of well known approaches to providing tolerance to faults, involving such techniques as consistency checking of the perceived behaviour of hardware with respect to a model of its expected behaviour, or replication of hardware and the use of comparison/voting strategies. The system level provides the best place to examine the application of fault-tolerant architectures and design patterns [3,4,5,6,7].

In an ideal world one can specify a real-time control system by specifying the *desired behaviour* of the equipment under the control of the system and by stating the *assumptions* a system developer can make about how equipment behaves in response to control commands [8]. In safety-critical systems one needs to distinguish between the desired behaviour of the system under normal operation and *safe operation properties* of the system that need to be maintained both under normal operation and when faults occur. There may be multiple safety properties with different levels of severity. Such systems can be specified using multiple levels of rely/guarantee conditions [9].

The emphasis of this paper is on dynamically detecting faulty behaviour of devices. Our goal is to come up with a systematic method to augment the control software for a system with code to detect and diagnose faults. Our presentation of the approach is based around the example of an industrial press, which is introduced in detail in Sect. 2. The remaining sections each give an overview of a stage in the process, followed by detailed application to the example. The approach consists of

- examining the possible device faults and hazards of the system (Sect. 3),
- devising integrity constraints for detecting the faults (Sect. 4),
- implementing the code to check the integrity constraints (Sect. 5), and
- examining methods for diagnosing possible faults when integrity constraints fail to hold (Sect. 6).

2 Industrial Press Example

We model a control program as a state machine using a notation based on action systems [10,11], which are in turn based on Dijkstra's guarded command language [12]. As an example, consider an industrial press in which a large weight is lifted by a *motor* and released to fall and press a sheet of metal into a mould to shape it [13]. It has boolean sensors *top*, *bottom* and *below_pnr*, to determine respectively whether the weight is at the top of its travel, the bottom of its travel, or that it is below the point of no return (abbreviated *pnr*), at which stage it can no longer be stopped safely from falling by the motor without damaging the equipment, with possible consequent operator injury. When the weight is at the top of its travel it can be *locked* at the top and will then remain at the top without requiring the motor to be on. If the weight is at the top of its travel, the operator may press two buttons simultaneously to release the weight. The operator must then hold both buttons while the weight is falling. If either button is released before the weight reaches the point-of-no-return sensor, the motor is engaged to stop the weight falling and lift it to the top position. Once the weight has reached the bottom of its travel, it will be lifted back to the top once the operator releases both buttons. The controller normally cycles through the sequence of modes *Top*, *Falling*, *Below_PNR*, *Bottom*, *Lifting*, and back to *Top*, but if a button is released while the controller is in mode *Falling*, it will switch into mode *Abort*. Mode *Abort* behaves in the same way as mode *Lifting* in the normal case but we want to distinguish it from mode *Lifting* because fault detection in the two modes is different. The distinction between these modes was added as part of the process of extending the system with fault detection, but we have included it from the start to avoid having to repeat the whole control program.

The following summarises the inputs (from sensors), outputs (to actuators), and local variables used by the software. We start variable names with a lowercase letter, and names of constants (including type names) with an uppercase letter. We distinguish between an input sensor, e.g., *top_sensor*, and the current sample of the sensor stored in a local variable, *top*. The command *top* : **read**(*top_sensor*) samples the *top_sensor* and places the result in the variable *top*. We explain the auxiliary variable, *action_deadline*, and the use of deadlines later.

```
input top_sensor, bottom_sensor, below_pnr_sensor,
      $pressed_1$_sensor, $pressed_2$_sensor : Boolean;
var top, bottom, below_pnr, $pressed_1$, $pressed_2$ : Boolean;
output motor : On | Off;
output locked : Boolean;
output alarm : Boolean;
var start_fall, start_lift : Time;
var mode : Top | Falling | Below_PNR | Bottom | Lifting |
      Abort | Below_PNR_Init;
aux action_deadline : Time
```

An action system to control a (fault-free) press is given in Fig. 1 and the corresponding initialisation in Fig. 2. After initialisation, the action system is repeatedly executed via the "**do** *true*" loop. At the beginning of each iteration it samples all the sensors and the current time, and then performs one of a set of possible actions. Each action has a guard, that determines whether or not the action is enabled. On each repetition an enabled action is selected and executed. If none of the guards is enabled the final "**otherwise**" alternative is executed, which executes the **skip** (or no-operation) command; this represents delaying until one of the guards becomes enabled. In general, more than one action may be enabled, in which case the selection of which action to execute is nondeterministic (although our example controller is deterministic). Most actions consist of either a simple assignment or a concurrent assignment.

To avoid inconsistencies in the guard evaluation, any input referenced in a guard is sampled just once at the start of the loop and that value is used throughout the evaluation of all the guards and in the action executed on that iteration. For example, in

The boxed sections are used in the checking of integrity constraints; ignore them on first reading.

```
$action\_deadline := \tau + Max\_iteration\_time$;
do $true \rightarrow$
    -- Sample sensors once at the start of each iteration.
    $pressed_1$ : read($pressed_1\_sensor$);
    $pressed_2$ : read($pressed_2\_sensor$);
    $top$ : read($top\_sensor$);
    $below\_pnr$ : read($below\_pnr\_sensor$);
    $bottom$ : read($bottom\_sensor$);
    $\boxed{now : \mathbf{gettime}}$;
    -- Perform a guarded action or skip.
    if $mode = Top \land pressed_1 \land pressed_2 \rightarrow$
        $(locked := False \parallel mode := Falling)$; $\boxed{start\_fall : \mathbf{gettime}}$
    [] $mode = Falling \land \neg\, below\_pnr \land \neg\, (pressed_1 \land pressed_2) \rightarrow$
        $(motor := On \parallel mode := Abort)$; $\boxed{start\_lift : \mathbf{gettime}}$
    [] $mode = Falling \land below\_pnr \rightarrow$
        $mode := Below\_PNR$
    [] $mode = Below\_PNR \land bottom \rightarrow$
        $mode := Bottom$
    [] $mode = Below\_PNR\_Init \land bottom \rightarrow$
        $(mode := Bottom \parallel alarm := False)$
    [] $mode = Bottom \land \neg\, pressed_1 \land \neg\, pressed_2 \rightarrow$
        $(motor := On \parallel mode := Lifting)$; $\boxed{start\_lift : \mathbf{gettime}}$
    [] $mode = Lifting \land top \rightarrow$
        $(motor := Off \parallel locked := True \parallel mode := Top)$
    [] $mode = Abort \land top \rightarrow$
        $(motor := Off \parallel locked := True \parallel mode := Top)$
    otherwise
        skip
    fi;
    $\boxed{\text{check integrity constraints}}$
    deadline $action\_deadline$;
    $action\_deadline := \tau + Max\_iteration\_time$;
    suspend
od
```

Fig. 1. Action system controller for an industrial press

mode *Falling* in Fig. 1 the two alternative guards both reference the input *below_pnr*. If *below_pnr* were to be sampled twice, the value for the first alternative could be false and that for the second true (or vice versa, depending on the order of evaluation of the guards) leading to both guards being false (or both potentially true).

The action system may be repeated non-stop in a busy-waiting fashion if it is the only program running on the computer, or, more likely, it may be one of a number of tasks on a computer and it is periodically scheduled for execution with a short enough period between executions to give the desired response time for the system. In a multi-tasking environment, to indicate that the action system has finished its scheduled iteration, it calls the system command **suspend**.

The auxiliary timing variable, *action_deadline*, has been added in order to express the fact that iterations should have a maximum separation in time. It is initialised to the maximum iteration time plus the current time, which is represented by the special variable τ, and there is a deadline command using its value at the end of the loop. The deadline command guarantees to terminate by the specified time [14]. The deadline command is a specification construct. No code is generated for it (and hence it takes no time to execute). To correctly implement a specification containing deadlines, a program must guarantee that whenever its execution reaches a deadline command, the deadline has not expired. The use of a deadline command allows one to abstract away from details of real-time scheduling, while still specifying the desired real-time behaviour. Reinitialisation of the action deadline occurs immediately after the deadline to ensure the time limit for the next iteration starts immediately. No code is generated for the use of the auxiliary variable or the deadline command. They are there purely so that the timing constraint can be specified within the code. More details on the use of deadlines for timing constraints can be found elsewhere [15].

```
-- Initialisation
$top : \mathbf{read}(top\_sensor);$
$below\_pnr : \mathbf{read}(below\_pnr\_sensor);$
$bottom : \mathbf{read}(bottom\_sensor);$
$alarm := False;$
$\mathbf{if}\ top \rightarrow$
    $(motor := Off \ || \ locked := True \ || \ mode := Top)$
$[]\ bottom \rightarrow$
    $(motor := Off \ || \ locked := False \ || \ mode := Bottom)$
$[]\ below\_pnr \wedge \neg\ bottom \rightarrow$
    $(motor := Off \ || \ locked := False \ || \ mode := Below\_PNR\_Init \ || \ alarm := True);$
    $\boxed{start\_fall : \mathbf{gettime}};\ start\_fall := start\_fall - Min\_time\_below\_pnr$
$[]\ \neg\ top \wedge \neg\ below\_pnr \rightarrow$
    $(motor := On \ || \ locked := False \ || \ mode := Abort);\ \boxed{start\_lift : \mathbf{gettime}}$
$\mathbf{fi}$
```

Fig. 2. Initialisation of controller for an industrial press

For initialisation of the control system we do not assume that the plant is in some predetermined state, because a power failure may lead to the system being initialised at any point in its cycle. The initialisation code (Fig. 2) checks the current values of the sensors and initialises the system to an appropriate mode. If the weight is falling but above the point of no return, the controller will abort the fall and lift the weight

Table 1. Device faults and corresponding hazards for the industrial press

Fault	Description	Hazards
TOn	*top* holds when the weight is not at the top of its travel	H1
TOff	$\neg$ *top* holds when the weight is at the top of its travel	H3
POn	*below_pnr* holds when the weight is above the point of no return	H1
POff	$\neg$ *below_pnr* holds when the weight is below the point of no return	H2
BOn	*bottom* holds when the weight is not at the bottom of its travel	H2
BOff	$\neg$ *bottom* holds when the weight is at the bottom of its travel	—
LOn	*locked* holds and the weight is not locked at the top	H1
LOff	$\neg$ *locked* holds and the weight is locked at the top	—
MOn	$motor = On$ and the motor is not actually running/lifting	H1, H2
MOff	$motor = Off$ and the motor is actually running/lifting	H1, H2, H3
$ButOn_i$	$pressed_i$ holds but no one actually pressed the button	H1
$ButOff_i$	$\neg$ $pressed_i$ holds but it is actually being pressed	—

The name of each fault is the perceived value of the state according to the computer, but this state does not correspond to reality. The fault may lead to the listed hazards.

to the top regardless of the state of the buttons. A mode *Below_PNR_Init* (which is different to *Below_PNR*) has been used to handle the hazardous case when initialisation takes place while the weight is below the point of no return. The weight is allowed to fall but an alarm is sounded. Unfortunately on initialisation the control software can't determine the direction or velocity of the weight, and hence it has to assume that the weight is falling with a velocity greater than that at which it can be safely stopped by the motor. Aside: Adding sensors for the direction and/or velocity of the weight would allow one to initialise the system more safely as well as improve the fault detection process discussed below.

3 Possible Faults and Hazards

Given a control system and the plant it is controlling, the next stage is to identify possible device faults, and the hazards that they can lead to. Because a control system can only perceive the plant through its sensors, it cannot distinguish a faulty sensor from a faulty device that is being sensed. Similarly, when the control system sets an actuator, it has knowledge of what state the actuator was set to, but it does not have direct knowledge of whether the signal got to the device or whether the device responds correctly to the signal (although it may indirect knowledge of the device's response via its sensors). Hence in identifying possible faults we focus on the interface between the control system and the plant it is controlling, i.e., we focus on the sensors and actuators.

For each input sensor, S, and each possible state of that sensor, M, we identify a fault, SM, which corresponds to the sensor S reading that it is in state M to the control system, but the device itself is not in the physical state corresponding to M. Similarly, for each output actuator, A, and possible state of the actuator, M, we identify a fault, AM, which corresponds to the control system having set the actuator to state M, but the controlled device does not behave as it should in state M. We follow this naming convention because it applies when an input or output has a type which has a (small) finite number of values.

Some possible device faults for the industrial press are summarised in Table 1. For example, *MOn* corresponds to the control system having set *motor* = *On*, but that this does not reflect the state in the real world, i.e., the motor is not actually running or it is running but not lifting the weight.

Hazards. Given a collection of possible faults, we also identify the possible hazards that each fault can lead to. This helps one understand the severity of a fault, as well as better understand what will be suitable recovery strategies.

A hazard analysis of the industrial press can be performed, which may come up with the following possibilities:

H1 an operator may be injured if the weight is allowed to fall while the operator is not pressing both buttons and the weight is above the point of no return,
H2 there may be a catastrophic equipment failure, possibly leading to operator injury, if the motor is turned on while the weight is falling and it is below the point of no return, and
H3 there may be a motor failure if the motor is left on while the weight is at the top of its travel, which could lead to the weight falling.

The conditions that may lead to hazard H1 are that

1. the weight is at the top of its travel, in which case it should be locked at the top but it isn't (*LOn*),
2. a button is not pressed but the button sensor reads that it is pressed, which can lead to the weight to be released when it shouldn't be ($ButOn_i$),
3. the weight is above the point of no return but the *below_pnr* sensor is indicating it is below (*POn* early),
4. the motor is turned off but actually runs for a while and lifts the weight from the bottom but then stops running and allows the weight to fall (*MOff*),
5. the weight is not at the top of its travel but the top sensor indicates it is (*TOn* early), and hence the motor is turned off but the locking of the weight will fail because it is not at the top (*LOn*), or
6. the motor fails in the lifting phase and the weight falls (*MOn*).

The conditions that may lead to hazard H2 are that

1. the *below_pnr* sensor is indicating that the weight is above the point of no return but it is really below (*POff*),
2. the weight has fallen below the point of no return but the motor starts running while it is turned off (*MOff*),
3. the bottom sensor indicates that the weight is at the bottom, but it is still falling (*BOn* early), which can lead to the motor be turned on to lift the weight, or
4. the motor is lifting the weight but fails and the weight falls and then the motor starts lifting again when the weight is travelling too fast (*MOn*).

The conditions that may lead to hazard H3 are that

1. the motor is actually running when it is turned off (*MOff*), or
2. the top sensor indicates that the weight is not at the top of its travel, when it is (*TOff*).

We note that a power failure can lead to the motor not running when it is supposed to be turned on (*MOn*). The locking mechanism for the weight should be designed to be fail safe in the event of a power failure.

4 Detecting Faults via Integrity Constraints

A device failure may lead to a system state that cannot be reached when the device is operating correctly. The correct operation of a device, or more generally, a collection of devices working together, can be characterised by a set of *integrity constraints* on the state of the computer system's variables, including its inputs and outputs. If at any stage an integrity constraint does not hold, this indicates an error in the system. In order to devise integrity constraints for a system we need to characterise the healthy behaviour of the devices.

The advantage of using integrity constraints is that it provides a more systematic approach to detecting faults. Rather than developing ad hoc code to detect faults, one can devise a set of integrity constraints, which are checked after every iteration of the control loop. We can make use of integrity constraints to detect:

- some combination of sensor values that is impossible if all devices and sensors are operating correctly,
- that a state invariant of the system has been violated,
- that the sensors are indicating values that should not be possible in the current program state,
- a state transition occured which should not be possible if the system and devices are functioning correctly,
- that an expected change of state of the system has not occured by the time at which it was expected,
- that a system state change occurs before it was expected,
- that a common device fault (e.g., a sensor is stuck in one state) has occurred, and
- whether the behaviour of a device is healthy.

In the remainder of this section we discuss approaches to devising integrity constraints, and give examples based on the industrial press controller.

Sensors indicating physically impossible states. Some combinations of sensor values are impossible if the sensors and the devices they are sensing are operating correctly. For the industrial press example, in normal operation the *top* sensor should never be on when either the *below_pnr* or *bottom* sensors are on, and whenever the *bottom* sensor is on the *below_pnr* sensor should also be on. From these we get integrity constraints (1), (2), and (3) in Fig. 3, which summarises the set of integrity constraints that we use with the industrial press.

Note that in order to check these integrity constraints the values of the sensors need to be monitored in states in which they would normally be ignored. The *top* sensor coming on while the weight is falling does not necessarily mean that the system is unsafe. However, it is likely to lead to problems when the weight is being lifted again that are best avoided by not attempting to lift the weight. These integrity constraints can

Mode independent constraints.

$$\neg\,(\mathit{top} \wedge \mathit{bottom}) \tag{1}$$
$$\neg\,(\mathit{top} \wedge \mathit{below\_pnr}) \tag{2}$$
$$(\mathit{bottom} \Rightarrow \mathit{below\_pnr}) \tag{3}$$

Mode dependent constraints (listed by mode).

$$\mathit{mode} = \mathit{Top} \Rightarrow \mathit{top} \tag{4}$$
$$\mathit{mode} = \mathit{Falling} \Rightarrow \mathit{now} \leq \mathit{start\_fall} + \mathit{Max\_time\_pnr} \tag{5}$$
$$\mathit{mode} = \mathit{Falling} \wedge \mathit{top} \Rightarrow \mathit{now} \leq \mathit{start\_fall} + \mathit{Max\_time\_not\_top} \tag{6}$$
$$\mathit{mode} = \mathit{Below\_PNR} \Rightarrow \mathit{below\_pnr} \tag{7}$$
$$\mathit{mode} = \mathit{Below\_PNR} \Rightarrow \mathit{now} \leq \mathit{start\_fall} + \mathit{Max\_time\_fall} \tag{8}$$
$$\mathit{mode} = \mathit{Below\_PNR\_Init} \Rightarrow \mathit{now} \leq \mathit{start\_fall} + \mathit{Max\_time\_fall} \tag{9}$$
$$\mathit{mode} = \mathit{Lifting} \Rightarrow \mathit{now} \leq \mathit{start\_lift} + \mathit{Max\_time\_lift} \tag{10}$$
$$\mathit{mode} = \mathit{Lifting} \wedge \mathit{bottom} \Rightarrow \mathit{now} \leq \mathit{start\_lift} + \mathit{Max\_time\_not\_bottom} \tag{11}$$
$$\mathit{mode} = \mathit{Lifting} \wedge \mathit{below\_pnr} \Rightarrow \mathit{now} \leq \mathit{start\_lift} + \mathit{Max\_time\_not\_below\_pnr} \tag{12}$$
$$\mathit{mode} = \mathit{Lifting} \wedge \neg\, \mathit{below\_pnr} \Rightarrow \mathit{now} \geq \mathit{start\_lift} + \mathit{Min\_time\_lift\_to\_pnr} \tag{13}$$
$$\mathit{mode} = \mathit{Lifting} \wedge \mathit{below\_pnr} \Rightarrow \mathit{below\_pnr}_{\mathit{prev}} \tag{14}$$
$$\mathit{mode} = \mathit{Bottom} \Rightarrow \mathit{bottom} \tag{15}$$
$$\mathit{mode} = \mathit{Abort} \Rightarrow \mathit{now} \leq \mathit{start\_lift} + \mathit{Max\_time\_lift\_from\_pnr} \tag{16}$$

Program invariants.

$$(\mathit{mode} = \mathit{Lifting} \vee \mathit{mode} = \mathit{Abort}) \Leftrightarrow \mathit{motor} = \mathit{On} \tag{17}$$
$$\mathit{mode} = \mathit{Top} \Leftrightarrow \mathit{locked} \tag{18}$$
$$\mathit{mode} = \mathit{Falling} \Rightarrow \mathit{pressed}_1 \wedge \mathit{pressed}_2 \tag{19}$$

Minimum time to transition constraints.

$$\mathit{mode} = \mathit{Lifting}_{\mathit{prev}} \wedge \mathit{mode} = \mathit{Top} \Rightarrow \mathit{now} \geq \mathit{start\_lift} + \mathit{Min\_time\_top} \tag{20}$$
$$\mathit{mode} = \mathit{Falling}_{\mathit{prev}} \wedge \mathit{mode} = \mathit{Below\_PNR} \Rightarrow \mathit{now} \geq \mathit{start\_fall} + \mathit{Min\_time\_below\_pnr} \tag{21}$$
$$\mathit{mode} = \mathit{Below\_PNR}_{\mathit{prev}} \wedge \mathit{mode} = \mathit{Bottom} \Rightarrow \mathit{now} \geq \mathit{start\_fall} + \mathit{Min\_time\_bottom} \tag{22}$$

Fig. 3. Integrity constraints for the industrial press

be dynamically checked at the end of every iteration. This code can appear at the end of the loop in the box labelled “check integrity constraints” in Fig. 1. We give the details of the code in Sect. 5.

Program invariants. If the control system for the industrial press is in the mode *Top*, indicating that the weight is at the top of its travel, then *locked* should be true, and furthermore this is the only mode in which *locked* should be true. This leads to integrity constraint (18) in Fig. 3, i.e.,

$$\mathit{mode} = \mathit{Top} \Leftrightarrow \mathit{locked}$$

In this case both *mode* and *locked* are under the control of the program and this invariant is established by the program initialisation and maintained by every iteration of action system. Checking such an invariant is only checking correct operation of the program and not the devices it is controlling.

In Fig. 3, integrity constraints (17) and (18) are of this form. Integrity constraint (19) is also an invariant of the program, but in this case the buttons are inputs; (19) is maintained as an invariant because the program will switch out of mode *Falling* as soon as a button is released.

Consistency between program states and sensors. For some program states only certain sensor values are valid. For our example, if the controller is in mode *Top* then the *top* sensor should be on; if it is in mode *Below_PNR* then the *below_pnr* sensor should be on; and if it is in mode *Bottom* then the *bottom* sensor should be on. These give us integrity constraints (4), (7), and (15) in Fig. 3.

Note that these integrity constraints are mode dependent, for example, (4) is written in the form

$$\mathit{mode} = \mathit{Top} \Rightarrow \ldots$$

This only needs to be checked in mode *Top*. We'll see further examples of mode-dependent integrity constraints below.

For each mode we can determine all the constraints we expect to hold on sensor values. As well as the constraints (4), (7), and (15) we get the constraints listed in Fig. 4. However, checking these constraints would be redundant: they are implied by the existing integrity constraints and/or program invariants, and hence we have not included them in Fig. 3. The third case in Fig. 4 is implied by a program invariant because in mode *Falling* the program will switch out of that mode when it senses *below_pnr* is true.

State transition integrity constraints. As well as checking the validity of the current state of the system, integrity constraints may be used to check the validity of transitions

$\mathit{mode} = \mathit{Top} \Rightarrow \neg\, \mathit{below\_pnr}$	implied by	(4) and (2)
$\mathit{mode} = \mathit{Top} \Rightarrow \neg\, \mathit{bottom}$	implied by	(4) and (1)
$\mathit{mode} = \mathit{Falling} \Rightarrow \neg\, \mathit{below\_pnr}$	implied by	program invariant
$\mathit{mode} = \mathit{Falling} \Rightarrow \neg\, \mathit{bottom}$	implied by	program invariant and (3)
$\mathit{mode} = \mathit{Below\_PNR} \Rightarrow \neg\, \mathit{top}$	implied by	(7) and (2)
$\mathit{mode} = \mathit{Below\_PNR} \Rightarrow \neg\, \mathit{bottom}$	implied by	program invariant
$\mathit{mode} = \mathit{Bottom} \Rightarrow \neg\, \mathit{top}$	implied by	(15) and (1)
$\mathit{mode} = \mathit{Bottom} \Rightarrow \mathit{below\_pnr}$	implied by	(15) and (3)
$\mathit{mode} = \mathit{Lifting} \Rightarrow \neg\, \mathit{top}$	implied by	program invariant

Fig. 4. Redundant integrity constraints

by adding variables to the state that record previous system states. Some forms of device failure cannot be detected by an integrity constraint on a single state. A failure may lead to a transition between two states in which both the previous state and current state are possible valid states, but the transition between them can only take place if a device is faulty.

To detect such faults, additional variables representing the value of a variable on the previous iteration, can be added to the system; these variables are indicated by a subscript of *prev*. Valid transitions can then be encoded in an integrity constraint. For example, for the industrial press, if the system is in mode *Lifting* and is above the point of no return, the *below_pnr* sensor cannot become true again. This can be captured by integrity constraint (14) in Fig. 3, i.e.,

$$mode = Lifting \wedge below\_pnr \Rightarrow below\_pnr_{prev}$$

Note that although state *Abort* is similar to *Lifting*, we can't apply a similar integrity constraint in mode *Abort* because the transition to mode *Abort* may occur when the weight is just above the point of no return, and even with the motor on, the weight may fall below the point of no return while it is decelerating.

The additional previous-state variables used for the integrity constraint checking need to be added to the code and code needs to be generated to keep them up to date. Additional "earlier" values of variables may also be useful for specifying integrity constraints; these can be explicitly added to the code and then used in integrity constraints.

As another example of an integrity constraint that uses previous state variables, if the *top* sensor was true on the previous iteration, then the *bottom* sensor should not be true for the current iteration because the time for the weight to fall is much larger than the maximum time allowed between iterations. This gives the following integrity constraint:

$$\neg\ (top_{prev} \wedge bottom) \tag{23}$$

While integrity constraint (23) is feasible, a more thorough check on the operation of the press can be afforded if we allow integrity constraints that check the rate-of-change of variables; we address these next.

Invalid system dynamics. Some forms of correctness properties relate to the valid rates of change of variables and temporal validity of data; i.e., that data is not too old. These can be checked by integrity constraints if we add "time stamp" variables that record the times of significant events.

To further describe the behaviour of the press, we can take into account its dynamics. If the motor is off then the weight falls under gravity. From this we can deduce that there is some maximum time, *Max_time_pnr*, that the weight takes to fall to the point of no return. Due to the complexities of friction, etc., such constants are likely to be determined empirically, and they should allow for timing errors due to sampling rates. If the weight takes longer than *Max_time_pnr* to actually fall, there is something wrong. By checking whether the weight reaches the *below_pnr* sensor within this time, such a fault can be detected.

If we record the time, *start_fall*, at which the weight starts to fall, then the *below_pnr* sensor should become true within *Max_time_pnr*. This explains the purpose of the boxed call to **gettime** in the transition from mode *Top* in Fig. 1. A suitable integrity constraint is (5) in Fig. 3, i.e.,

$$mode = Falling \Rightarrow now \leq start\_fall + Max\_time\_pnr,$$

in which the variable *now* represents the current time, which was sampled at the start of the iteration.

The more frequently this condition is checked while the system is in mode *Falling*, the more likely it is that this fault will be detected. Note that most of the checks will be performed after the "**otherwise**" case in the action system in Fig. 1. Similar checks can be devised for reaching the *bottom* sensor in modes *Below_PNR* and *Below_PNR_Init*, or the *top* sensor in modes *Lifting* and *Abort*, giving us integrity constraints (8), (9), (10), and (16) in Fig. 3.

Minimum time to transitions. The above considered the maximum time allowed to reach a mode transition, but in many cases the physical timing of device operation implies that there is a minimum time to a transition. For example, it should take a time of at least *Min_time_below_pnr* for the weight to reach the point of no return. This is captured in integrity constraint (21) in Fig. 3, i.e.,

$$mode = Falling_{prev} \wedge mode = Below\_PNR \Rightarrow now \geq start\_fall + Min\_time\_below\_pnr$$

Because time can only increase, this integrity constraint only needs to be checked once on the transition from mode *Falling* to *Below_PNR*. Additional minimum time constraint integrity constraints can be devised for the time to fall to the bottom and the time to lift to the top, i.e., integrity constraints (22) and (20) in Fig. 3. Again these integrity constraints only need to be checked on the transitions from *Below_PNR* to *Bottom* and from *Lifting* to *Top*, respectively.

Common device faults. The above gives an indication of how integrity constraints can be devised by examining the expected behaviour of devices. A complementary approach is to consider the common faults that devices can exhibit. For example, common sensor faults are for a sensor to become stuck in one state. One can then devise integrity constraints that will detect such faults. For example, one can check for the *top* sensor being stuck in the true state: when the weight is released the *top* sensor should become false within some short period of time, *Max_time_not_top*, representing the maximum time for the *top* sensor to become false if the system is operating correctly. This gives integrity constraint (6) in Fig. 3, i.e.,

$$mode = Falling \wedge top \Rightarrow now \leq start\_fall + Max\_time\_not\_top.$$

By the time the weight reaches the *below_pnr* sensor, the *top* sensor should have been false for quite some time and hence the integrity constraint of $\neg\ (top \wedge below\_pnr)$ will also eventually detect this fault. However, failure of integrity constraint (6) indicates a problem with the top sensor or that the weight has not been released, whereas $\neg\ (top \wedge below\_pnr)$ could fail because either the *top* sensor or the *below_pnr* sensor is faulty. Together they give a finer determination of the likely fault. Similar checks can be applied to the other sensors to give integrity constraints (11) and (12) in Fig. 3. A further integrity constraint (13) can be devised for the *below_pnr* sensor being stuck off in mode *Lifting* because there is a minimum time by which the weight should be lifted above the point of no return.

Monitoring device healthiness. If a device does not satisfy its specification, it may or may not satisfy the assumptions made of it by the system. If it does satisfy the assumptions, the device is still *unhealthy* and this should be monitored and reported [16]. A failure to meet its specification may indicate that a device is “worn” and needs replacement, even though the overall system may still be behaving as desired. For example, the time taken for an operation by a worn device may be too long to meet the device’s specification, but may still be short enough to meet the assumptions made by the system about the device. If the worn device is left in the system, it may eventually lead to a system failure.

Undetectable faults. If device failures cannot be (easily) detected in the proposed system design, this may indicate that the design needs to be extended to allow fault detection. To better detect faulty inputs from sensors, they may be replicated, and to avoid common mode failures we may use such techniques as using different types of sensors, locating the sensors at different physical locations, using different wiring paths, and using different polarities of signals so that they have different open circuit and noise spike behaviour, including differential pairs which may indicate that there is no signal. For example, the failure of a button in the industrial press example is not detectable by the control system, which is why we have duplicated the button.

For actuators we can also use replication as well as ensuring that we have sensors that feed back information from which one can determine whether the actuator is behaving as expected.

Extending the design is especially important for faults that have severe consequences if they go undetected. Similarly, if a detected fault has multiple possible causes, the design may need extension to allow the causes to be differentiated, e.g., multiple sensors may aid in distinguishing the likelihood of sensor failure from device failure. Being able to differentiate faults is especially important if the recovery actions required for the different faults are different.

5 Checking Integrity Constraints

Although all the integrity constraints could be treated as a single (large) predicate to be checked after every transition, it is more effective to decompose it into multiple integrity constraints, $I_1, I_2, \ldots I_n$, each of which is to be checked. This allows better fault diagnosis because for a given integrity constraint one can identify a set of faulty behaviours of devices that could give rise to it becoming false. We can generate straightforward code to check integrity constraints. Mode-independent integrity constraint are evaluated directly on every iteration of the control loop, but mode-dependent integrity constraints are only evaluated if the program is in that mode. Integrity constraints that are program invariants will only be violated if there is a fault in the computer’s hardware or software.

```
$I_1 := \neg\ (top \wedge bottom)$;
$I_2 := \neg\ (top \wedge below\_pnr)$;
$I_3 := \neg\ bottom \vee below\_pnr$;
if $mode = Top \rightarrow$
        $I_4 := top$
[] $mode = Falling \rightarrow$
        $I_5 := (now \leq start\_fall + Max\_time\_pnr)$;
        $I_6 := \neg\ top \vee now \leq start\_fall + Max\_time\_not\_top$
[] $mode = Below\_PNR \rightarrow$
        $I_7 := below\_pnr$;
        $I_8 := (now \leq start\_fall + Max\_time\_fall)$
[] $mode = Below\_PNR\_Init \rightarrow$
        $I_9 := (now \leq start\_fall + Max\_time\_fall)$
[] $mode = Lifting \rightarrow$
        $I_{10} := (now \leq start\_lift + Max\_time\_lift)$;
        $I_{11} := \neg\ bottom \vee now \leq start\_lift + Max\_time\_not\_bottom$;
        $I_{12} := \neg\ below\_pnr \vee now \leq start\_lift + Max\_time\_not\_below\_pnr$;
        $I_{13} := below\_pnr \vee now \geq start\_lift + Min\_imte\_lift\_to\_pnr$;
        $I_{14} := \neg\ below\_pnr \vee below\_pnr_{prev}$
[] $mode = Bottom \rightarrow$
        $I_{15} := bottom$
[] $mode = Abort \rightarrow$
        $I_{16} := (now \leq start\_lift + Max\_time\_lift\_from\_pnr)$
fi
```

Fig. 5. Code to check integrity constraints

Integrity constraints requiring a transition to occur after some minimum time only need to be checked when the transition occurs, not on every iteration of the control loop.

The code to check the integrity constraints for the industrial press in given in Fig. 5. We check all the integrity constraints in Fig. 3, except that we omit (17), (18) and (19), because they are are program invariants, and hence do not correspond to device failures. We use boolean variables I_1 through I_{22} to represent whether each integrity constraint holds. These variables are assumed to all be initially true so that mode dependent integrity constraints only need to be evaluated in the corresponding mode.

Integrity constraints (20), (21), and (22) only need to be evaluated on transitions from mode *Lifting* to *Top*, from mode *Falling* to *Below_PNR*, and from mode *Below_PNR* to *Bottom*, respectively. Hence the code to evaluate these integrity constraints is added directly into the corresponding transitions. The augmented transitions (only) are given in Fig. 6.

Sampling issues. At this stage we should point out a subtlety with checking integrity constraint (1) due to the fact that it references more than one input from the environment. Technically, the integrity constraint requires that *top* and *bottom* do not hold at the same time, but in order to check this one needs to sample both sensors at *exactly* the same time; this is not physically possible, because it is impossible to guarantee that the

```
⋮
[] $mode = Falling \wedge below\_pnr \rightarrow$
      $mode := Below\_PNR;$
      $I_{21} := now \geq start\_fall + Min\_time\_below\_pnr$
[] $mode = Below\_PNR \wedge bottom \rightarrow$
      $mode := Bottom;$
      $I_{22} := now \geq start\_fall + Min\_time\_bottom$
⋮
[] $mode = Lifting \wedge top \rightarrow$
      $(motor := Off \,||\, locked := True \,||\, mode := Top);$
      $I_{20} := now \geq start\_lift + Min\_time\_top$
⋮
```

Fig. 6. Augmented transition to check integrity constraints

samples are taken at *exactly* the same time.[1] For the checking of this integrity constraint to be valid we require that the normal behaviour of the system devices is such that

- only one of the sensors is ever true at any one time, and
- there is a minimum time gap, g, between the times either sensor is true, during which both sensors are false.

Provided that on each iteration both sensors are sampled in a time less than g, the checking of the integrity constraint will be valid. Note that in the code in Fig. 1 all the sensors are sampled together, making it easier to ensure that they are sampled within the minimum time gap, g. Caspi and Salem address this issue in more detail [17].

6 Fault Diagnosis

Each integrity constraint, I_j, has a set of possible causes associated with its failure. To diagnose an error we can make use of

- which integrity constraints failed;
- the current values of state variables; and
- the values of input variables as used in the guard evaluation.

Further diagnostic information can be incorporated into the program state by additional variables that keep track of such things as the previous state or a longer history of the states after transitions (in a software "black-box recorder" buffer).

There may be multiple faults that can cause an integrity constraint error. For example, failure of the integrity constraint (2), i.e., $\neg\ (top \wedge below\_pnr)$, may be caused

[1] Even if the signals are checked at the hardware level there can be different (admittedly quite small) circuit delays for the two signals.

Table 2. Likely faults corresponding to an integrity constraint failure

	Mode	Negation of integrity constraint	Likely faults
1		$\mathit{top} \wedge \mathit{bottom}$	$\mathit{TOn}, \mathit{BOn}$
2		$\mathit{top} \wedge \mathit{below\_pnr}$	$\mathit{TOn}, \mathit{POn}$
3		$\mathit{bottom} \wedge \neg\, \mathit{below\_pnr}$	$\mathit{BOn}, \mathit{POff}$
4	*Top*	$\neg\, \mathit{top}$	$\mathit{TOff}, \mathit{LOn}$
5	*Falling*	$\mathit{now} > \mathit{start\_fall} + \mathit{Max\_time\_pnr}$	$\mathit{POff}, \mathit{LOff}, \mathit{MOff}$
6	*Falling*	$\mathit{top} \wedge (\mathit{now} > \mathit{start\_fall} + \mathit{Max\_time\_not\_top})$	$\mathit{TOn}, \mathit{LOff}, \mathit{MOff}$
7	*Below_PNR*	$\neg\, \mathit{below\_pnr}$	$\mathit{POff}, \mathit{MOff}$
8	*Below_PNR*	$\mathit{now} > \mathit{start\_fall} + \mathit{Max\_time\_fall}$	$\mathit{BOff}, \mathit{MOff}$
9	*Below_PNR_Init*	$\mathit{now} > \mathit{start\_fall} + \mathit{Max\_time\_fall}$	$\mathit{BOff}, \mathit{MOff}$
10	*Lifting*	$\mathit{now} > \mathit{start\_lift} + \mathit{Max\_time\_lift}$	$\mathit{TOff}, \mathit{MOn}$
11	*Lifting*	$\mathit{bottom} \wedge$ $(\mathit{now} > \mathit{start\_lift} + \mathit{Max\_time\_not\_bottom})$	$\mathit{BOn}, \mathit{MOn}$
12	*Lifting*	$\mathit{below\_pnr} \wedge$ $(\mathit{now} > \mathit{start\_lift} + \mathit{Max\_time\_not\_below\_pnr})$	$\mathit{POn}, \mathit{MOn}$
13	*Lifting*	$\neg\, \mathit{below\_pnr} \wedge$ $(\mathit{now} < \mathit{start\_lift} + \mathit{Max\_lift\_to\_pnr})$	POff
14	*Lifting*	$\mathit{below\_pnr} \wedge \neg\, \mathit{below\_pnr}_{\mathit{prev}}$	$\mathit{POff}, \mathit{POn}, \mathit{MOn}$
15	*Bottom*	$\neg\, \mathit{bottom}$	$\mathit{BOff}, \mathit{MOff}$
16	*Abort*	$\mathit{now} > \mathit{start\_lift} + \mathit{Max\_time\_lift\_from\_pnr}$	$\mathit{TOff}, \mathit{MOn}$
20	*Top*	$\mathit{now} < \mathit{start\_lift} + \mathit{Min\_time\_top}$	TOn
21	*Below_PNR*	$\mathit{now} < \mathit{start\_fall} + \mathit{Min\_time\_pnr}$	POn
22	*Bottom*	$\mathit{now} < \mathit{start\_fall} + \mathit{Min\_time\_bottom}$	BOn

by: a faulty *top* sensor; a faulty *below_pnr* sensor; or even faulty computer hardware or software. Where possible, more detailed integrity constraints can be used to differentiate which fault is likely to have caused the particular failure. When causes cannot be distinguished, which is highly likely for the example above, then the recovery action has to cover all possibilities: those that have more severe consequences take priority, but all safety-related consequences should be handled.

Table 2 gives a summary of the likely faults associated with the failure of each integrity constraint for the industrial press example. Rather than give the integrity constraints used in Fig. 3, in Table 2 we give their negations, i.e., the condition that holds when the integrity constraint fails.

Note that all the faults listed in Table 1 appear as possibilities within Table 2, with the notable exceptions of faults ButOn_i and ButOff_i. This highlights an issue with the current design: there is no way for the program to detect a button is faulty. However, the failure of a *single* button leads to fail safe behaviour assuming the correct behaviour of the other button and the operator. If the button sensor is indicating it is not pressed, when it is actually pressed, we have the following possible behaviours: the weight won't be released in state *Top*; it will be lifted back to the top in state *Falling*; and it will be lifted back to the top in state *Bottom* provided the second button is released. If the button sensor is indicating it is pressed, when it is actually not pressed, we have the following possible behaviours: in state *Top* the weight can be released if the other button is pressed (i.e., just one button is actually pressed); in state *Falling* it will continue to fall if the other button is pressed (i.e., just one button is actually pressed); and in state *Bottom* the weight won't be lifted even if the other button is released (i.e., both buttons are actually released).

In these cases the operator is able to detect that the system is not behaving as expected. If there was only a single button, a button failure would be a significant problem because it is not detectable and it can lead to a hazard (which is why our design here has two buttons). In the current design failure of both buttons can lead to a hazard.

The failure of a single integrity constraint may be caused by a number of possible faults, and hence a single integrity constraint failure may not allow one to diagnose which fault has occured. If we assume that there is only a single fault, then if multiple integrity constraints have failed, the likely fault is in the intersection of the sets of possible faults associated with each failed integrity constraint. For example, the failure of both integrity constraints (1) and (2) would indicate that the likely fault is *TOn*, (although a simultaneous failure of *BOn* and *POn*, perhaps due to some common mode failure, is possible).

In some cases there may be a sequence of failures of integrity constraints over time which allow one to progressively narrow down the likely faults. For example, if integrity constraint (6) in Table 2 fails it indicates possible faults of *TOn*, *LOff* and *MOff* in mode *Falling*. These faults lead to different recovery actions. If the *top* sensor has failed on then the recovery action should be to raise an alarm and allow the weight to continue to fall, because if we try to lift the weight we won't know when it is at the top. But if the lock has failed in the locked position or the motor is turned on when it should be off we should revert to explicitly locking the weight and raise an alarm. If after a failure of integrity constraint (6) we continue monitoring integrity constraints then there are two possibilities:

- integrity constraint (2) may fail, which indicates possible faults of *TOn* or *POn*, in which case, taking into account the earlier failure, the most likely fault is *TOn*, and the first action above is appropriate; or
- integrity constraint (5) may fail, which indicates possible faults of *POff*, *LOff* or *MOff*, in which case the most likely fault is either *LOff* or *MOff*, and the second action above is appropriate.

As the focus of this paper is systematically detecting faults, we leave it as a exercise for the reader to explore the error responses for the industrial press.

A more thorough analysis of the multiple integrity constraint failures could take into account the probabilities of each failure in order to determine the most likely cause.

The general approach to fault diagnosis is to start from a failed integrity constraint, *I*, and examine the set, *S*, of faults that may lead to its failure. Then we need to look at the other integrity constraints whose failure indicates faults in *S*. From these constraints we need to look at those that can happen at the same time as *I*, or may follow in sequence from *I*, until we have narrowed the set down to a single fault.

7 Conclusions

Software controlling hardware devices has to be able to cope with failures in the devices being controlled, or failures of the sensors and actuators through which it controls them.

This is especially true in safety-critical applications. Code to explicitly detect faults in devices can complicate the software considerably. In this paper we have examined the use of integrity constraints as a technique for detecting faults, with the aim being that the code to detect faults is generated from the integrity constraints. The integrity constraints considered include:

- those that can only fail if the sensors indicate a physically impossible state,
- invariants on the state of the system,
- those that indicate an inconsistency between the program state and the sensor values,
- those that indicate an invalid transition between states,
- invalid system dynamics, where transitions do not occur within some maximum expected time,
- invalid system dynamics, where the time to a transition is less than some minimum expected time, and
- those that can detect common device or sensor failures.

One advantage of the integrity constraint approach is that once an integrity constraint has been identified, it can be checked after every iteration. This provides a more thorough and systematic approach to checking for faults than ad hoc development of fault detection code. The exception here is the case above where a transition occurs before it is expected to occur; this case only needs to be checked by the code that performs the transition.

Fault diagnosis can be tailored to make use of which integrity constraints have failed. For each integrity constraint one can list the likely faults that caused it to fail. The failure of multiple integrity constraints that have a common cause can help narrow down the most likely fault. One can also identify faults that cannot be detected. Such undetectable faults either require new integrity constraints to be devised to check for them or, if this is not possible, redesign of the system to allow the faults to be detected.

While the methods presented here are phrased in terms of action systems, they could be applied to other equivalent systems of presenting control systems, such as Event-B [18] or TLA [19].

Acknowledgments. This paper has benefited greatly from my collaborations with Cliff Jones and Michael Jackson and other members of IFIP Working Group 2.3 on Programming Methodology. I would like to thank Phil Cook and Larissa Meinicke for feedback on earlier drafts of this paper. This research was supported by Australian Research Council (ARC) Discovery Grant DP0558408, *Analysing and generating fault-tolerant real-time systems* and the EPSRC-funded Trustworthy Ambient Systems (TrAmS) Platform Project.

References

1. Storey, N.: Safety-Critical Computer Systems. Addison-Wesley, Reading (1996)
2. Leveson, N.G.: Safeware: System Safety and Computers. Addison-Wesley, Reading (1995)
3. Anderson, T., Lee, P.A.: Fault Tolerance: Principles and Practice, 2nd edn. Prentice-Hall, Englewood Cliffs (1990)

4. Torres-Pomales, W.: Software fault tolerance: A tutorial. Technical Report TM-2000-210616, NASA Langley Research Centre (October 2000)
5. Randell, B.: On failures and faults. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 18–39. Springer, Heidelberg (2003)
6. Hanmer, R.S.: Patterns for Fault Tolerant Software. Wiley, Chichester (2007)
7. Breitling, M.: Modeling faults of distributed, reactive systems. In: Joseph, M. (ed.) FTRTFT 2000. LNCS, vol. 1926, pp. 58–69. Springer, Heidelberg (2000)
8. Hayes, I., Jackson, M., Jones, C.: Determining the specification of a control system from that of its environment. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 154–169. Springer, Heidelberg (2003)
9. Jones, C.B., Hayes, I.J., Jackson, M.A.: Deriving specifications for systems that are connected to the physical world. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems. LNCS, vol. 4700, pp. 364–390. Springer, Heidelberg (2007)
10. Back, R.J., Sere, K.: Action systems with synchronous communication. In: Programming Concepts, Methods, and Calculi (PROCOMET 1994), pp. 107–126. North-Holland, Amsterdam (1994)
11. Butler, M., Sekerinski, E., Sere, K.: An action system approach to the steam boiler problem. In: Abrial, J.-R., Börger, E., Langmaack, H. (eds.) Dagstuhl Seminar 1995. LNCS, vol. 1165. Springer, Heidelberg (1996)
12. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)
13. McDermid, J., Kelly, T.: Industrial press: Safety case. Technical report, High Integrity Systems Engineering Group, University of York (1996)
14. Fidge, C.J., Hayes, I.J., Watson, G.: The deadline command. IEE Proceedings—Software 146(2), 104–111 (1999)
15. Hayes, I.J., Utting, M.: A sequential real-time refinement calculus. Acta Informatica 37(6), 385–448 (2001)
16. Jackson, M.A.: Problem Frames: Analyzing and structuring software development problems. Addison-Wesley, Reading (2001)
17. Caspi, P., Salem, R.: Threshold and bounded-delay voting in critical control systems. In: Joseph, M. (ed.) FTRTFT 2000. LNCS, vol. 1926, pp. 70–81. Springer, Heidelberg (2000)
18. Abrial, J.R., Mussat, L.: Introducing dynamic constraints in B. In: Bert, D. (ed.) B 1998. LNCS, vol. 1393, pp. 83–128. Springer, Heidelberg (1998)
19. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, Reading (2003)

Event-B Patterns for Specifying Fault-Tolerance in Multi-agent Interaction

Elisabeth Ball and Michael Butler[*]

Dependable Systems and Software Engineering, Electronics and Computer Science,
University of Southampton, UK
{ejb04r,mjb}@ecs.soton.ac.uk

Abstract. Interaction in a multi-agent system is susceptible to failure. A rigorous development of a multi-agent system must include the treatment of fault-tolerance of agent interactions for the agents to be able to continue to function independently. Patterns can be used to capture fault-tolerance techniques. A set of modelling patterns is presented that specify fault-tolerance in Event-B specifications of multi-agent interactions. The purpose of these patterns is to capture common modelling structures for distributed agent interaction in a form that is re-usable on other related developments. The patterns have been applied to a case study of the contract net interaction protocol.

1 Introduction

A fault-tolerant system is one that can continue to function as it was designed in the presence of faults [1]. Fault-tolerance can be introduced into the design of a software system.

Multi-agent systems are systems of distributed software entities that cooperate or compete to achieve individual or shared goals [2]. Agents encapsulate their behaviour and are motivated by their internal goals. The agents can individually respond, pro-actively and reactively, to changes in their environment [3]. The agent metaphor is one approach to creating software systems that are capable of solving distributed problems.

Formal methods are the application of mathematics to model and verify software or hardware systems [4]. Event-B is a mathematical approach for developing formal models of distributed systems that can be used to analyse and reason about the system [5]. Using a formal method to model a system results in a specification of the system that is unambiguous and can be formally verified. The model can be analysed for flaws before the system based on the model is developed [6].

Patterns are intended to make software engineering easier by capturing the expertise of experienced software developers and making it available in a manner that can be re-applied in other developments [7]. The purpose of a pattern is

[*] This research was carried out as part of the EU research projects IST 511599 RODIN (Rigorous open development environment for complex systems rodin.cs.ncl.ac.uk) and ICT 214158 DEPLOY (Industrial deployment of system engineering methods providing high dependability and productivity www.deploy-project.eu/).

M. Butler et al. (Eds.): Fault Tolerance, LNCS 5454, pp. 104–129, 2009.
© Springer-Verlag Berlin Heidelberg 2009

to capture structures and decisions within a design that are common to similar modelling and analysis tasks. They can be re-applied when undertaking similar tasks to in order reduce the duplication of effort.

This paper presents a set of fault-tolerance patterns that have been developed to help specify fault-tolerance in Event-B models of multi-agent systems. A case study based on a specification of the contract net interaction protocol by the Foundation for Intelligent Physical Agents (FIPA) [8] illustrates the application of the patterns.

This paper is structured as follows: Section 2 examines the aspects of multi-agent systems that require fault-tolerance. Section 3 provides an overview of Event-B. Section 4 examines how fault-tolerance patterns can be used in Event-B. Section 5 introduces the contract net case study. The following sections describe each of the patterns in turn. Related work is then examined followed by a conclusion with an outline of possible future work.

2 Fault-Tolerance in Agent Interaction

A fault-tolerant system is one that can continue to function as it was designed in the presence of faults [1]. A multi-agent system has to be able to cope with the faults that can occur in any distributed system.

Fault-tolerance in distributed systems requires that the system can cope with faults in communication and faults in the behaviour of the distributed components. The system must be able to continue to function if a fault leads to a failure in communication between nodes or to a node ceasing to communicate. The system must also be able to cope if a node in the system is prevented from completing a task that it has been delegated.

In this paper we understand a multi-agent system [2] as a grouping of agents that either cooperate or compete in order to fulfill individual or collective goals. Multi-agent systems require many dynamic interactions to be able to function. The agents in the system behave both rationally and autonomously. A fault-tolerant multi-agent system needs to be able to cope with this behaviour. The agent's autonomy can make their behaviour difficult to predict. Rational agents will stop pursuing a goal if they believe that the goal has already been achieved or that it cannot be achieved. An agent that is autonomous is not required to complete any tasks requested by other agents. The task may conflict with its existing goals and, therefore, not be desirable for the agent to complete. The heterogeneity and dynamic interactions of a multi-agent system may lead to agents receiving messages that they do not understand or that are out of expected order. These are not always faults in the individual agents, but they are faults in the interactions of the system. The agents must be able to handle such faults in their interactions and communicate their reactions to these faults.

Development using formal methods can help to ensure correctness by construction [9]. Using formal methods does not guarantee that the developer has not ommitted some aspects of system behaviour from the model that may lead to failure. The patterns presented in this paper add events and variables to Event-B specifications of multi-agent systems to provide tolerance of possible faults that can occur because of the distributed and rational nature of multi-agent systems. The faults dealt with by the patterns are an excessive delay in response, a refusal in response to a request, the request to cancel a previous request, the failure to complete a committed task and the receipt of an unexpected communication.

3 Event-B

Event-B is a mathematical approach for developing formal models of systems [10]. An Event-B model is constructed from a collection of modelling elements. These elements include invariants and events with guards and actions. The modelling elements have attributes expressed using set theory and predicate logic. The development of an Event-B model begins with abstraction and continues with refinement of the abstraction. The abstract machine specifies the initial requirements of the system. The refinement of a model is the process of adding more detail to a model. The refinement of an Event-B abstract machine can be carried out in several steps. More detail is added to the model at each step. Refinement allows models at different abstraction levels to be related. Development is generally, but not exclusively, top-down. Refinement may highlight errors or elements missing from the model that require changes to be made to abstract models.

The focus on atomic events in Event-B creates a representation of a reactive system [11]. The guard of an event represents the necessary conditions on the state of the system for the event to be triggered. When the guard is true the actions of the event may be executed, possibly changing the state and allowing another event to be triggered.

Event-B is designed for modelling distributed systems [5]. Event-B allows new events to be added and single events to be refined into multiple concrete events. This allows a system behaviour to be modelled as a single atomic event and then refined to a set of events that separately model the behaviour. This refinement can model individual processes executing in parallel to perform the behaviour of an abstract event or different events that result in the same actions as the abstract event. Refinement ensures that refined models are consistent with the abstract machine. Creating models of reactive and distributed systems makes Event-B an appropriate formalism as a basis for modelling multi-agent systems.

To create a textual representation of the Event-B models in this paper the events will be presented using the keywords `ANY`, `WHERE`, `THEN` and `END` to structure the model. The event variables of an event will be written between `ANY` and `WHERE`. The guards of the event will be written between `WHERE` and `THEN` and the actions of the event will be written between `THEN` and `END`.

4 Modelling Patterns for Fault-Tolerance

Fault-tolerance is not necessarily a feature of a system that is appropriate to model in detail at the most abstract level. It is often a part of the communication infrastructure or a component of individual nodes and, therefore, will be modelled in refinement.

Each pattern includes a description, interaction diagram and Event-B extracts from the contract net case study. The description for each of the patterns includes a *name*, *fault* statement and *tolerance pattern* statement. The description can be applied to any event-based specification. The fault statement is the potential fault for which the application of the pattern will model a solution. The tolerance pattern statement describes the steps that can be taken to solve the problem in an event-based specification.

The interaction diagrams show how the fault-tolerance techniques can be included in the interactions between the different agent roles. Several of the interaction diagrams show the variations required for one-to-many interaction.

The patterns also include Event-B extracts from a case sutdy that how how the patterns can be applied to a development. These examples make the patterns specific to Event-B development. The other elements of the pattern are more generic and could be suitable for other event-based formal methods.

The fault-tolerance techniques modelled by the patterns will help an agent to continue to provide a service when a fault occurs within a particular interaction. If a tolerated fault occurs in a conversation between two agents an agent may fail to fulfill its goal, but the fault should not prevent the agent from performing its role in another conversation.

The set of fault-tolerance patterns presented in this paper model solutions for faults that can arise in multi-agent systems. This includes faults that are found in ordinary distributed systems. The Timeout pattern prevents an agent from indefinitely waiting for a communication. This allows the agent to cope with faults in either the communication medium, or other nodes or agents in the system. The failure of a node to complete a delegated task is modelled by the Failure pattern. A rational agent altering its goals is modelled by the Cancel pattern. The Refuse pattern allows the system to cope with an agent deciding not to participate in an interaction. The Not-Understood pattern models the reaction of agents to unexpected communications. With the patterns specified in an Event-B development the developer can then refine the models to include more detail on how the system or individual agents will manage these faults.

Applying a Pattern

The patterns can be applied to an existing Event-B development of a multi-agent system to introduce the fault-tolerance techniques to the model. Figure 1 shows how the patterns can be applied to the refinement chain of an existing Event-B development i.e., an abstract model and its refinement. The events and variables that model the abstraction of the pattern can be added to the abstract machine. The events and variables that model the concrete pattern can then be added to a refinement of the abstract machine. The developer can decide where in the refinement chain they want to extend a refinement model to include the concrete pattern. Several refinement steps may be required for refinements between the abstract machine and the model extended by the concrete pattern. The extend relationship requires the addition to, or modification of, the events and variables in the model for the pattern to be included. If the events and variables required for

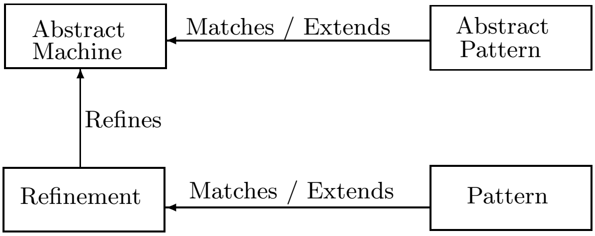


Fig. 1. Using the patterns with an existing model

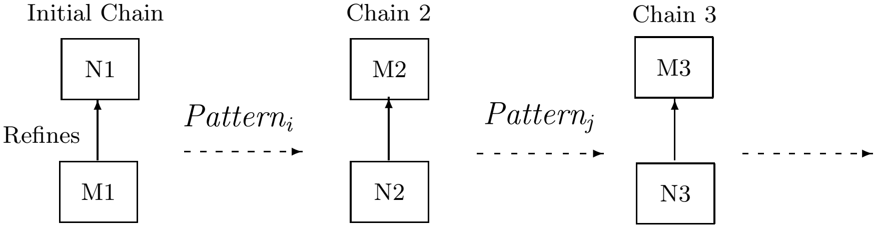


Fig. 2. Effect of applying patterns

the pattern already exist in the model no additions or modifications are necessary. The Event-B examples include gluing invariants that relate the abstract variables to the concrete variables. These can potentially be re-used in the extended refinement chain to help the developer specify the refines relationship.

The Event-B extracts include an abstraction of the pattern and a concrete pattern. Pattern extracts are demonstrated on the basis of the contract net case study. The abstraction of the pattern will need to be added to the Event-B abstract machine. When there are several refinement steps between the abstract machine and the refinement extended by the concrete pattern it will be necessary to make refinement steps to the intermediate models.

Not including the concept of refinement in an Event-B modelling pattern would limit the usefulness of the pattern for providing a complete solution. Including a complete refinement chain may confuse the developer should their refinement chain differ from the one provided. Not providing an abstraction may make it difficult for the developer to find an appropriate abstraction. This approach would only provide an incomplete solution and may lead to the incorrect application of the pattern.

The patterns have each been applied in separate developments to the initial chain. This ensures that there is no dependency between the patterns and, therefore, the order in which they are applied has no importance. All of the patterns have also been applied sequentially to the initial chain. This is to provide assurance that there are no conflicts between the patterns. Figure 2 illustrates how a collection of patterns can be used to extend an Event-B refinement chain. Extending the *Initial Chain* by applying $Pattern_i$ produces *Chain*2 and applying $Pattern_j$ by further extending *Chain*2 produces *Chain*3. $Pattern_j$ could be applied before $Pattern_i$ to produce the same result ($Chain3$). A possible direction for future work is to find a method to prove the orthogonality of the patterns.

5 Case Study

This section describes the contract net interaction protocol. A simplified version of the protocol has been modelled as an Event-B refinement chain. Each of the patterns have been applied to the case study models. The contract net case study involves multiple participants. The developer may need to adapt the patterns for models of one-to-one interaction.

The contract net interaction protocol is a distributed negotiation process [12]. The goal of the contract net is for the initiating agent to find an agent, or group of agents, that offer the most advantageous proposal to carry out a required task. The initiator of the protocol advertises the existence of a task that it needs completing by broadcasting a *call for proposals*. The agents that receive the *call for proposals* can place a bid to complete the task by sending a *proposal*. Participants in the protocol are committed to the bids that they propose. When the initiator selects a bid or a group of bids the participants are informed of the decision and those selected will complete the task. The contract is completed when the participants *inform* the initiator that the task is completed. The case study has been developed using the FIPA specification of the contract net [8].

The development presented here includes an abstract model and one refinement model. The abstract model models conversations between agents and the refinement introduces the agents involved in the conversation to the model. There will be one initiator agent and one or more other agents participating in the conversation. The events of the abstract model show the behaviour of the system moving the conversation to different states to model the progression of the interaction between the agents. The refinement model shows the agents in the system and links the agents to the conversations in which they are involved and moves this relationship between the different states. Initially only successful conversations of the contract net interaction protocol are modelled. The abstract model shown in Figure 3 includes four variables that represent states that the conversation will move through. The set CONVERSATION is a set of abstract values that represent the type for conversations. The *cfp* variable represents the state after a call for proposals has been initiated by an agent. The *responded* variable represents the participating agents responding to the call for proposals. The *selected* variable represents the initiator choosing one or more proposals to accept. The *informed* variable models the state where the selected agents have informed the initiator of the successful completion of the task. The variables are not modelled as disjoint sets. Instead, the order of the conversation is enforced by specifying the variable for each state as a subset of the previous state.

The events of the abstract machine move the conversation through the different states as the conversation progresses. The *callForProposals* event adds a conversation to the *cfp* state. The *respond* event takes a conversation that is in the *cfp* state and puts it in the *responded* state. The *responded* event occurs once and represents sufficient agents sending proposals. The *select* event takes

```
INVARIANTS
  $cfp \subseteq CONVERSATION$
  $responded \subseteq cfp$
  $selected \subseteq responded$
  $informed \subseteq selected$
EVENTS
  callForProposals
       ANY c WHERE
          $c \in CONVERSATION$
          $c \notin cfp$
       THEN
          $cfp := cfp \cup \{c\}$
       END
  respond
       ANY c WHERE
          $c \in cfp$
          $c \notin responded$
       THEN
          $responded := responded \cup \{c\}$
       END
  select
       ANY c WHERE
          $c \in responded$
          $c \notin selected$
       THEN
          $selected := selected \cup \{c\}$
       END
  inform
       ANY c WHERE
          $c \in selected$
          $c \notin informed$
       THEN
          $informed := informed \cup \{c\}$
       END
```

Fig. 3. Abstract machine of the initial chain

$$cfpS,\ proposeS,\ acceptS,\ rejectS,\ informS \in CONVERSATION \leftrightarrow AGENT,$$

$$cfpR \subseteq cfpS,\ proposeR \subseteq proposeS,\ acceptR \subseteq acceptS,$$
$$acceptS \subseteq proposeR,\ informR \subseteq informS,\ rejectR \subseteq rejectS,$$
$$informS \subseteq acceptR,\ proposeS \subseteq cfpR,$$

$$selected = dom(acceptS \cup rejectS),\ cfp = dom(cfpS)$$

Fig. 4. Invariants for the refinement of the initial chain

a conversation that is in the *responded* state and adds it the the *selected* state. The *inform* event takes a conversation that is in the *selected* state and adds it to the *informed* state to complete the conversation.

The refinement of the abstract model incorporates the interaction between the agents involved in the conversation. The invariants for the refinement model are shown in Figure 4. The variables of the model represent messages being sent and received by the agents in the system. The variables that represent a message being sent are suffixed with an ‘S’ and those that represent a message being received are suffixed with an ‘R’. The conversation is between multiple agents and so the variables are specified as relationships between a set of conversations and a set of agents. For example, $c \mapsto a \in cfpS$ means that agent a has been sent a call for proposals message within conversation c and $c \mapsto a \in cfpR$ means that agent a has received a call for proposals message within conversation c. A message must be sent before it can be received and this is modelled by specifying a subset relationship between the sent variables and the received variables, e.g. $cfpR \subseteq cfpS$. Some of the variables from the abstract machine are replaced by

```
sendCfp  REFINES callForProposals
  ANY  c, as, a  WHERE
    $c \in CONVERSATION$
    $c \notin dom(cfpS)$
    $as \in CONVERSATION \leftrightarrow AGENT$
    $a \in AGENT$
    $dom(as) = \{c\}$
    $ran(as) = AGENT \setminus \{a\}$
  THEN
    $cfpS := cfpS \cup as$
  END
sendProposal
  ANY  c, a  WHERE
    $c \mapsto a \in cfpR$
    $c \mapsto a \notin proposeS$
  THEN
    $proposeS := proposeS \cup \{c \mapsto a\}$
  END
responded  REFINES respond
  ANY c WHERE
    $c \in dom(proposeS)$
    $c \notin responded$
  THEN
    $responded := responded \cup \{c\}$
  END
receiveAccept
  ANY  c, a  WHERE
    $c \mapsto a \in acceptS$
    $c \mapsto a \notin acceptR$
  THEN
    $acceptR := acceptR \cup \{c \mapsto a\}$
  END
sendInform
  ANY  c, a  WHERE
    $c \mapsto a \in acceptR$
    $c \mapsto a \notin informS$
  THEN
    $informS := informS \cup \{c \mapsto a\}$
  END
informed  REFINES inform
  ANY  c  WHERE
    $c \in dom(informR)$
    $c \notin informed$
  THEN
    $informed := informed \cup \{c\}$
  END
```

```
receiveCfp
  ANY  c, a  WHERE
    $c \mapsto a \in cfpS$
    $c \mapsto a \notin cfpR$
  THEN
    $cfpR := cfpR \cup \{c \mapsto a\}$
  END
receiveProposal
  ANY  c, a  WHERE
    $c \mapsto a \in proposeS$
    $c \mapsto a \notin proposeR$
  THEN
    $proposeR := proposeR$
          $\cup \{c \mapsto a\}$
  END
select  REFINES select
  ANY c, as, ar WHERE
    $c \in dom(proposeR)$
    $c \notin dom(acceptS)$
    $c \notin dom(rejectS)$
    $as \subseteq \{c\} \lhd proposeR$
    $ar = \{c\} \lhd proposeR \setminus as$
    $c \in responded$
  THEN
    $acceptS := acceptS \cup as$
    $rejectS := rejectS \cup ar$
  END
receiveReject
  ANY  c, a WHERE
    $c \mapsto a \in rejectS$
    $c \mapsto a \notin rejectR$
  THEN
    $rejectR := rejectR$
          $\cup \{c \mapsto a\}$
  END
receiveInform
  ANY  c , a WHERE
    $c \mapsto a \in informS$
    $c \mapsto a \notin informR$
  THEN
    $informR := informR$
          $\cup \{c \mapsto a\}$
  END
```

Fig. 5. Events of the refinement of the initial chain

the message variables in the refinement. The last two invariants are the gluing invariant and specify the refinement relationships between the abstract variables that represent the state of the conversation and the concrete variables that model messages being broadcast. The *responded* and *informed* variables from the abstract model represent states that are internal to the agents. Because they are not included in a conversation they are not refined by relationships between the conversation and agent and no gluing invariant is required.

The events of the refinement are shown in Figure 5. The *sendCfp* event refines the abstract *callForProposals* event. It models the broadcast of a call for proposals message from agent a to all other agents ($AGENT \setminus \{a\}$) by a set of relationships, *as*, between a conversation and the agents in the system and adds it to the *cfpS* variable. The *receiveCfp* event models a message being received by an agent by selecting a relationship, $c \mapsto a$, that is in the *cfpS* variable and adding it to the *cfpR* variable. The *sendProposal* event can occur when there is a relationship in the *cfpR* variable and the proposal is sent when the relationship is added to the *proposeS* variable. The *receiveProposal* event adds a relationship that is in the *proposeS* variable to the *proposeR* variable. The *responded* event is a refinement of the abstract *respond* event and represents the initiator receiving the required responses. The *select* event broadcasts two different messages. One group of agents, *as*, will receive an accept message in response to their proposal and another group of agents, *ar*, will receive a reject message. The *receiveAccept* and *receiveReject* events represent those messages being received by the participants. The event and variables that model the rejection, receiveReject, rejectS and rejectR can be ommitted, as they do not affect the rest of the interaction, but in a multi-agent system it may be useful for an agent to know that it has been rejected, so it can adapt its behaviour in the future. The *sendInform* event models an agent that has received an accept message, sending an inform message following the successful completion of their task. The *receiveInform* event represents this message being received. The final *informed* event refines the abstract *inform* event and models the initiator concluding that the contract has been successfully completed following the receipt of at least one inform message.

6 Timeout Pattern

Name:Timeout
Fault: An agent may become blocked during a conversation whilst waiting for replies.
Tolerance Pattern: Specify a state for the conversation that models a deadline passing. Add an event to the specification that will change the state of the conversation from before the deadline to after the deadline. Split the event for receiving the replies into two. One event will have a guard that is true before the deadline and one will have a guard that is true after the deadline. The action of the event after the deadline will inform agents of their failure to meet the deadline.

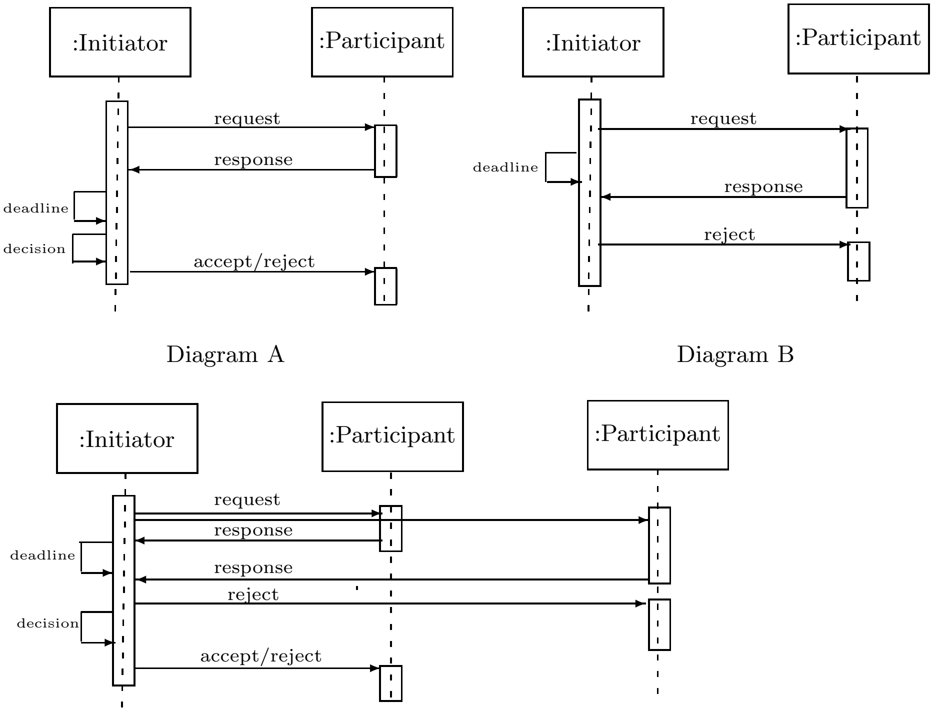


Diagram A

Diagram B

Diagram C

Fig. 6. Timeout: Interaction diagrams

In the case of a communication failure, or the failure of another agent or node in the system, an agent that continues to wait for a response to a communication may wait an excessively long time or may never receive the reply. This is not practical for most systems, especially a multi-agent system that may be expected to be able to adapt under such circumstances. An agent should be able to decide to either continue the conversation without waiting for a response or to resolve its goal in another way, when it becomes likely that a response will not be forthcoming. An agent may be required to make a decision on how long it should wait depending on its goals for the efficiency of its current task.

The Timeout pattern prevents an agent from becoming blocked whilst waiting for a reply. It does this by modelling a deadline after which the behaviour of the system changes. The interaction diagrams in Figure 6 show the messages that are exchanged between the roles involved in the conversation.

The Timeout pattern requires that any messages received after the deadline will lead to the responding agent being informed of their failure to meet the deadline. The agent that has the role of initiating the request will be responsible for enforcing the deadline. The assumption is made in the model that the deadline will be specified by the initiator in the messages sent and that a global clock is available to all of the agents involved. Diagram A shows a successful one-to-one interaction with the response to a request being received by the

```
INVARIANTS
    $failed \subseteq cfp$
EVENTS    callForProposals
        ANY  c  WHERE
            $c \in CONVERSATION$
            $c \notin cfp$
        THEN
            $cfp := cfp \cup \{c\}$
        END
    failure
        ANY  c  WHERE
            $c \in cfp$
            $c \notin failed$
            $c \notin responded$
        THEN
            $failed := failed \cup \{c\}$
        END
    respond
        ANY  c  WHERE
            $c \in cfp$
            $c \notin responded$
        THEN
            $responded := responded \cup \{c\}$
        END
```

Fig. 7. Timeout: Abstract events

$$beforeTimeout \subseteq dom(cfpS)$$
$$afterTimeout \subseteq beforeTimeout$$
$$proposeRD \subseteq proposeS$$
$$rejectSD \subseteq proposeRD$$
$$rejectRD \subseteq rejectSD$$
$$failedCfp \subseteq afterTimeout$$
$$failedCfp \cap dom(proposeR) = \varnothing$$
$$failed = failedCfp$$

Fig. 8. Timeout: Refinement invariants

initiator before the deadline. In this case the reply from the initiator will depend on the initiator's decision about the response. Diagram B shows the initiator's deadline occurring before the response is received and in this case the reply from the initiator is a rejection of the response. Diagram C shows how a one-to-many interaction can affect the Timeout pattern. Responses are received from different participating agents before and after the deadline has passed. Those received before the deadline will elicit replies that depend on a decision that is made by the initiating agent. Those received after will result in a reject notification.

Figure 7 shows the *callForProposals*, *respond* and *failure* events that are required in the abstract model for the Timeout pattern to be applied. The *callForProposals* and *respond* events are already present in the initial chain. The *failure* event has been added to model the system responding when no proposals are received before the deadline. The pattern for the timeout could be more general than that taken from the contract net case study. Any request by an agent that waits for a response could use the Timeout pattern to ensure that the requesting agent does not wait indefinitely. The abstract pattern in Figure 7 conforms to this general request-response pattern. The invariant conditions for the refinement are shown in Figure 8. To create the states for before and after the deadline two variables have been added to the model; *beforeTimeout* and *afterTimeout*. The pattern could have been specified with just the *afterTimeout* variable. Both variables were included to make the effect of the deadline clear in the model. The *beforeTimeout* variable is specified as a subset of the domain of the *cfpS* variable so the timeout cannot occur before the conversation has begun. Variables have been added to the model to represent the proposals that are received after the deadline, *proposeRD*, the reject messages sent in response to these proposals, *rejectSD*, and then received, *rejectRD*. The *failedCfp* variable refines the abstract *failed* variable to model the state when the deadline has passed, $failedCfp \subseteq afterTimeout$, and no proposals have been received, $failedCfp \cap dom(proposeR) = \varnothing$.

Events have been added to the initial chain and existing events have been modified to apply the Timeout pattern. The new and modified events are shown in Figure 9 where the names of the new events, and the modifications to existing events, are underlined. The *sendCfp* event has an additional action that adds the conversation to the *beforeTimeout* variable. The guard of the *receiveProposal* event has been strengthened so that it can only occur when the conversation is not in the *afterTimeout* variable. The new *deadline* event moves the conversation from the state *beforeTimeout* into the state *afterTimeout*. The new *receiveProposal2* event can only occur when the conversation is in the *afterTimeout* variable. The action of the event adds the relationship from the *proposeS* variable to the new *proposeRD* variable. The new *sendReject* event will take a relationship that is in the *proposeRD* variable and add it to the *rejectSD* variable. This models the initiator responding with a reject message to any proposals received after the timeout. The new *receiveReject2* event will take a relationship that is in the *rejectSD* variable and add it to *rejectRD* variable. Instead of adding this as a new event a developer could merge it with the existing *receiveReject* event from the initial chain. The new *failToPropose* event refines the abstract *failure* event that was added to the abstract model for the Timeout pattern. It can occur after the deadline has passed and no proposals have been received.

7 Refuse Pattern

Name: Refuse
Fault: An agent cannot support the action requested.
Tolerance Pattern: Add an event for an agent to send a refuse message in response to a request and an event for an agent to receive a refuse message.

Not all agents that receive a request will be able to fulfill it. The request may be in conflict with the agent's own goals. This could be due to the agent being overloaded, or the agent is competing against the requestor and it would not be in their interest to help. Software design does not always implement the concept of a refusal. Object-based systems use the term 'design by contract' to describe an obligation held by an object that it cannot alter at runtime [13]. The autonomy of agents means that the obligations between agents are weaker than in design by contract and a multi-agent system must be designed to cope when an agent refuses to undertake a request.

```
sendCfp  REFINES callForProposals
  ANY  c, as, a  WHERE
    $c \in CONVERSATION$
    $c \notin dom(cfpS)$
    $as \in CONVERSATION \leftrightarrow AGENT$
    $a \in AGENT$
    $dom(as) = \{c\}$
    $ran(as) = AGENT \setminus \{a\}$
  THEN
    $cfpS := cfpS \cup as$
    $\underline{beforeTimeout := beforeTimeout \cup \{c\}}$
  END
deadline
  ANY  c   WHERE
    $c \in beforeTimeout$
    $c \notin afterTimeout$
  THEN
    $afterTimeout := afterTimeout \cup \{c\}$
  END
sendReject
  ANY  c , a   WHERE
    $c \mapsto a \in proposeRD$
    $c \mapsto a \notin rejectSD$
  THEN
    $rejectSD := rejectSD \cup \{c \mapsto a\}$
  END

failToPropose  REFINES failure
  ANY  c  WHERE
    $c \notin dom(proposeR)$
    $c \in afterTimeout$
    $c \notin failedCfp$
  THEN
    $failedCfp := failedCfp \cup \{c\}$
  END
```

```
receiveProposal
  ANY  c, a  WHERE
    $c \mapsto a \in proposeS$
    $c \mapsto a \notin proposeR$
    $\underline{c \notin afterTimeout}$
  THEN
    $proposeR := proposeR$
        $\cup \{c \mapsto a\}$
  END
receiveProposal2
  ANY  c, a   WHERE
    $c \mapsto a \in proposeS$
    $c \mapsto a \notin proposeR$
    $c \mapsto a \notin proposeRD$
    $c \in afterTimeout$
  THEN
    $proposeRD := proposeRD$
        $\cup \{c \mapsto a\}$
  END
receiveReject2
  ANY  c, a   WHERE
    $c \mapsto a \in rejectSD$
    $c \mapsto a \notin rejectRD$
  THEN
    $rejectRD := rejectRD$
        $\cup \{c \mapsto a\}$
  END
```

Fig. 9. Timeout: Concrete events

The Refuse pattern allows an agent to respond to a request that it cannot support, that is not correctly requested or that the requesting agent is not authorised to request. An agent is allowed a choice when responding to a request. The agent can either agree to fulfill the request or it can refuse.

Figure 10 shows interaction diagrams for the Refuse pattern. Diagram A shows a one-to-one interaction. The initiator agent sends a request to a participant agent. The participant agent can respond with either an accept or refuse message. The initiator will then make a decision and the interaction may fail if the accept message is not suitable or a refuse message was sent. Diagrams B and C show one-to-many interaction. Diagram B shows the case where a combination of accept and refuse messages are received in response to the request. Diagram C

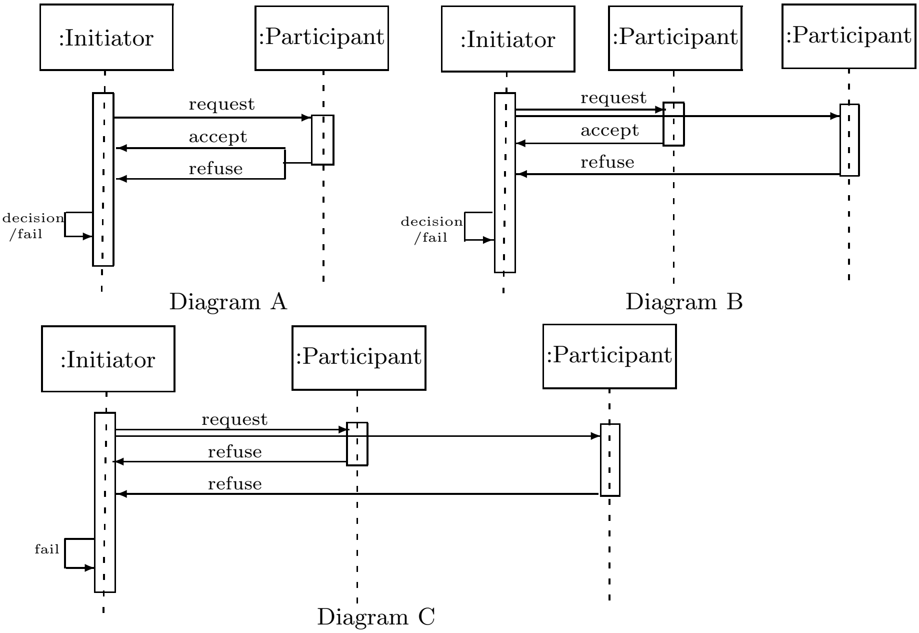


Diagram A

Diagram B

Diagram C

Fig. 10. Refuse: Interaction diagrams

$$
\begin{array}{l}
\mathit{refuseS} \ \subseteq \ \mathit{cfpR} \\
\mathit{refuseR} \ \subseteq \ \mathit{refuseS} \\
\mathit{refuseS} \ \cap \ \mathit{proposeS} \ = \ \varnothing \\
\mathit{failedCommit} \ \subseteq \ \ \mathit{dom}(\mathit{refuseR}) \\
\mathit{failedCommit} \ \cap \ \ \mathit{dom}(\mathit{proposeR}) \ = \ \ \varnothing \\
\mathit{failed} \ = \ \mathit{failedCommit}
\end{array}
$$

Fig. 11. Refuse: Concrete invariants

shows the case where only refuse messages are received and the only outcome is a failure of the interaction.

The events that are required in the abstract machine for the Refuse pattern are the same as those shown in Figure 7 for the Timeout pattern. To model the Refuse pattern in the refinement of the initial chain three variables and three events have been added. In the contract net case study the refusals are modelled so they are equivalent to the proposals. The invariants in Figure 11 specify variables that model sending and receiving refuse messages. An additional invariant specifies that the proposals and refusals for a conversation cannot be from the same agent, $\mathit{refuseS} \cap \mathit{proposeS} = \varnothing$. The *failedCommit* variable models that state of the conversation when all of the replies are refusals. This variable refines the abstract *failed* variable.

The events for the pattern are shown in Figure 12. The guard of the *sendProposal* event from the initial chain has been modified to prevent an agent that has made a refusal for the conversation from also making a proposal. The

```
sendProposal
    ANY c, a WHERE
        $c \mapsto a \in cfpR$
        $c \mapsto a \notin proposeS$
        $c \mapsto a \notin refuseS$
    THEN
        $proposeS := proposeS \cup \{c \mapsto a\}$
    END
```

```
sendRefusal
    ANY c, a WHERE
        $c \mapsto a \in cfpR$
        $c \mapsto a \notin proposeS$
        $c \mapsto a \notin refuseS$
    THEN
        $refuseS := refuseS$
            $\cup \{c \mapsto a\}$
    END
```

```
receiveRefusal
    ANY c, a WHERE
        $c \mapsto a \in refuseS$
        $c \mapsto a \notin refuseR$
    THEN
        $refuseR := refuseR \cup \{c \mapsto a\}$
    END
```

```
failToCommit REFINES failure
    ANY c WHERE
        $c \in dom(refuseR)$
        $c \notin dom(proposeR)$
        $c \notin failedCommit$
    THEN
        $failedCommit :=$
            $failedCommit \cup \{c\}$
    END
```

Fig. 12. Refuse: Concrete events

sendRefusal event adds a relationship that is in the *cfpR* variable to the *refuseS* variable. The *receiveRefusal* event takes a relationship that is in the *refuseS* variable and adds it to the *refuseR* variable. The *failToCommit* event models the case when all of the responses are refusals and the conversation fails. This is a refinement of the abstract *fail* event.

8 Cancel Pattern

Name: Cancel
Fault: The requesting agent no longer requires an action to be performed.
Tolerance Pattern: Add an event to the specification for an agent to send a cancel message to an agent that has agreed to perform an action on its behalf. Add an event for that agent to receive a cancel message. Further events need to be added to allow the agent to reply with either an inform, if they have cancelled the action, or a failure, if they have not, and for those messages to be received.

Once an agent has requested an action they can then request that it is cancelled. An agent that exhibits rational behaviour may change its goals because the goal conflicts with other goals, the agent no longer desires the goal is fulfilled or the agent no longer believes that the goal can be fulfilled [2]. For the initiating agent to ensure that its beliefs about its environment are consistent it needs to know if the agents to whom it has delegated tasks have managed to undo any

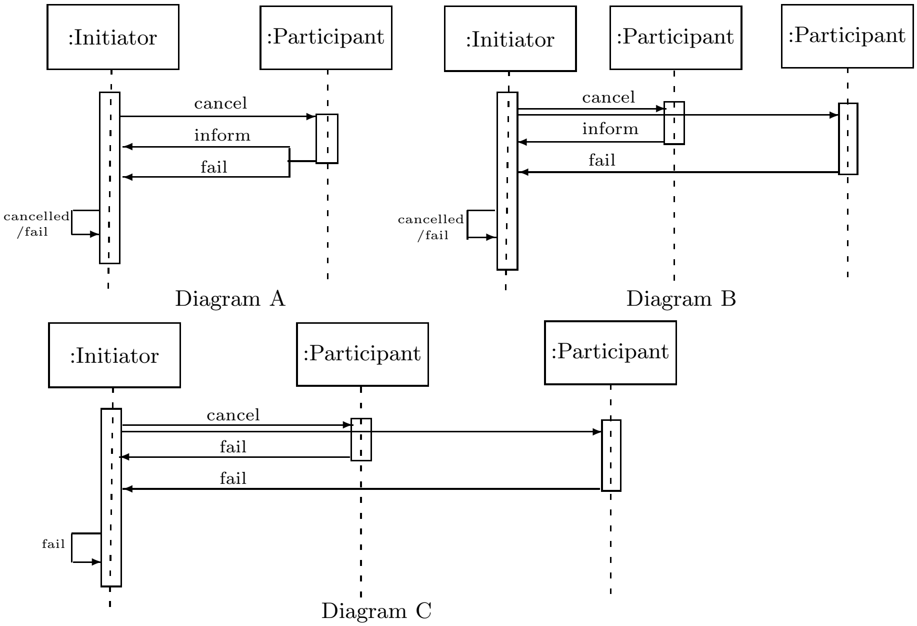


Fig. 13. Cancel: Interaction diagrams

actions they have performed. The responses of the agents may affect the actions the initiating agent takes in response to its change of goals.

The Cancel pattern allows the agent that initiated the conversation to cancel the conversation at any point. The Cancel pattern will cancel a single request in a one-to-one conversation and will broadcast the cancellation in a one-to-many conversation to cancel all of the requests. Figure 13 shows interaction diagrams for the Cancel pattern. Diagram A shows a one-to-one interaction. The initiator agent sends a cancel message to a participant agent. The participant agent can respond with either an inform message if they have successfully cancelled or a fail message if they have not. The initiator will then act according to its knowledge about the state of the system. Diagrams B and C show one-to-many interaction. Diagram B shows the case where a combination of inform and fail messages are received in response to the cancel message. Diagram C shows the case where only fail messages are received and the cancelling of the action fails.

The Cancel pattern requires a new variable and event to be added to the abstract machine of the initial chain. The abstract pattern in Figure 14 shows the *cancel* event moving the conversation into the *cancelled* state.

The Cancel Pattern is modelled in the refinement as a collection of events that can occur at any point in the conversation. Events model a cancel message being sent from the initiating agent and received by the other agents involved. Events are also required to model the participating agents responding to the

```
INVARIANTS
 $cancelled \subseteq cfp$
EVENTS
  cancel
    ANY c WHERE
      $c \in cfp$
      $c \notin cancelled$
    THEN
      $cancelled := cancelled \cup \{c\}$
    END
```

Fig. 14. Cancel: Abstract events

$$\begin{array}{l}
cancelS \subseteq cfpS \\
cancelR \subseteq cancelS \\
informCancelS \subseteq cancelR \\
informCancelR \subseteq informCancelS \\
failCancelS \subseteq cancelR \\
failCancelR \subseteq failCancelS \\
informCancelled \subseteq dom(informCancelR) \\
failCancelled \subseteq dom(failCancelR) \\
informCancelS \cap failCancelS = \varnothing \\
informCancelled \cap failCancelled = \varnothing \\
cancelled = informCancelled \cup failCancelled
\end{array}$$

Fig. 15. Cancel: Refinement invariants

cancel request to inform the initiating agent whether they have managed to cancel their actions.

Figure 15 shows the invariant conditions from the Event-B extract of the Cancel pattern. The variables represent the states of the system as messages are sent and received. The *cancelS* variable is a subset of the *cfpS* variable so a conversation cannot be cancelled before it has begun. All of the other variables are specified as subsets according to the order of the messages that they represent being sent and received. *InformCancelS* and *failCancelS* are specified so the same agent cannot send an inform and fail message in the same conversation. The *informCancelled* and *failCancelled* variables are specified so the conversation cannot be in both states. An invariant condition specifies the intersection of the two variables as empty. The final invariant condition is the gluing invariant that relates the abstract *cancel* variable to a conjunction of the *informCancelled* and *failCancelled* variables.

Figure 16 shows the events that have been added to the initial refinement model to specify the Cancel pattern. The *sendCancel* event can be triggered by the initiating agent at any point in the conversation. The cancel message is broadcast to every agent involved in the conversation, $as = \{c\} \lhd cfpS$. The *receiveCancel* event allows the participants to receive the cancel message. The *sendInformCancel* and *sendFailCancel* events model the participants sending a message to the initiator about the success or failure of the cancellation. The *receiveInformCancel* and *receiveFailCancel* events model the initiator receiving the message. The last two events, *informCancelled* and *failCancelled*, refine the abstract *cancel* event and model the initiator evaluatings the success of the cancellation. The guards for the two events specify that at least one inform or fail cancel message has been received. The developer may want to strengthen these guards. For example, the guard of the *informCancelled* event could be strengthened to specify that all of the agents have replied with an inform message, $\{c\} \lhd informCancelR = AGENT \setminus \{a\}$, or that no fail messages have been received, $c \notin dom(failCancelR)$.

```
sendCancel
  ANY  c , as   WHERE
    $c \in dom(cfpS)$
    $as = \{c\} \lhd cfpS$
    $c \notin dom(cancelS)$
  THEN
    $cancelS := cancelS \cup as$
  END
sendInformCancel
  ANY  c, a   WHERE
    $c \mapsto a \in cancelR$
    $c \mapsto a \notin informCancelS$
    $c \mapsto a \notin failCancelS$
  THEN
    $informCancelS := informCancelS$
          $\cup \{c \mapsto a\}$
  END
receiveInformCancel
  ANY  c, a   WHERE
    $c \mapsto a \in informCancelS$
    $c \mapsto a \notin informCancelR$
  THEN
    $informCancelR := informCancelR$
          $\cup \{c \mapsto a\}$
  END
informCancelled REFINES cancel
  ANY  c   WHERE
    $c \in dom(informCancelR)$
    $c \notin informCancelled$
    $c \notin failCancelled$
  THEN
    $informCancelled := informCancelled$
                        $\cup \{c\}$
  END
```

```
receiveCancel
  ANY  c, a   WHERE
    $c \mapsto a \in cancelS$
    $c \mapsto a \notin cancelR$
  THEN
    $cancelR := cancelR \cup$
                $\{c \mapsto a\}$
  END
sendFailCancel
  ANY  c, a   WHERE
    $c \mapsto a \in cancelR$
    $c \mapsto a \notin failCancelS$
    $c \mapsto a \notin informCancelS$
  THEN
    $failCancelS := failCancelS$
                 $\cup \{c \mapsto a\}$
  END
receiveFailCancel
  ANY  c, a   WHERE
    $c \mapsto a \in failCancelS$
    $c \mapsto a \notin failCancelR$
  THEN
    $failCancelR := failCancelR$
                 $\cup \{c \mapsto a\}$
  END
failCancelled REFINES cancel
  ANY  c   WHERE
    $c \in dom(failCancelR)$
    $c \notin failCancelled$
    $c \notin informCancelled$
  THEN
    $failCancelled := failCancelled$
                        $\cup \{c\}$
  END
```

Fig. 16. Cancel: Concrete events

9 Failure Pattern

Name: Failure
Fault: An agent is prevented from carrying out an agreed action.
Tolerance Pattern: Add an event for an agent to send a failure message after they have committed to perform an action on behalf of another agent. Add an event for an agent to receive a failure message and an event for the system to respond to the failure.

An agent that makes a commitment to perform an action may be prevented from carrying it out. The agent that requested the action should be informed of this failure so that its beliefs do not become inconsistent.

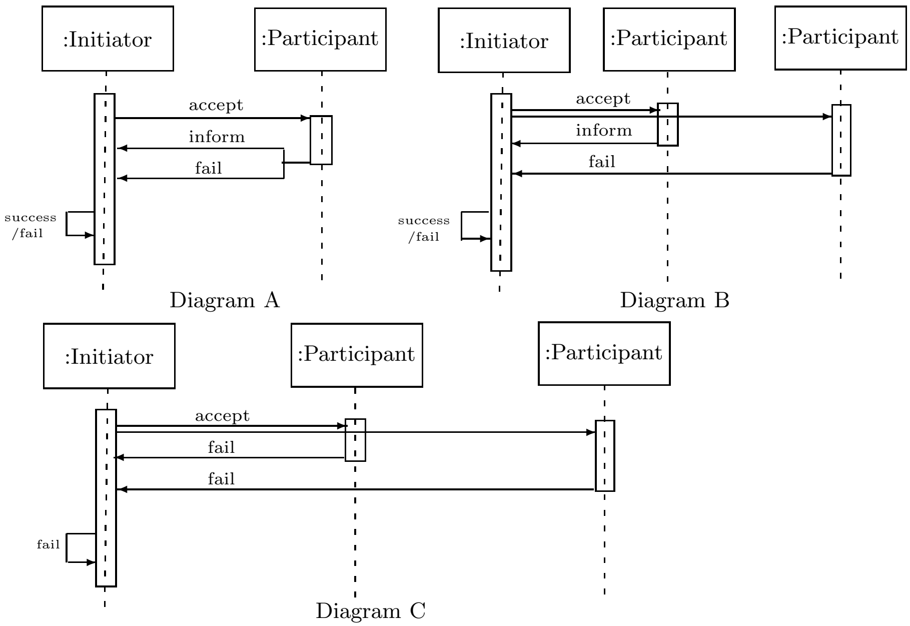


Diagram A

Diagram B

Diagram C

Fig. 17. Failure: Interaction diagrams

Figure 17 shows interaction diagrams for the Failure pattern. Diagram A shows a one-to-one interaction. The initiator agent sends a message that requests an action to a participant agent. The participant agent can respond with either an inform message, if they have successfully carried out the action, or a fail message, if they have not. Diagrams B and C show one-to-many interaction. Diagram B shows the case where a combination of inform and fail messages are received in response to the accept message. The initiator will then be able to evaluate whether the task was carried out successfully. Diagram C shows the case where only fail messages are received. The Failure pattern is similar to the Refuse pattern where the responding agent has a choice of two replies that affect the outcome of the interaction differently. It occurs at a different point in the conversation. The Refuse pattern is used before a commitment is made and the Failure pattern is required after a commitment has been made.

Figure 18 shows the events from the abstract machine that are related to the Failure pattern. The failure pattern specifies the failure of the conversation after the selection of the proposals has been made. Either the *inform* event or the *failure* event can complete the conversation.

Figure 19 shows the invariants added for the Failure pattern. The Event-B models the agents involved in the contract net interaction protocol sending failure messages instead of inform messages after they have had their proposal accepted. The conversation cannot succeed and fail and this is modelled by

```
inform
    ANY  c  WHERE
      $c \in selected$
      $c \notin informed$
      $\underline{c \notin failed}$
    THEN
      $informed := informed \cup \{c\}$
    END
failure
    ANY  c  WHERE
      $c \in selected$
      $c \notin failed$
      $c \notin failed$
      $c \notin informed$
    THEN
      $failed := failed \cup \{c\}$
    END
```

Fig. 18. Failure: Abstract Events

```
$failS \subseteq acceptR$
$failR \subseteq failS$
$failed1 \subseteq dom(failR)$
$informed \cap failed1 = \varnothing$
$failed = failed1$
```

Fig. 19. Failure: Refinement invariants

```
sendFail
    ANY   c, a   WHERE
      $c \mapsto a \in acceptR$
      $c \mapsto a \notin failS$
      $c \mapsto a \notin informS$
    THEN
      $failS := failS \cup \{c \mapsto a\}$
    END
failed  REFINES failure
    ANY  c   WHERE
      $c \in dom(failR)$
      $c \notin failed1$
      $c \notin informed$
    THEN
      $failed1 := failed1 \cup \{c\}$
    END
```

```
receiveFail
    ANY  c, a   WHERE
        $c \mapsto a \in failS$
        $c \mapsto a \notin failR$
    THEN
        $failR := failR \cup \{c \mapsto a\}$
    END
```

Fig. 20. Failure: Concrete events

an invariant condition that specifies the intersection of the informed and failed variables as empty.

Figure 20 shows the three events that are added to the initial concrete model. The *sendFail* event models a participant having received an accept message that instructs it to carry out a task, $c \mapsto a \in acceptR$, sending a failure message in response. The *receiveFail* event models the initiator receiving the failure message. The *failed* event refines the abstract *failure* event and can occur after a failure message has been received.

10 Not-Understood Pattern

Name:Not-Understood
Fault: An agent receives a message that it does not expect or does not recognise.
Tolerance Pattern: Specify an event for receiving a message with an unknown or unexpected performative. Specify the action as replying with a not-understood message. Specify an event for receiving a not-understood message.

The autonomy of the agents means that there is no guarantee of their behaviour and the non-hierarchical nature of multi-agent systems often means that there is no single point of control. For agents in a multi-agent system to maintain a correct understanding of their environment they need to communicate with the other agents in the system to be aware of the actions of the other agents. This can create a large number of messages being passed between agents for them to be able to negotiate, query and inform. The possible heterogeneity of the agents means that they may have a different understanding of interaction protocols. The possibility of receiving arbitrary messages increases with each of these factors and the system needs to be able to cope with such faults. In a multi-agent system that has been developed in a top-down manner the faults that may lead to an arbitrary message being sent should not occur. However, it may be that some of the system components have been developed separately or that the formal development of the system is limited to modelling the interactions. In these cases the inclusion of the Not-Understood pattern will provide assurance that the system can still tolerate the faults outlined above.

The concept of the not-understood message is described in [8]. The not-understood message communicates that the sending agent has received a message that it does not understand. A not-understood message can be sent or received at any point in the conversation.

It is suggested in [8] that the action taken in response to a not-understood message should be different when the conversation involves broadcast messages and sub-protocols than that taken as part of a one-to-one conversation. It may be inappropriate to cancel the conversation when there are multiple agents performing sub-protocols. Each response to a not-understood message should be evaluated depending on the status of the conversation and is not specified by the Not-Understood pattern.

The Not-Understood pattern involves agents receiving an arbitrary message, responding with a not-understood message and agents receiving a not-understood message. The action taken by the agent to cope with the potential fault is not modelled and is left for the developer to treat.

Figure 21 shows an interaction diagram for the Not-Understood pattern. The interaction diagram shows an interaction between any two agent roles. One agent sends another agent a message that the receiving agent does not understand. The response from the receiving agent will be to reply with a not-understood message. The action taken by the agent that receives the not-understood message depends on their role in the conversation and the stage of the conversation.

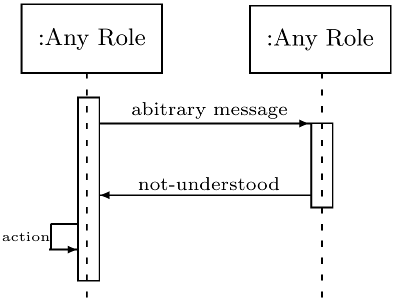


Fig. 21. Not-Understood: Interaction diagram

```
INVARIANTS
  $recUnknown \subseteq CONVERSATION$
  $recNotUnderstood \subseteq cfp$
EVENTS
  arbitraryComm
     ANY  c   WHERE
        $c \in cfp$
     THEN
        $recUnknown := recUnknown \cup \{c\}$
     END

  receiveNotUnderstood
     ANY  c   WHERE
        $c \in cfp$
     THEN
        $recNotUnderstood :=$
           $recNotUnderstood \cup \{c\}$
     END
```

Fig. 22. Not-Understood: Abstract events

To model the Not-Understood pattern two events have been added to the initial abstract machine for the contract net case study. The events and variables are shown in Figure 22. The *arbitraryComm* event models the unrecognised message being received. The *receiveNotUnderstood* event abstractly model an agent receiving a not-understood message.

Figure 23 shows the extract from the Event-B refinement model that models the Not-Understood pattern in the Contract Net case study. The *unknownR* variable represents an arbitrary message being received and the *notUnderstoodS* variable represents a not-understood message being sent in response to the receipt of an arbitrary message. The *notUnderstoodR* variable represents a not-understood message being received.

The *receiveArbitraryComm* event models the receipt of a message that is not understood by the receiving agent. The *sendNotUnderstood* event models a not-understood message being sent in response to the receipt of this message. The *receiveNotUnderstood* event models an agent receiving a not-understood message. Further refinements of the pattern will model the agent's reactions to receiving the not-understood message. An initiator agent may decide to cancel the conversation or they may decide that the conversation has failed. The decisions by the agents will depend on the stage of the conversation when the not-understood message is received. This is left for the developer to decide and model.

```
INVARIANTS
          $unknownR \subseteq cfpS$
          $notUnderstoodS \subseteq unknownR$
          $recUnknown = dom(notUnderstoodS)$
          $notUnderstoodR \subseteq notUnderstoodS$
          $recNotUnderstood = dom(notUnderstoodR)$

EVENTS
receiveArbitraryComm
      ANY  c, a   WHERE
         $c \mapsto a \in cfpS$
      THEN
         $unknownR := unknownR \cup \{c \mapsto a\}$
      END
sendNotUnderstood  REFINES arbitraryComm
      ANY  c, a   WHERE
         $c \mapsto a \in unknownR$
         $c \mapsto a \notin notUnderstoodS$
      THEN
         $notUnderstoodS := notUnderstoodS \cup \{c \mapsto a\}$
      END
receiveNotUnderstood  REFINES receiveNotUnderstood
      ANY  c, a   WHERE
         $c \mapsto a \in notUnderstoodS$
         $c \mapsto a \notin notUnderstoodR$
      THEN
         $notUnderstoodR := notUnderstoodR \cup \{c \mapsto a\}$
      END
```

Fig. 23. Not-Understood: Concrete invariants and events

11 Related Work

This section describes work that is related to the ideas presented in this paper. This work outlines approaches for constructing patterns in the B-Method and Event-B. Other work of interest are design patterns for multi-agent systems, particularly patterns that can be integrated with goal models that are used in multi-agent system design. Other fault-tolerance techniques for multi-agent systems have been investigated and are summarised in this section.

The B-Method is used in [14] to specify patterns, such as those identified in [7], as abstract machines. The pattern machines are instantiated by including another B model in the machine using the B-Method's inclusion mechanism. Pattern models can be composed to create a new pattern by using the inclusion mechanism to construct a new machine from the separate patterns. These patterns are specified at a single level of abstraction and are based on object-oriented development methods.

A set of patterns that solve design problems that are common when using the B-Method has been produced in [15]. The patterns they present include a pattern to associate multiple B machines, a pattern to produce unique objects and patterns for creating sub and super-types of B machines. The patterns are implemented as either extracts of B machines or a description of how different mechanisms from the B-Method can be used to solve a described problem alongside an example of the patterns use. As with those described above, these patterns attempt to introduce some object-oriented concepts into B machines, are at a single level of abstraction and mainly address structural relationships between machines.

A refinement pattern for modelling time constraints in Event-B is presented in [16]. A pattern is produced by constructing a generic Event-B model that specifies the time constraints as a superpostion refinement. This model can be re-used to produce new refinements of the model to which the pattern is being applied. The authors suggest that it would be possible to prove the pattern model and the proof obligations generated by the pattern would not need to be discharged for the development model.

There are several methods for the use of patterns in the development of multi-agent systems. They are described and used with informal models. Coordination patterns, including a pattern of the contract net protocol, are presented in [17], patterns for mobile agent design are presented in [18], and [19] present patterns for implementing agents in object-oriented architectures. A strategy for constructing and using design patterns for agent systems that uses goals can be found in [20]. The patterns can be combined using a pattern language to construct a multi-agent system design.

The *extend* relationship used in Figure 1 of this paper is similar to those found in [21], but has not been formally defined.

The patterns presented in this paper provide fault-tolerance for the agents so they can continue to provide a service. Further strategies for managing faults in agent conversations include adapting general fault-tolerance techniques, such as replication [22], redundancy [23] and checkpoints [24], to multi-agent systems. Creating patterns for the specification of these fault-tolerance strategies in multi-agent systems is a possible direction for future work.

12 Summary

Event-B has been developed for modelling reactive and distributed systems and our experience shows that it is suited to the specification of multi-agent systems. The patterns presented above allow the developer to incorporate fault-tolerant behaviour in an Event-B development of a multi-agent system.

The patterns are presented as three elements: a description, interaction diagrams and Event-B examples. The Event-B examples make the patterns specific to Event-B development. The other elements of the pattern are more generic and could be suitable for other event-based formal methods. The inclusion of an abstraction of the pattern creates a pattern that can be fully integrated into the refinement chain of a development. The Event-B extracts included from the Contract Net case study show how the patterns can be applied to the model of a complex multi-agent system. They also provide a re-usable specification of the pattern at a single level of refinement.

Providing an abstract and concrete pattern example will offer the developer guidance on how the pattern can be integrated into an Event-B development that uses refinement. The related work described above use patterns either as a superposition refinement to an Event-B model or as a component to a model. Integrating a pattern into the refinement chain of a development offers the advantages of making the pattern a fundamental part of the development. It is present in the abstraction of the model and can be analysed at all levels of abstraction.

References

1. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable Secure Computing 1(1), 11–33 (2004)
2. Ferber, J.: Multi-Agent Systems: Introduction to Distributed Artificial Intelligence. Addison-Wesley, Reading (1999)
3. Jennings, N.R.: On agent-based software engineering. Artificial Intelligence 117, 277–296 (2000)
4. Storey, N.: Safety-Critical Computer Systems. Pearson Education Limited, Bath (1996)
5. Abrial, J.R., Mussat, L.: Introducing dynamic constraints in B. In: Bert, D. (ed.) B 1998. LNCS, vol. 1393, pp. 83–128. Springer, Heidelberg (1998)
6. Jackson, D.: Dependable software by design. Scientific American - American Edition 294(6), 68 (2006)
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
8. FIPA: FIPA contract net interaction protocol specification. Available From: Technical report, FIPA (2002), `http://www.fipa.org/specs/fipa00029/SC00029H.pdf`
9. Leavens, G., Abrial, J., Batory, D., Butler, M., Coglio, A., Fisler, K., Hehner, E., Jones, C., Miller, D., Peyton-Jones, S.: Roadmap for enhanced languages and methods to aid verification. In: Proceedings of the 5th international conference on Generative programming and component engineering, Portland, Oregan, USA, pp. 221–236. ACM Press, New York (2006)
10. Abrial, J., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: Application to Event-B. Fundamenta Informaticae, XXI 77(1-2), 1–28 (2006)
11. Jones, C.B.: RODIN deliverable D9. preliminary report on methodology. Technical report, University of Newcastle-upon-Tyne, UK (2005), `http://rodin.cs.ncl.ac.uk/deliverables/rodinD9.pdf`
12. Smith, R.: The contract net protocol: High-level communication and control in a distributed problem solver. IEEE Transactions on Computers 29(12), 1104–1113 (1980)
13. Meyer, B.: Object-oriented software construction. Prentice-Hall, Inc., Upper Saddle River (1997)

14. Blazy, S., Gervais, F., Laleau, R.: Reuse of specification patterns with the B Method. In: Bert, D., Bowen, J.P., King, S. (eds.) ZB 2003. LNCS, vol. 2651, pp. 40–57. Springer, Heidelberg (2003)
15. Chan, E., Robinson, K., Welch, B.: Patterns for B: Bridging formal and informal development. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 125–139. Springer, Heidelberg (2006)
16. Cansell, D., Méry, D.: Time constraint patterns for Event B development. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 140–154. Springer, Heidelberg (2006)
17. Deugo, D., Weiss, M., Kendall, E.: Reusable patterns for agent coordination. In: Coordination of Internet Agents: Models, Technologies and Applications, pp. 347–368. Springer, Heidelberg (2001)
18. Aridor, Y., Lange, D.: Agent design patterns: elements of agent application design. In: Proceedings of the second international conference on autonomous agents, pp. 108–115. ACM Press, New York (1998)
19. Schelfthout, K., Coninx, T., Helleboogh, A., Holvoet, T., Steegmans, E., Weyns, D.: Agent implementation patterns. In: Debenham, J., Henderson-Sellers, B., Jennings, N., Odell, J. (eds.) Proceedings of the OOPSLA 2002 Workshop on Agent-Oriented Methodologies, pp. 119–130 (2002)
20. Weiss, M.: Pattern-driven design of agent systems: Approach and case study. In: Eder, J., Missikoff, M. (eds.) CAiSE 2003. LNCS, vol. 2681, pp. 711–723. Springer, Heidelberg (2003)
21. Back, R.: Incremental software construction with refinement diagrams. In: Broy, M., Gruenbauer, J., Harel, D., Hoare, T. (eds.) Engineering Theories of Software Intensive Systems: Proceedings of the NATO Advanced Study Institute on Engineering Theories of Software Intensive Systems, Marktoberdorf, Germany, pp. 3–46. Springer, Heidelberg (2005)
22. Fedoruk, A., Deters, R.: Improving fault-tolerance by replicating agents. In: Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2, pp. 737–744. ACM Press, New York (2002)
23. Kumar, S., Cohen, P.: Towards a fault-tolerant multi-agent system architecture. In: Proceedings of the Fourth International Conference on Autonomous Agents, pp. 459–466. ACM Press, New York (2000)
24. Wang, L., Hon, F.L., Goswami, D., Wei, Z.: A fault-tolerant multi-agent development framework. In: Cao, J., Yang, L.T., Guo, M., Lau, F. (eds.) ISPA 2004. LNCS, vol. 3358, pp. 126–135. Springer, Heidelberg (2004)

Formal Reasoning about Fault Tolerance and Parallelism in Communicating Systems

Linas Laibinis[1], Elena Troubitsyna[1], and Sari Leppänen[2]

[1] Åbo Akademi University, Finland
{Linas.Laibinis,Elena.Troubitsyna}@abo.fi
[2] Nokia Research Center, Finland
Sari.Leppanen@nokia.com

Abstract. Telecommunication systems should have a high degree of availability, i.e., high probability of correct provision of requested services. To achieve this, correctness of software for such systems and system fault tolerance should be ensured. In our previous work we proposed an approach to formalisation and extension of Lyra – a top-down service-oriented method for development of communicating systems. Lyra is based on transformation and decomposition of models expressed in UML2. We formalised Lyra in the B Method by proposing a set of formal specification and refinement patterns reflecting the essential Lyra models and transformations. At the same time, we also focused on integration of fault tolerance mechanisms into the entire Lyra development flow. In this paper, we extend our Lyra formalisation to model parallel execution of services. This significantly increases both complexity and flexibility of the presented models.

Keywords: communicating systems, service-oriented development, fault tolerance, parallel execution, UML, B Method.

1 Introduction

Modern telecommunication systems are usually distributed software-intensive systems providing a large variety of services to their users. Development of software for such systems is inherently complex and error prone. However, software failures might lead to unavailability or incorrect provision of system services, which in turn could incur significant financial losses. Hence it is important to guarantee correctness of software for telecommunication systems.

Nokia Research Center has developed the design method Lyra [7] – a UML2-based service-oriented method specific to the domain of communicating systems and communication protocols. The design flow of Lyra is based on the concepts of decomposition and preservation of the externally observable behaviour. The system behaviour is modularised and organised into hierarchical layers according to the external communication and related interfaces. It allows the designers to derive the distributed network architecture from the functional system requirements via a number of model transformations.

M. Butler et al. (Eds.): Fault Tolerance, LNCS 5454, pp. 130–151, 2009.
© Springer-Verlag Berlin Heidelberg 2009

From the beginning Lyra has been developed in such a way that it would be possible to bring formal methods (such as program refinement, model checking, model-based testing etc.) into more extensive industrial use. A formalisation of the Lyra development would allow us to ensure correctness of system design via automatic and formally verified construction. The achievement of such a formalisation would be considered as significant added value for industry.

In our previous work [6,5] we proposed a set of formal specification and refinement patterns reflecting the essential models and transformations of Lyra. Our approach is based on stepwise refinement of a formal system model in the B Method [1] – a formal refinement-based framework with automatic tool support. Moreover, to achieve system fault tolerance, we extended Lyra to integrate modelling of fault tolerance mechanisms into the entire development flow. We demonstrated how to formally specify error recovery by rollbacks as well as reason about error recovery termination.

In this paper we show how to extend our Lyra formalisation to model parallel execution of services. This presents us with a number of challenges. We show how to gradually unfold hierarchical structure of service execution in the presence of parallelism. Moreover, we demonstrate how such an extension affects the fault tolerance mechanisms incorporated into our formal models. The extension makes our formal models significantly more complicated. However, it also gives the developers more flexibility in defining service architecture as well as choosing possible recovery actions.

2 Previous Work

In this section we give a brief overview of on our previous results [6,5] on formalising and verifying the Lyra development process. This work form the basis for new results presented in the next section.

2.1 Formalising Lyra

Lyra [7] is a model-driven and component-based design method for the development of communicating systems and communication protocols, developed in the Nokia Research Center. The method covers all industrial specification and design phases from pre-standardisation to final implementation.

Lyra has four main phases: *Service Specification*, *Service Decomposition*, *Service Distribution* and *Service Implementation*. The *Service Specification* phase focuses on defining services provided by the system and their users. In the *Service Decomposition* phase the abstract model produced at the previous stage is decomposed in a stepwise and top-down fashion into a set of service components and logical interfaces between them. In the *Service Distribution* phase, the logical architecture of services is distributed over a given platform architecture. Finally, in the *Service Implementation* phase, the structural elements are integrated into the target environment. Examples of Lyra UML models from the Service Specification phase of a positioning system are shown on Fig.1.

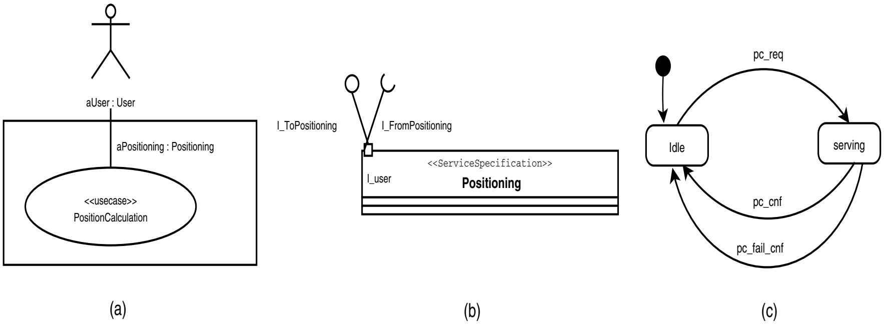


Fig. 1. (a) Domain Model. (b) Class Diagram of Positioning. (c) State Diagram.

To formalise the Lyra development process, we choose the B Method as our formal framework. The B Method [1] is an approach for the industrial development of highly dependable software. Recently the B method has been extended by the Event B framework [2,8], which enables modelling of event-based systems. Event B is particularly suitable for developing distributed, parallel and reactive systems. The tool support available for B, e.g. Atelier B [4] and the Rodin platform [10], provides us with the assistance for the entire development process. The formal development presented in this paper was verified in the Event B framework using the Rodin platform.

The B Method adopts the top-down approach to system development. The basic idea underlying stepwise development in B is to design the system implementation gradually, by a number of correctness preserving steps called *refinements*. The refinement process starts from creating an abstract specification and finishes with generating executable code. The intermediate stages yield the specifications containing a mixture of abstract mathematical constructs and executable programming artefacts.

While formalising Lyra, we single out a generic concept of a communicating service component and propose B patterns for specifying and refining it. In the refinement process a service component is decomposed into a set of service components of smaller granularity specified according to the proposed pattern. Moreover, we demonstrate that the process of distributing service components between network elements can also be captured by the notion of refinement. Below we present an excerpt from a B specification pattern of an abstract communicating service component (*ACC*).

MACHINE ACC
SEES ACC_Data

VARIABLES
  in_data
  out_data
  res
  ...

```
INVARIANTS
    $inv1 : in\_data \in DATA$
    $inv2 : out\_data \in DATA$
    $inv3 : res \in DATA$
    ...

EVENT input
    ANY
        $param$
    WHERE
        $grd1 : param \in DATA \wedge \neg(param = NIL)$
    THEN
        $act1 : in\_data := param$
    END

EVENT calculate
    WHEN
        $grd1 : \neg(in\_data = NIL)$
    THEN
        $act1 : out\_data :\in DATA \setminus \{NIL\}$
    END

EVENT output
    WHEN
        $grd1 : \neg(out\_data = NIL)$
    THEN
        $act1 : res := out\_data$
        $act2 : in\_data, out\_data := NIL, NIL$
    END

END
```

A B specifcation, called an *abstract machine*, encapsulates a local state (program variables) and provides operations on the state. In the Event B framework, such operations are called *events*. The events can be defined as

$$\textbf{WHEN}\ g\ \textbf{THEN}\ S\ \textbf{END}$$

or, in case of a parameterised event, as

$$\textbf{ANY}\ vl\ \textbf{WHERE}\ g\ \textbf{THEN}\ S\ \textbf{END}$$

where vl is a list of new local variables (parameters), g is a state predicate, and S is a B statement describing how the program state is affected by the event.

The events describe system reactions when the given **WHEN** or **WHERE** conditions are satisfied. The **INVARIANT** clause contains the properties of the system (expressed as predicates on the program state) that should be preserved during system execution. The data structures needed for specification of the system are defined in a separate module called *context*. For example, the abstract type $DATA$ and constant NIL used in the above specification are defined in the context ACC_Data, which can be accessed ("seen") by the abstract machine ACC.

The presented specification pattern is deliberatively made very simple. It describes a service component in a very abstract way – a service component simply receives some request data as the input, non-deterministically calculates nonempty result, which is then returned as the output. Using this specification as the starting point of our formal development gives us sufficient freedom to refine it into different kinds of service components. In particular, both the service components providing single services and the service components responsible for orchestrating service execution (called service directors) can be developed as refinements of the presented specification. Moreover, the defined specification and refinement patterns can be repeatedly used to gradually unfold the hierachical structure of service execution.

The proposed approach to formalising Lyra in B allows us to verify correctness of the Lyra decomposition and distribution phases. In development of real systems we merely have to establish by proof that the corresponding components in a specific functional or network architecture are valid instantiations of these patterns. All together this constitutes a basis for automating industrial design flow of communicating systems.

2.2 Introducing Fault Tolerance in the Lyra Development Flow

Currently the Lyra methodology addresses fault tolerance very abstractly, by representing not only successful but also failed service provision in the Lyra UML models. However, it leaves aside modelling of mechanisms for detecting and recovering from errors – the fault tolerance mechanisms. We argue that, by integrating explicit representation of the means for fault tolerance into the entire development process, we establish a basis for constructing systems that are better resistant to errors, i.e., achieve better system dependability.

In practice, service execution can fail in different ways – service components (including service directors) can be "too busy" or not responding, they can fail to produce the expected results because of subcomponent failure or internal software error, communication messages between service components can be lost and so on.

In our top-down approach for system development we start with a very abstract representation of the system. At this point, we implicitly assume that the low-level mechanisms to detect different kinds of failures are already in place. Therefore, we can focus on high-level handling of such failures. For example, we can rely on the assumption that, e.g., timeouts are used to detect lost communication and produce the corresponding error messages for service components to react to such situations. The subsequent formal system development by refinement allows us to not only gradually unfold the hierarchical system architecture but also introduce implementation details for both handling and detecting different kinds of failures. Next we will discuss how to extend the Lyra design method to integrate modelling of fault tolerance.

In the first development stage of Lyra we set a scene for reasoning about fault tolerance by modelling not only successful service provision but also service failure. In the next development stage – *Service Decomposition* – we elaborate on representation of the causes of service failures and the means for fault tolerance.

In the *Service Decomposition* phase we decompose the service provided by a service component into a number of stages (subservices). The service component can execute certain subservices itself as well as request other service components to do it. According to Lyra, the flow of service execution is managed by a special service component called *Service Director*. *Service Director* co-ordinates the execution flow by requesting the required subservices from the external service components.

In general, execution of any stage of a service can fail. In its turn, this might lead to failure of the entire service provision. Therefore, while specifying *Service Director*, we should ensure that it does not only orchestrates the fault-free execution flow but also handles erroneous situations. Indeed, as a result of requesting a particular subservice, *Service Director* can obtain a normal response containing the requested data or a notification about an error. As a reaction to the occurred error, *Service Director* might

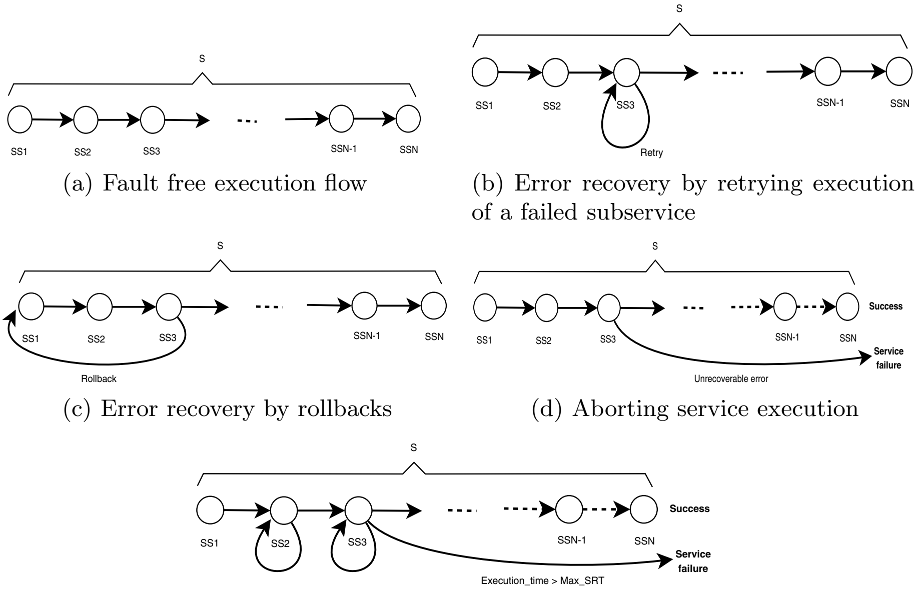
- retry the execution of the failed subservice,
- repeat the execution of several previous subservices (i.e., roll back in the service execution flow) and then retry the failed subservice,
- abort the execution of the entire service.

The reaction of *Service Director* depends on the criticality of an occurred error: the more critical is the error, the larger part of the execution flow has to be involved in the error recovery. Moreover, the most critical (i.e., unrecoverable) errors lead to aborting the entire service. In Fig.2(a) we illustrate a fault free execution of the service S composed of subservices $S_1, \ldots,\ S_N$. Different error recovery mechanisms used in the presence of errors are shown in Fig.2(b) - 2(d).

Let us observe that each service should be provided within a certain finite period of time – the *maximal service response time Max_SRT*. In our model this time is passed as a parameter of the service request. Since each attempt of subservice execution takes some time, service execution might be aborted even if only recoverable errors have occurred but the overall service execution time has already exceeded *Max_SRT*. Therefore, by introducing *Max_SRT* in our model, we also guarantee termination of error recovery, i.e., disallow infinite retries and rollbacks, as shown in Fig.2(e).

3 Fault Tolerance in the Presence of Parallelism

Our formal model briefly described in the previous section assumes sequential execution of subservices. However, in practice, some of subservices can be executed in parallel. Such simultaneous service execution directly affects the fault



(a) Fault free execution flow

(b) Error recovery by retrying execution of a failed subservice

(c) Error recovery by rollbacks

(d) Aborting service execution

(e) Aborting the service due to timeout

Fig. 2. Service decomposition: faults in the execution flow

tolerance mechanisms incorporated into our B models. As a result, they become more complicated. However, at the same time it provides additional, more flexible options for error recovery that can be attempted by *Service Director*.

3.1 Modelling Execution Flow

The information about all subservices and their required execution order becomes available at the Service Decomposition phase. This knowledge can be formalised as a sequence of (subsets of) subservices, e.g., as a data structure

$$Task \ : \ seq(\mathcal{P}(SERVICE))$$

Here $SERVICE$ is a set of all possible subservices. The sequence $Task$ essentially describes the control flow for the top service in terms of required subservices. At the same time, it also indicates which subservices can be executed in parallel.[1]

However, currently the Event B framework does not include sequences as a part of the supported language. Therefore, we define $Task$ in an equivalent way – as a partial surjection, for each involved subservice returning its order of execution:

$$Task \ : \ SERVICE \mathrel{+\!\!\!\!\!\twoheadrightarrow} 1..max\_task$$

[1] In this formalisation we assume that a particular subservice can occur only once in the execution flow.

where $1..max\_task$ is the interval (set) of integer numbers between 1 and the predefined constant max_task, which specifies the number of steps in the service execution flow[2].

For example, $Task$ can be instantiated as

$$Task \ = \ < \{S1, S2\}, \ \{S3, S4, S5\}, \ \{S6\} >$$

defines the top service as a task that should start by executing the services $S1$ and $S2$ in parallel, then continuing by parallel execution of the services $S3$, $S4$, and $S5$, and, finally, finishing the task by executing the single service $S6$.

Essentially, the sequence $Task$ defines the data dependencies between subservices. Also, $Task$ can be considered as the most liberal (from point of view of parallel execution) model of service execution. In the Service Distribution phase the knowledge about the given network architecture becomes available. This can reduce the parallelism of the service control flow by making certain services that can be executed in parallel to be executed in a particular order enforced by the provided architecture.

Therefore, $Task$ is basically the desired model of service execution that will serve as the reference point for our formal development. The actual service execution flow is modelled in by the sequence $Next$, which is defined in a similar way as $Task$:

$$Next \ : \ SERVICE \twoheadrightarrow INDEX$$

where INDEX is defined as the interval $1..max\_next$. The predefined constant max_next specifies the number of steps (tasks) in the actual service execution flow.

Since at the Service Decomposition phase we do not know anything about future service distribution, $Next$ is modelled as an abstract function (sequence), i.e., without giving its exact definition. However, it should be compatible with $Task$. More precisely, if $Task$ requires that certain services S_i and S_j should be executed in a particular order, this order should be preserved in the sequence modelled by $Next$. However, $Next$ can split (allowed by $Task$) parallel execution of the given services by sequentially executing them in any order.

Thus the function $Next$ abstractly models the actual control flow of the top service. It is fully defined (instantiated) only in the refinement step corresponding to the Service Distribution phase. For example, the following instantiation of $Next$ would be correct with respect to $Task$ defined above:

$$Next \ = \ < \{S2\}, \ \{S1\}, \ \{S4\}, \ \{S3, S5\}, \ \{S6\} >$$

It is easy to see that, as required by $Task$, the services $S1$ and $S2$ are still executed before the services $S3$, $S4$, and $S5$, which in turn are executed before the final service $S6$.

[2] Sequences are often modelled in a reverse way – as total functions mapping an interval of integers to the set of data items. However, the proposed here equivalent representation is more suitable for formulating the required properties.

In general, the compatibility property between $Task$ and $Next$ can be formulated in the following way:

$$\forall s1, s2. \{s1, s2\} \subseteq dom(Task) \wedge Task(s1) < Task(s2) \Rightarrow Next(s1) < Next(s2)$$

The definitions of $Task$ and $Next$ makes it evident that this is essentially an order preservation property.

The described model connecting the desired service execution flow $Task$ and the actual execution flow $Next$ covers only simple case when the current service director is responsible for directly managing all the involved subservices. However, in practice the service architecture is often of hierarchical nature, when the current service component (*Service Director*) can delegate a part of service execution to other service directors. In other words, we have to take into account that *Service Director* itself can become distributed, i.e., different parts of service execution could be orchestrated by distinct lower level service directors residing on different network elements. In that case, for every service director, there is a separate $Next$ sequence modelling the corresponding part of the service execution flow. All these control flows should complement each other and also be compatible with $Task$.

To model such a complex case, we introduce other service directors as special services that can be requested by the current service director. This allows us to describe the actual service execution flow as a mixture of single subservices and special service director services. For example, in the above example $Next$ can be instantiated as

$$Next = < \{S2\}, \{SD1\}, \{S3\}, \{SD2\} >$$

where $SD1, SD2$ are lower level service directors that we delegate a part of service execution. All such special services are introduced as elements of the special set SD which is a (strict) subset of $SERVICE$:

$$SD \in \mathcal{P}(SERVICE) \wedge SD \subset SERVICE$$

It is not enough to introduce lower level service directors into the service execution flow. We have to know exactly execution of which subservices they are responsible for. This can be modelled as a separate data structure SD_task relating single subservices to the introduced service directors:

$$SD\_tasks \in SERVICE \nrightarrow SD$$

For the above example, SD_tasks can be instantiated as follows:

$$SD\_tasks = \{S1 \mapsto SD1, S4 \mapsto SD1, S5 \mapsto SD2, S6 \mapsto SD2\}$$

The domain of SD_tasks are all the services that are delegated to lower level service directors, while the remaining services covered by $Next$ can be called single services. Mathematically,

$$dom(Next) = dom(SD\_tasks) \cup Single\_services$$

where

$$dom(SD\_tasks) \;=\; dom(Task)\backslash dom(Next)$$

and

$$Single\_services \;=\; dom(Task) \cap dom(Next)$$

where $S_1 \backslash S_2$ is the set subtraction operation.

Combining the function *Next* with the information provided by the function *SD_tasks* gives us the approximate execution order of all the involved subservices. For this goal, we introduce the new data structure *Ex_Order* defined as the following functional composition:

$$Ex\_Order \;=\; (id(Single\_services) \cup SD\_tasks);\; Next$$

where *id* is the relational identity operator.

In the expression $id(Single\_services) \cup SD\_tasks$ all the single subservices are mapped to themselves, while the other subservices are mapped to the corresponding service directors responsible for their execution. Composing this function with *Next* gives us a new function that maps all the involved subservices to their corresponding execution order. The order of service execution is only ”approximated” because we cannot know the exact order of service execution within lower level service directors since they will be modelled in detail later, in the following refinement steps.

Using the defined mathematical structures, we can now reformulate the compatibility condition to cover this more complex case of service architecture:

$$\forall s1, s2. \{s1, s2\} \subseteq dom(Task) \;\wedge\; Not\_same\_SD(\{s1, s2\}) \wedge$$

$$Task(s1) < Task(s2) \;\;\Rightarrow\;\; Ex\_Order(s1) < Ex\_Order(s2)$$

where the additional condition $Not\_same\_SD(\{s1, s2\})$ is a shorthand for

$$\{s1, s2\} \subseteq dom(SD\_tasks) \Rightarrow SD\_tasks(s1) \neq SD\_tasks(s2)$$

It requires that the subservices in question should not belong to the same (lower level) service director. The precise execution order within lower level service directors will be enforced in the later refinement steps focusing on them.

3.2 Modelling Recovery Actions

As we described before, *Service Director* is a service component responsible for orchestrating service execution. It monitors execution of the activated subservices and attempts different possible recovery actions when these services fail. Obviously, introducing parallel execution of subservices (described in the previous subsection) directly affects the behaviour of *Service Director.*

Now, at each execution step in the service execution flow, several subservices can be activated and run simultaneously. *Service Director* should monitor their execution and react asynchronously whenever any of these services sends its response. This response can indicate either success or a failure of the corresponding subservice.

The formal model for fault tolerance presented in Section 2.2 is still valid. However, taking into account parallel execution of services presents *Service Director* with new options for its recovery actions. For example, getting response from one of active subservices may mean that some or all of the remaining active subservices should be cancelled (i.e., interrupted). Also, some of the old recovery action (like retrying of service execution) are now parameterised with a set of subservices. The parameter indicates which subservices should be affected by the corresponding recovery actions.

Below we present the current full list of actions that *Service Director* may take after it receives and analyses the response from any of active subservices. Consequently, *Service Director* might

- **Proceed** to the next service execution step. In case of successful termination of all involved subservices (**complete success**).
- **Wait** for response from the remaining active subservices. In case of successful termination of one of few active subservices (**partial success**).
- **Abort** the entire service and send a failure response to the user or requesting component. In case of an unrecoverable error or the service timeout.
- **Cancel** a set of subservices by sending the cancelling requests to interrupt their execution (partial abort). In case of a failure which requires to retry or rollback in the service execution flow.
- **Repeat** a set of subservices by sending the initial requests to re-execute the corresponding subservices. In case of a recoverable failure.
- **Rollback** to a certain point of the service execution flow. In case of a recoverable failure.

Similarly like modelling the service execution flow, we cope with arising complexity by introducing abstract data structures and then stating their expected properties. These abstract data structures will be instantiated with concrete data during actual development of communication systems.

First, we introduce an abstract function *Eval* that, for given subservice and the current state of *Service Director*, returns the specific action that *Service Director* should take in that particular situation. It is defined as

$$Eval \in INDEX \times STATE \rightarrow RESPONSE$$

Recall that INDEX is defined as the interval $1..max\_next$. $RESPONSE$ is the enumerated set containing possible outcomes of evaluation, i.e., it is defined as $\{SUCCESS, ABORT, CANCEL,\ REPEAT, ROLLBACK,\ CONTINUE\}$. We assume that the state of *Service Director* contains the information about the status of all currently executed parallel subservices. Therefore, the *Service Director*'s decision is not just based on failure or success of a particular subservice, but also takes the whole picture into account.

In some cases, *Service Director* needs additional information about, e.g., which active subservices should be affected (for **Cancel** and **Repeat**) or to which service execution point to rollback (for **Rollback**). Again, this information is abstractly introduced by the corresponding functions $Cancel$, $Repeat$ and $Rollback$. They are defined as follows.

$Repeat \in INDEX \times STATE \nrightarrow \mathbb{P}(SERVICE)$
$Cancel \in INDEX \times STATE \nrightarrow \mathbb{P}(SERVICE)$
$Rollback \in (2 \mathinner{..} max\_next) \times STATE \nrightarrow 1 \mathinner{..} (max\_next - 1)$

Moreover, these abstract function should be consistent with the main evaluation function $Eval$. In particular, the domain of $Eval$ is partially partitioned by the corresponding domains of $Cancel$, $Repeat$ and $Rollback$.

$Eval[dom(Repeat)] = \{REPEAT\}$
$Eval[dom(Cancel)] = \{CANCEL\}$
$Eval[dom(Rollback)] = \{ROLLBACK\}$

In the next section we will demonstrate how all the introduced abstract data structures are used to model *Service Director* handling simultaneously executed subservices.

4 Specification of *Service Director*

In this section we present a specification of *Service Director* – a component responsible for orchestrating (possibly parallel) service execution involving several external subservices. This specification is a refinement of the specification pattern ACC presented earlier. In this refinement step we elaborate on the "input" and "calculate" operations of the above pattern. In addition, we introduce new events controlling the service execution flow based on the responses from the involved service components. This is modelled by using the information about the Lyra Service Decomposition and Service Distribution phases.

At the same time, fault tolerance mechanisms are introduced. In other words, we model handling of both normal (successful) and abnormal (failure) events by *Service Director. Service Director* reacts to these events by analysing asynchronous responses from the corresponding service components and then deciding on the further course of action. The possible recovery actions were described in the previous section.

The given specification heavily relies on the introduced abstract data structures modelling the service execution flow and possible recovery actions. The definitions of these data structures as well as their expected properties were defined in the previous section.

In the terms of the modelled order of events within a service component, the behaviour of the ACC component can be graphically represented as

$$\text{Input} \rightarrow \text{Calculate} \rightarrow \text{Output}$$

The refined specification of *Service Director* adds additional execution steps before calculating the output. These execution steps model interleaving of time progress and handling of responses from other service components by *Service Director*. The behaviour of the *ServiceDirector* then can be represented as

$$\text{Input} \rightarrow (\text{Timer} \rightarrow \text{HandlingResponse})^* \rightarrow \text{Calculate} \rightarrow \text{Output}$$

Timer and *HandlingResponse* are repeatedly executed until service execution terminates by either producing the expected results or aborting. *HandlingResponse* itself is a composite operation including asynchronous reading of the responses, calculating and storing intermediate results, and, if necessary, enabling possible recovery actions.

Below we present several excerpts from the *Service Director* specification, illustrating the most important aspects of its functionality.

4.1 New Variables

The new specification *ServiceDirector* is a superposition refinement of *ACC*, i.e., it inherits all old variables from *ACC* and also adds new ones to model execution flow and time.

MACHINE ServiceDirector
REFINES ACC
SEES SD_Data

VARIABLES
...
curr_state
curr_task
results
finished
active
time_left
old_time_left
resp
...

The variable *curr_state* abstractly models the internal state of a service component, while the variable *curr_task* contains the index of the current task to be executed. The task can involve several subservices to be executed in parallel. *curr_task* is basically an index of service execution sequence modelled by *Next*. The variable *results* stores the intermediate results from all previous (successful) service execution steps. In case of succesful completion of service execution, these results are then used to calculate the final service result in *Calculate*. The variable *finished* is just a flag indicating whether service execution orchestrated by *Service Director* is finished. The variable *active* contains the currently active subservices that *Service Director* is waiting responses from. The variables *time_left* and *old_time_left* are two snapshots of service execution time. They are used to ensure that service execution does not exceed the given maximal service execution time. And, finally, *resp* contains the result of the last analysed subservice response, which indicates what the course of action *Service Director* is going to take at the moment.

4.2 Refining Input

The **input** event from the *ACC* specification is refined by splitting it into two different events **input_START** and **input_STOP**. It models two distinct situations in which a service component can get a request. The first situation is when a service component or a service director is asked to start (or restart) its execution by a higher level component or user. The other situation is when a service component is asked to stop (cancel) a request that it is already executing.

The exact nature of a request is indicated by the forwarded parameters that are analysed in the guards of the corresponding events, e.g., $Mode(param) = START$ or $Mode(param) = STOP$. The additional parameter *time* gives the maximal service time that cannot be exceeded by service component execution. It is used to initialise the variables $time\_left$ and old_time_left. The input parameters are also used to set correct values to the internal state, e.g., $curr\_state := Init\_state(param)$, and other variables.

```
EVENT input_START
REFINES input
    ANY
        $param$
        $time$
    WHERE
        $grd1 : param \in DATA \wedge time \in \mathbb{N}_1$
        $grd2 : \neg(param = NIL) \wedge Mode(param) = START$
    THEN
        $act1 : in\_data, curr\_state := param, Init\_state(param)$
        $act2 : curr\_task, active := 1, Next^{-1}[\{1\}]$
        $act3 : finished, received := FALSE, TRUE$
        $act4 : time\_left, old\_time\_left := time, time$
        $act5 : results, resp := \varnothing, SUCCESS$
    END
```

```
EVENT input_STOP
REFINES input
    ANY
        $param$
    WHERE
        $grd1 : param \in DATA \wedge \neg(param = NIL) \wedge Mode(param) = STOP$
    THEN
        $act1 : in\_data := param$
        $act3 : finished := TRUE$
        $act5 : resp := ABORT$
    END
```

4.3 Refining Calculate

Similarly, we split the **calculate** event of *ACC* based on whether service execution has been succesfully finished or it was aborted. The latter could happen in the situations when *Service Director* could not handle failures of some subservice providers, the maximal service execution time has expired, or *Service Director* has been interrupted by an external request (*Service Director* from a higher level or the user).

In the case of successful termination, the final output is calculated based on the internal state of *Service Director* and the accumulated results from previous succesful execution steps. In case of abortive termination, the corresponding error data are returned.

```
EVENT calculate_SUCCESS
REFINES calculate
    WHEN
        $grd1 : \neg(in\_data = NIL) \wedge finished = TRUE \wedge \neg(resp = ABORT)$
    THEN
        $act1 : out\_data := Output(curr\_state \mapsto results)$
    END
```

```
EVENT calculate_ABORT
REFINES calculate
    WHEN
        $grd1 : \neg(in\_data = NIL) \wedge finished = TRUE \wedge resp = ABORT$
    THEN
        $act1 : out\_data := Abort\_data$
    END
```

4.4 Modelling Progress of Time

The progress of time and interuption of service execution by the given timeout (the maximal service execution time) is modelled by the corresponding timer events. The first, **timer** event models the progress of time by nondeterministically decreasing the variable *time_left*. If that is not possible, the second, **timer_out** event triggers the abortive termination of service execution by setting the corresponding flags.

Such abstract modelling of time progress implicitly assumes the existence of a constantly ticking clock. If necessary, the explicit variable(s) and event specifying the behaviour of such a clock may be easily introduced. Then nondeterministic update of the variable $time\_left$ can be refined into an ordinary assignment by directly using the accumulated value of the passed (since the last check) time.

The timer events are executed by interleaving them with the events handling received responses from the underlying service components. To ensure that they are executed in the proper order, the second timestamp variable old_time_left is used. The handler events are enabled only when time is passed (i.e., $time\_left < old\_time\_left$). The variable old_time_left is assigned the value of $time\_left$ then, which in turn enables the timer events, which are enabled only when $time\_left = old\_time\_left$.

Together, $time\_left$ and old_time_left are also used to guarantee termination of all the new event operations introduced in this refinement step. To make a refinement step valid, we have to suggest *a variant* – a natural number expression that is provably decreased by execution of new events. The variant expression for this refinement step is $time\_left + old\_time\_left$.

```
EVENT timer
    WHEN
        $grd1 : \neg(in\_data = NIL) \wedge finished = FALSE$
        $grd2 : time\_left = old\_time\_left$
        $grd3 : \{tt | tt \in \mathbb{N}_1 \wedge tt < time\_left\} \neq \varnothing$
    THEN
        $act1 : time\_left :\in \{tt | tt \in \mathbb{N}_1 \wedge tt < time\_left\}$
    END

EVENT timer_out
    WHEN
        $grd1 : \neg(in\_data = NIL) \wedge finished = FALSE$
        $grd2 : time\_left = old\_time\_left$
        $grd3 : \{tt | tt \in \mathbb{N}_1 \wedge tt < time\_left\} = \varnothing$
    THEN
        $act1 : time\_left, resp := 0, ABORT$
        $act2 : finished := TRUE$
    END

   ...

VARIANT
   $time\_left + old\_time\_left$
```

4.5 Handling Responses and Recovery Actions

The main goal of *Service Director* is to co-ordinate the service execution flow by requesting the required subservices from the external service providers. According to the given service execution order (stored in the abstract function $Next$), at each execution step *Service Director* activates certain subservices that are executed in parallel. The currently active subservices are stored in the variable *active*. Then *Service Director* asynchronously reads their responses and evaluates them in terms of possible further actions. The abstract function *Eval* is used for this purpose. As described above, these actions may include continuation of service execution, retrying or cancelling certain active subservices, rollbacking in the execution flow, or aborting the whole service.

In a Event B specification, we distribute these service director activities over a number of separate events. The event **read_response** is responsible for reading responses from the currently active subservices. In addition, we introduce separate events for handling different classes of responses based on the result of the evaluation function *Eval*.

Below we present the event operation **read_response** as well as the events handling (partial) success, cancelling of subservices, rollbacking to a certain point of the execution flow, and aborting the whole service.

```
EVENT read_response
    ANY
        $ss$
        $data$
    WHERE
        $grd3 : ss \in active \wedge data \in DATA \setminus \{NIL\}$
        $grd1 : received = FALSE$
        $grd2 : active \neq \varnothing$
        $grd4 : time\_left < old\_time\_left$
    THEN
        $act1 : curr\_state := update(ss \mapsto curr\_state \mapsto data)$
        $act2 : active := active \setminus \{ss\}$
        $act3 : received := TRUE$
    END
```

The event simply updates the internal state of *Service Director* and then enables the handling events by setting the flag *received*.

```
EVENT handle_SUCCESS
    WHEN
        $grd1 : \neg(in\_data = NIL) \wedge finished = FALSE \wedge received = TRUE$
        $grd2 : Eval(curr\_task \mapsto curr\_state) = SUCCESS$
        $grd3 : curr\_task < max\_next \wedge time\_left < old\_time\_left$
    THEN
        $act1 : results := results \cup \{curr\_task \mapsto curr\_state\}$
        $act2 : curr\_task := curr\_task + 1$
        $act3 : active := Next^{-1}[\{curr\_task + 1\}]$
        $act4 : old\_time\_left := time\_left$
        $act5 : received := FALSE$
        $act6 : resp := SUCCESS$
    END
```

The event is enabled when *Service Director* successfully finishes the current task (i.e., service execution step) possibly involving several parallel subservices. The calculated results are saved in the variable *results*, and then *Service Director* moves to executing of the next task. The subservices to be activated are decided on the basis of the service execution flow stored in $Next$.

The additional event **handle_SUCCESS_complete** (omitted here) complements **handle_SUCCESS** by covering the situations when $curr\_task = max\_next$, i.e., when *Service Director* succesfully finished execution of the whole service (or the part of the service delegated to it). As a result, the flag $finished$ becomes $TRUE$.

```
EVENT handle_CANCEL
    WHEN
        $grd1 : \neg(in\_data = NIL) \wedge finished = FALSE \wedge received = TRUE$
        $grd2 : Eval(curr\_task \mapsto curr\_state) = CANCEL$
        $grd3 : time\_left < old\_time\_left$
    THEN
        $act1 : old\_time\_left := time\_left$
        $act2 : received := FALSE$
        $act3 : resp := CANCEL$
    END
```

The event is enabled when there is necessary to cancel some or all currently active subservices. This might be needed before retrying, rollbacking, or aborting service execution. Note that the set of active services stored in the variable *active* is not updated in this operation. This is because the actual services to be cancelled (determined using the abstract function *Cancel*) are already a subset of the currently active services.

```
EVENT handle_ROLLBACK
    WHEN
        $grd1 : \neg(in\_data = NIL) \wedge finished = FALSE \wedge received = TRUE$
        $grd2 : Eval(curr\_task \mapsto curr\_state) = ROLLBACK$
        $grd5 : time\_left < old\_time\_left$
    THEN
        $act1 : curr\_task := Rollback(curr\_task \mapsto curr\_state)$
        $act2 : results := (1 \,..\, Rollback(curr\_task \mapsto curr\_state) - 1) \lhd results$
        $act3 : active := Next^{-1}[\{Rollback(curr\_task \mapsto curr\_state)\}]$
        $act4 : old\_time\_left := time\_left$
        $act5 : received := FALSE$
        $act6 : resp := ROLLBACK$
    END
```

The event is enabled when, to recover from the current failure, *Service Director* has to rollback to some previous point in the service execution flow. The necessary information is stored in the abstract function *Rollback*. Also, the corresponding stored results from the skipped execution steps are removed. The latter is accomplished by using the relational domain restriction operator $\lhd$, which in this case retains only those elements of *results* that have the indexes belonging to the interval $(1 \,..\, Rollback(curr\_task \mapsto curr\_state) - 1)$.

```
EVENT handle_ABORT
    WHEN
        $grd1 : \neg(in\_data = NIL) \wedge finished = FALSE \wedge received = TRUE$
        $grd2 : Eval(curr\_task \mapsto curr\_state) = ABORT$
        $grd3 : time\_left < old\_time\_left$
    THEN
        $act1 : finished := TRUE$
        $act2 : old\_time\_left := time\_left$
        $act3 : resp := ABORT$
    END
```

The event handles the situation when an unrecoverable failure has occurred and the execution of the whole service has to be aborted. The flags *finished* and *resp* are set accordingly to trigger the corresponding **calculate** and **output** events modelling the abortive termination of service execution.

We proved a number of invariant properties for the specification described above. Some important ones are presented below.

```
INVARIANTS
    ...
    $inv7 : finished = TRUE \wedge \neg(resp = ABORT) \Rightarrow curr\_task = max\_next$
    $inv8 : resp = ABORT \Rightarrow finished = TRUE$
    $inv10 : finished = TRUE \Rightarrow resp \in \{SUCCESS, ABORT\}$
    $inv15 : finished = FALSE \Rightarrow time\_left > 0$
```

The first property states that if service execution is succesfully finished, then the whole service execution flow (given in *Next*) must be completed. The second one requires that generating the *ABORT* flag at any point of service execution must lead to immediate termination of the whole service. The third property states that any service execution should end either by success or abort. Finally, the last one requires that service execution can continue only if the maximal execution time is not exceeded. This is also equivalent to saying that expiring of time will immediately lead to termination of service execution.

In the presented refinement step, the execution of external service providers is modelled implicitly by nondeterministically setting the received response data in the event **read_response**. The next refinement step will explicitly introduce the external service components following the original specification pattern *ACC*. As a result, the activating the corresponding subservices and reading their responses can be modelled by using explicitly introduced communication channels.

If some of these new service components represent service directors then the corresponding refinement step can be applied again to introduce necessary details. Therefore, this refinement step itself can be considered as a refinement pattern that can repeatedly applied in formal development of communicating systems. The result of such development would be hierarchical architecture of service components (including service directors) responsible for providing a particular service.

5 Conclusions and Related Work

In this paper we proposed a formal approach to development of communicating distributed systems. Our approach formalises and extends Lyra [7] – the UML2-based design methodology adopted in Nokia. The formalisation is done within the B Method [1] and its new version EventB [2] – formal frameworks supporting system development by stepwise refinement. The proposed approach establishes a basis for automatic translation of UML2-based development of communicating systems into the refinement process in B. Such automation would enable smooth integration of formal methods into existing development practice.

The initial formalization of Lyra has been undertaken using model checking techniques [7]. However, since telecommunicating systems tend to be large and data intensive, this formalization was prone to the state explosion problem. Our approach helps to overcome this limitation, since it is based on theorem proving, verifying the desired properties for all possible situations allowed by a given model. Moreover, the core of our approach is the stepwise refinement paradigm, which allows to introduce implementation details gradually and thus deal with explosion of complexity.

Development of distributed communicating systems has been a topic of ongoing research over several decades. Our review of related work is confined to the consideration of the recent research conducted within B.

The pioneering work on formal development of distributed systems in Event B was done by Abrial et al. [3]. They demonstrated how to prove termination of a complex distributed protocol in Event B. In our work we use the principles defined in [3] to formalize the service-oriented development of complex communicating systems.

Yadav and Butler [12] used Event B to design fault tolerant transactions for replicated distributed database systems. They demonstrated how to formally verify by refinement that the design of a replicated database confirms to the one copy database abstraction. Similarly, in our work we use refinement to verify that the externally observable behaviour of distributed implementation of a service is equivalent to its centralized abstraction. However, our primary goal was not only formal verification of service development but also integration of modelling and refinement in B into the existing UML2-based development flow.

In this paper we focused on integrating fault tolerance mechanisms into the formalised Lyra development process. One of big challenges is formal modelling of parallel service execution and its effect on system fault tolerance. The ideas presented in this paper are implemented by extending our previously developed B models. The formalised Lyra development has been originally verified by completely proving the corresponding B refinement steps using the Atelier B tool. Later, the formal development has been moved to the RODIN platform supporting the new Event B language developed within the RODIN project [9].

The newly developed Event B tools (i.e. the RODIN platform and its extensions) have been very helpful verifying the formal development. In particular, for the presented refinement step introducing *Service Director*, 131 proof obligations were generated. Out of them, 119 (91%) were proved automatically. The remaining 12 proof obligations were proved using the interactive prover, with a few simple hints to the prover being sufficient to finish the proof. This allows us to be optimistic that, after some small tuning, the verification process could become fully automatic.

Acknowledgements

This work is supported by IST FP6 RODIN Project.

We also would like to thank the anonymous reviewers for their very helpful comments.

References

1. Abrial, J.-R.: The B-Book. Cambridge University Press, Cambridge (1996)
2. Abrial, J.-R.: Extending B without Changing it (for Developing Distributed Systems). In: Proceedings of 1st Conference on the B Method, Nantes, France, pp. 169–191. Springer, Heidelberg (1996)
3. Abrial, J.-R., Cansell, D., Mery, D.: A mechanically proved and Incremental development of IEEE 1394 Tree Identity Protocol. Formal Aspects of Computing 14, 215–227 (2003)
4. Clearsy. AtelierB: User and Reference Manuals, `http://www.atelierb.societe.com/index_uk.html`
5. Laibinis, L., Troubitsyna, E., Leppänen, S., Lilius, J., Malik, Q.A.: Formal service-oriented development of fault tolerant communicating systems. In: Butler, M., Jones, C.B., Romanovsky, A., Troubitsyna, E. (eds.) Rigorous Development of Complex Fault-Tolerant Systems. LNCS, vol. 4157, pp. 261–287. Springer, Heidelberg (2006)
6. Laibinis, L., Troubitsyna, E., Leppänen, S., Lilius, J., Malik, Q.: Formal model-driven development of communicating systems. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 188–203. Springer, Heidelberg (2005)
7. Leppänen, S., Turunen, M., Oliver, I.: Application Driven Methodology for Development of Communicating Systems. In: Forum on Specification and Design Languages, Lille, France (2004)
8. Rigorous Open Development Environment for Complex Systems (RODIN). Deliverable D7, Event B Language, `http://rodin.cs.ncl.ac.uk/`
9. Rigorous Open Development Environment for Complex Systems (RODIN). IST FP6 STREP project, `http://rodin.cs.ncl.ac.uk/`

10. The RODIN platform, `http://rodin-b-sharp.sourceforge.net/`
11. Treharne, H., Schneider, S., Bramble, M.: Composing specifications using communication. In: Bert, D., Bowen, J.P., King, S. (eds.) ZB 2003. LNCS, vol. 2651, pp. 58–78. Springer, Heidelberg (2003)
12. Yadav, D., Butler, M.: Application of Event B to Global Causal Ordering for Fault Tolerant Transactions. In: Proceedings of Workshop on Rigorous Engineering of Fault Tolerant Systems (REFT 2005), Newcastle upon Tyne, UK, pp.93–102 (July 2005)

Formal Development of a Total Order Broadcast for Distributed Transactions Using Event-B

Divakar Yadav* and Michael Butler**

School of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ, U.K.
divakar.yadav@ietlucknow.edu, mjb@ecs.soton.ac.uk

Abstract. In a replicated database system, copies of the database are kept across several sites for fault-tolerance and availability. Data access in such systems is usually done within a transactional framework. A read-only transaction accesses data locally and an update transaction modifies the database at all sites. Total order broadcast primitives have been proposed to support transactions and allow fault-tolerant cooperation between the sites in a distributed system. In this paper, we identify and analyze the problem of formation of deadlocks among conflicting update transactions due to race conditions and outline how a system of total order broadcast prevents deadlocks and transaction failures. Later we outline how a refinement based approach with Event-B can be used for formal development of the models of total order broadcast. In this approach we begin with the abstract model of a total order broadcast and verify that the required ordering properties are preserved by the system. Subsequently, in a series of refinement steps we outline how an abstract total order can correctly be implemented by using a notion of sequence number. This technique requires us to discharge proof obligations due to consistency and refinement checking. To discharge the proof obligations we are required to discover invariants that describes the relationship between the abstract total order and the underlying mechanism.

1 Introduction

A replicated database system can be defined as a distributed system where copies of the database are kept across several sites. Data access in a replicated database can be done within a transactional framework. A distributed transaction may span several sites reading or updating data objects. It is advantageous

[*] Currently working at Institute of Engineering and Technology, U P Technical University, Lucknow, India. This work was supported by the Commonwealth Scholarship Commission in the United Kingdom.

[**] Michael Butler's contribution is part of EU projects IST project IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems) and ICT 214158 DEPLOY (Industrial deployment of system engineering methods providing high dependability and productivity www.deploy-project.eu/).

M. Butler et al. (Eds.): Fault Tolerance, LNCS 5454, pp. 152–176, 2009.
© Springer-Verlag Berlin Heidelberg 2009

to replicate the data if the transaction workload is predominantly read only. However, during updates, the complexity of keeping the replicas in a consistent state arises due to race conditions among conflicting update transactions. A typical distributed transaction contains a sequence of database operations which must be processed at all of the participating sites or none of the sites to maintain the integrity of the database. The strong consistency criterion in the replicated database requires that the database remains in a consistent state despite transaction failures. In addition to providing fault-tolerance, one of the important issues to be addressed in the design of replica control protocols is consistency. The *one copy equivalence* [9] criteria requires that a replicated database is in a mutually consistent state only if all copies of data objects *logically* have the same identical value.

No common global clock or shared memory exist in a distributed system. The sites communicate by exchange of messages which are delivered to them after arbitrary time delays. In such systems up-to-date knowledge of the system is not known to any process or site. This problem can be dealt by relying on group communication primitives that provide ordering guarantees on the delivery of messages. The group communication primitives have been proposed as a mechanism for the development of reliable fault-tolerant distributed applications [16]. A *total order broadcast* is one such primitive that guarantees the delivery of messages to the sites in the same order. Introduction of the transactions based on group communication primitives represents an important step towards extending the power of group communication in an asynchronous distributed system [34]. These primitives have been proposed for processing transactions and managing replicated databases [21,35,22]. In a replicated database that uses a reliable broadcast without ordering guarantees, the operations of the conflicting update transactions may arrive at different sites in different orders. This may lead to the formation of deadlock among conflicting transactions involving several sites. The blocking of the transactions at a site is usually resolved through aborting the transaction by timeouts. The abortion of conflicting transactions can be avoided by using a total order broadcast which delivers and executes the conflicting operations at all sites in the same order.

In this paper we present an incremental development of a model of total order broadcast using Event-B [28], which is a variant of B Method [1]. Event-B is a formal technique for the development of models of distributed systems. This technique consists of describing rigorously the problem in an abstract model, introducing solutions or design details in refinement steps to obtain more concrete specifications, and verifying that the proposed solutions are correct. The B tools provide a significant automated proof support for generating the proof obligations and discharging them. This technique requires the discharge of proof obligations for consistency checking and refinement checking. The technique is supported by several industrial level B tools such as, Rodin [3] and Click'n'Prove [4] that provide a significant automated proof support for generation of the proof obligations, factorizing complex proof obligations into simpler proofs and discharging them. The majority of the proof obligations are proved by the automatic prover of the tools. However, some complex proof obligations require user guidance through the interactive prover. These proof obligations also help in the discovery of new system invariants. The proof obligations and the invariants help to understand the complexity of the problem and the correctness of the solutions. They also provide a clear insight into the system and enhance our understanding of why a design decision should work. The essential features of the modelling and proof guidelines to obtain an high degree of automated proof for an Event-B development are outlined in [15]. We have used the Click'n'Prove [4] B tool for proof obligation generation and to discharge them.

The remainder of this paper is organized as follows: Section 2 outlines the system model, Section 3 identifies the problem of formation of deadlocks among the transactions due to unordered delivery of update messages, Section 4 describes informal specifications of a total order broadcast and mechanism for implementation, Section 5 outlines the abstract model of total order broadcast, and shows how an abstract total order is constructed on the messages. Section 6 present the invariant properties of the system. We also outline how the proof obligations generated by the B tool help us discover new invariants. Section 7 illustrates essential features of the refinement chain. Section 8 present related work on group communication and the application of formal methods to the problem. Finally, Section 9 concludes the paper.

2 Background

We have presented a rigorous design of distributed transactions for a replicated database using Event-B in [38]. Our system model consist of a sets of sites and data objects. Users interact with the database by *starting transactions.* We consider the case of full replication and assume all data objects are updateable. The *Read Anywhere Write Everywhere* [9,30] replica control mechanism is considered for updating replicas. In our model, update transactions are processed within the framework of a two phase commit protocol [19] to ensure global atomicity.

2.1 Transaction Model

A transaction is considered as a sequence of read/write operations executed atomically, i.e., a transaction will either *commit* or *abort* the effect of all database operations. The following types of transactions are considered for this model of replicated database.

- *Read-Only Transactions* : These transactions are submitted locally to the site and *commit* after reading the requested data object locally.
- *Update Transactions* : These transactions update the requested data objects. The effect of update transactions are global, thus when committed, replicas of data objects maintained at all sites must be updated. In the case of abort, none of the sites update the data object.

Let the sequence of read/write operations issued by the transaction T_i be defined by a set of objects $objectset[T_i]$ where $objectset[T_i] \neq \varnothing$. Let the set $writeset[T_i]$ represents the set of object to be *updated* such that $writeset[T_i] \subseteq objectset[T_i]$. A transaction T_i is a read-only transaction if $writeset[T_i]= \varnothing$. Similarly a transaction T_i is an update transaction if its $writeset[T_i] \neq \varnothing$.

2.2 Conflicting Transactions

Two update transactions T_i and T_j are in *conflict* if the sequence of operations issued by T_i and T_j are defined on set of object $objectset[T_i]$ and $objectset[T_j]$ respectively and $objectset[T_i] \cap objectset[T_j] \neq \varnothing$. To meet the strong consistency requirements, *conflicting* transactions need to be executed in isolation. We ensure this property by not *starting* a transaction at a site if any conflicting update transaction is *active* at that site. In our model the transactions are executed as follows.

- A read-only transaction T_i is executed locally at the initiating site of T_i(also called the coordinator site of T_i) by acquiring locks on the data object defined by $objectset[T_i]$.
- A global update transaction T_i is executed by broadcasting an update message to the participating sites. On delivery, a participating site S_j initiates a sub-transaction T_{ij} by acquiring locks on $objectset[T_i]$. If the objects are currently locked by another transaction, T_{ij} is blocked. The activity of a global update transaction at a given site is referred as sub-transaction.
- The coordinator site of T_i waits for the vote commit/abort messages from all participating site. A global commit/abort message is broadcast by the coordinator site of T_i only if it receives all local commit message from all participating sites or at-least one vote-abort message from participating sites.

The commit or abort decision of a global transaction T_i is taken at the coordinator site within the framework of a two phase commit protocol as shown in Fig. 1 as follows. A global transaction T_i *commits* if *all* T_{ij} *commit* at S_j. The global transaction T_i *aborts* if *some* T_{ij} *aborts* at S_j.

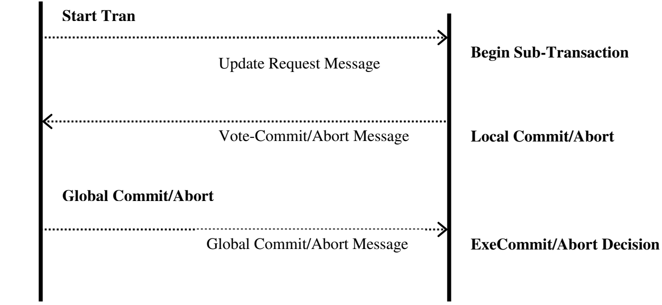


Fig. 1. Events of Update Transaction

3 Blocking and Failures of Conflicting Transactions

This section outlines how conflicting update transactions in our model can be deadlocked. A formal refinement based approach using Event-B to model and analyze distributed transaction is given in [38]. In our abstract model, an update transaction modifies the abstract one copy database through a single atomic event. In the refinement, an update transaction consists of a collection of interleaved events updating each replica separately. The transaction mechanism on the replicated database is designed to provide the illusion of atomic update of a one copy database. Through the refinement proofs, we verify that the design of the replicated database conforms to the one copy database abstraction despite transaction failures at a site. The global atomicity of update transactions is ensured by processing update transactions within the framework of two phase commit protocol. We assume that the sites communicate by a reliable broadcast which eventually deliver messages without any ordering guarantees.

In the abstraction, the global state of update transactions is represented by a variable *transstatus* in the abstract model of the transactions. The variable *transtatus* is defined as $transtatus \in trans \rightarrow TRANSSTATUS$, where $TRANSSTATUS = \{COMMIT, ABORT, PENDING\}$. The *transstatus* maps each transaction to its global state. An update transaction commits by updating abstract variable *database*. With respect to an update transaction, activation of the following events change the global transaction states.

- *StartTran(tt)* : The activation of this event *starts* a fresh transaction and the state of the transaction is set to *pending*.
- *CommitWriteTran(tt)* : This event models global commit of an update transaction. A *pending* update transaction commits atomically by updating the abstract database and it status is set to *commit*.
- *AbortWriteTran(tt)* : This event models global abort of an update transaction. A *pending* update transaction aborts by making no change in the abstract database and its status is set to *abort*.

In the refined model, a global update transaction can be submitted to any one site, called the coordinator site for that transaction. Upon submission of an update transaction, the coordinating site of the transaction broadcasts all operations of the transaction to the participating sites by an *update* message. Upon receiving the update message at a participating site, the transaction manager at that site starts a sub-transaction. The activity of a global update transaction at a given site is referred as a sub-transaction. The *BeginSubTran(tt,ss)* event models starting a sub-transaction of *tt* at participating site *ss*. The specifications of this event is given in the Fig. 2. In this refinement, the state of a transaction at a site is represented by a variable *sitetransstatus*. The variable *sitetransstatus* maps each transaction, at a site, to transaction states given by a set *SITETRANSTATUS*, where *SITETRANSTATUS*={*pending*, *commit*, *abort*, *precommit*}. A transaction t is said to be active at a site s if it has acquired the locks of the object set at that site.

```
BeginSubTran ( tt∈ TRANSACTION ,ss∈ SITE)≅
    WHEN   ∧  tt ∈ trans
           ∧  (ss↦ tt)∉ activetrans
           ∧  ss ∉ dom(sitetransstatus)
           ∧  ran(transeffect(tt))≠{∅}
           ∧  objectset(tt) ⊆ freeobject[{ss}]
           ∧  transstatus(tt)=PENDING
           ∧  ∀tz.(tz ∈ trans ∧ (ss ↦ tz)∈ activetrans
                        ⇒ objectset(tt) ∩ objectset(tz) = ∅)
    THEN       activetrans := activetrans ∪ {ss↦ tt}
           ||  sitetransstatus(tt)(ss) := pending
           ||  freeobject := freeobject - {ss} × objectset(tt)
    END;
```

Fig. 2. Sub Transaction

Our model prevents starting sub-transaction at a site if any conflicting transaction is already active at that site. Following guard of *BeginSubTran(tt)* event ensures that a sub-transaction of *tt* is started at site *ss* when no active transaction *tz* running at *ss* is in *conflict* with *tt* :

$$(ss \mapsto tz) \in activetrans \Rightarrow objectset(tt) \cap objectset(tz) = \varnothing$$

The guard $ss \notin dom(sitetransstatus(tt))$ prevents starting a sub-transaction again at the site *ss*. As a consequence of the occurrence of this event, transaction *tt* becomes *active* at site *ss* and the *sitetransstatus* of *tt* at *ss* is set to pending. The guard $ran(transeffect(tt)) \neq \{\varnothing\}$ states that *tt* is an update transaction, i.e., $writeset(tt) \neq \varnothing$. Instead of giving the specifications of all events of the refinement in the similar detail, brief descriptions of the new events in this refinement are outlined below.

- *BeginSubTran(tt)* : This event models *starting* a sub-transaction at a site. The status of the transaction *tt* at site *ss* is set to *pending.*
- *SiteAbortTx(ss,tt)* : This event models *local abort* of a transaction at a site. The transaction is said to complete execution at the site. The status of the transaction *tt* at site *ss* is set to *abort.*
- *SiteCommitTx(ss,tt)* : This event models *precommit* of a transaction at a site. The status of the transaction *tt* at site *ss* is set to *precommit.*
- *ExeAbortDecision(ss,tt)* : This event models *abort* of a *precommitted* transaction at a site. This event is activated once the transaction has globally aborted. The status of the transaction *tt* at site *ss* is set to *abort.* The transaction is said to complete execution at the site.
- *ExeCommitDecision(ss,tt)* : This event models *commit* of a *precommitted* transaction at a site. This event is activated once the transaction has globally committed. The status of the transaction *tt* at site *ss* is set to *precommit.* The replica at the site is updated with the transaction effects and the transaction is said to complete execution at this site.

In our model, update messages from the coordinator site are broadcast using a reliable broadcast. A reliable broadcast imposes no restriction on the order in which messages are delivered to the participating sites. This may lead to the formation of the deadlocks due to race conditions and the sites may abort one or more of the conflicting transaction by timeouts. For example, consider two conflicting update transactions T_i and T_j initiated at site S_i and S_j respectively. Both of the transactions may be blocked in the following scenario :

- S_i starts transaction T_i and acquire locks on $objectset[T_i]$ at site S_i. Site S_i broadcast update message of T_i to participating sites. Similarly, another site S_j starts a transaction T_j , acquires locks on $objectset[T_j]$ at site S_j and broadcast update message of T_j to participating sites.
- The site S_i delivers update message of T_j from S_j and S_j delivers update message of T_i from S_i. The T_j is blocked at S_i as S_i waits for vote-commit from S_j for T_i. Similarly, T_i is blocked at S_j waiting for vote-commit from S_i for T_j

In order to recover from the above scenario where two conflicting transactions are blocked, either or both transactions may be aborted by the sites. The abort of these conflicting update transactions may be avoided if a reliable broadcast also provides ordering guarantees on the message delivery such that all update messages are delivered to various participating site including the sender in a total order. In the remaining sections we formally model and analyze a system of total order broadcast and verify that the required ordering properties are satisfied.

4 Informal Specifications of a Total Order Broadcast

A reliable broadcast [20] eventually deliver the messages to all participating sites. A *total order* [16,20] broadcast is a stronger notion of a reliable broadcast that delivers messages to all processes in a same delivery order. A *total order broadcast*[1] can be defined as a reliable broadcast which satisfies following requirement.

> *If processes p and q both deliver messages m1 and m2, then q delivers m1 before m2 if and only if p delivers m1 before m2.*

The *agreement* property of a reliable broadcast and *total order* requirements imply that all correct processes eventually deliver the same *sequence* of messages [20]. As shown in the Fig. 3 all processes has same delivery order of messages. However, as shown in Fig. 4 the delivery order violates total order requirement as delivery order at process *P1* and *P2* are different.

[1] The *Total Order Broadcast* is also known as Atomic Broadcast. Both of the terms are used interchangeably. However we prefer the former as the term *atomic* suggests the *agreement* [20] property rather than *total order*.

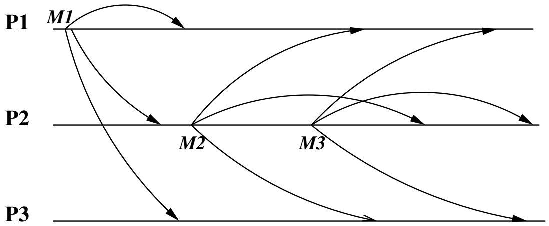


Fig. 3. Total Order

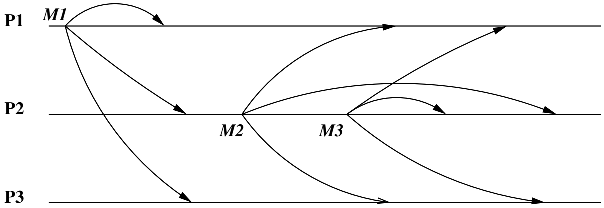


Fig. 4. Violation of Total Order

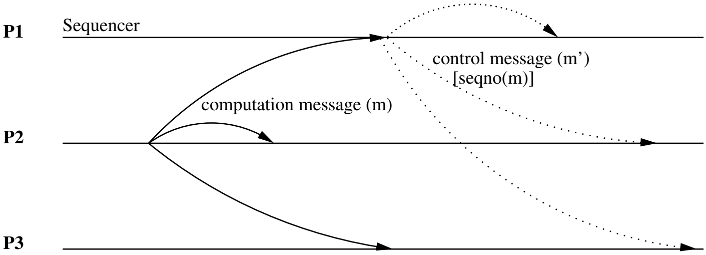


Fig. 5. Broadcast Broadcast variant

Mechanism for Total Order Implementations: The key issues with respect to the total order broadcast algorithms are how to build a total order and what information is necessary for defining a total order. In our development we consider *Broadcast Broadcast*(*BB*) [16] variant of a sequencer based system. In sequencer based system, a specific process takes the role of a *sequencer* and becomes responsible for building a total order. The protocol consists of first broadcasting m to all destinations including the sequencer, followed by an another broadcast of its sequence number by the sequencer. All destination processes deliver messages according to their sequence numbers assigned by the sequencer process. As shown in the Fig. 5 process *P2* broadcast a *computation message m*. Upon delivery of m to a *sequencer* process, sequencer assigns a sequence number and broadcast its sequence number by a *control message*(*m'*). Upon receipt of the control messages, a destination process deliver its computation message according to the sequence numbers.

MACHINE *TotalOrder*
SETS *PROCESS*; *MESSAGE*
VARIABLES *sender, totalorder, delorder, tdeliver*
INVARIANT $sender \in MESSAGE \nrightarrow PROCESS \wedge$
$totalorder \in MESSAGE \leftrightarrow MESSAGE \wedge$
$delorder \in PROCESS \nrightarrow (MESSAGE \leftrightarrow MESSAGE) \wedge$
$tdeliver \in PROCESS \leftrightarrow MESSAGE$
INITIALISATION $sender := \varnothing \quad \| \quad totalorder := \varnothing \quad \|$
$delorder := PROCESS \times \{\varnothing\} \quad \| \quad tdeliver := \varnothing$

Fig. 6. Initial Part : Level-0

5 Abstract Model of Total Order Broadcast

The abstract model of total order broadcast system is given in Fig. 6 and Fig. 7. The *PROCESS* and *MESSAGE* sets define types for the model. The specification contains of four variables *sender*, *totalorder*, *tdeliver* and *delorder*.

The *sender* is defined as a partial function from *MESSAGE* to *PROCESS*. The mapping $(m \mapsto p) \in sender$ indicates that message m was sent by a process p. The variable *totalorder* is defined as a relation among the messages. A mapping of the form $(m1 \mapsto m2) \in totalorder$ indicate that message *m1* is *totally ordered before m2*. The variable *tdeliver* represent the messages delivered following a total order. A mapping of form $(p \mapsto m) \in tdeliver$ represents that a process p has delivered m following a *total order*. In order to represent the delivery order of messages at a process, variable *delorder* is used. A mapping $(m1 \mapsto m2) \in$ *delorder(p)* indicate that process p has delivered *m1* before *m2*.

The event *Broadcast* given in the Fig. 7 models the broadcast of a message. Similarly, the event *Order* models the construction of total order on a message when it is delivered to a process in the system for the first time, i.e., an abstract *global total order* is constructed on a message at the *first ever delivery* of it to any process in the system. Later in the refinement we show that it is a role of a sequencer process. The *TODeliver* models the delivery of the messages to a process when a total order on the message has been constructed.

5.1 Constructing a Total Order

The event *Order* models the construction of an abstract total order on message *mm* its *first* ever delivery to a process *pp*. The following guards of this event ensures that the message *mm* has not been delivered elsewhere and that each message delivered at any other process has also been delivered to this process(pp).

$$mm \notin ran(tdeliver)$$
$$ran(tdeliver) \subseteq tdeliver[\{pp\}]$$

Later in the refinement we show that this is a function of a designated process called *sequencer*. As a consequence of the occurrence of *Order* event, the

```
Broadcast $(pp \in PROCESS , mm \in MESSAGE ) \cong$
    WHEN    $mm \notin dom(sender)$
    THEN    $sender := sender \cup \{mm \mapsto pp\}$
    END;
Order $(pp \in PROCESS ,mm \in MESSAGE ) \cong$
    WHEN    $mm \in dom(sender) \wedge$
            $mm \notin ran(tdeliver) \wedge$
            $ran(tdeliver) \subseteq tdeliver[\{pp\}]$
    THEN    $tdeliver := tdeliver \cup \{pp \mapsto mm\} \parallel$
            $totalorder := totalorder \cup ( ran(tdeliver) \times \{mm\}) \parallel$
            $delorder(pp) := delorder(pp) \cup ( tdeliver[\{pp\}] \times \{mm\})$
    END;
TODeliver $(pp \in PROCESS , mm \in MESSAGE) \cong$
    WHEN    $mm \in dom(sender) \wedge$
            $mm \in ran ( tdeliver ) \wedge$
            $pp \mapsto mm \notin tdeliver \wedge$
            $\forall m.( m \in MESSAGE \wedge (m \mapsto mm) \in totalorder$
                                    $\Rightarrow (pp \mapsto m) \in tdeliver)$
    THEN    $tdeliver := tdeliver \cup \{pp \mapsto mm\} \parallel$
            $delorder(pp) := delorder(pp) \cup ( tdeliver[\{pp\}] \times \{mm\})$
    END
```

Fig. 7. Events : Level-0

message *mm* is delivered to the process *pp* and variable *totalorder* is updated by mappings in $(ran(tdeliver) \times mm)$. This indicates that all messages delivered at any process in the system are *ordered* before *mm*. Similarly, the delivery order at the process is also updated such that all messages delivered at any process precedes *mm*. It can be noticed that the total order for a message is built when it is delivered to a process for the *first* time.

The event *TODeliver*(*pp*,*mm*) models the delivery of a message *mm* to a process *pp* respecting the *total order*. As the guard $mm \in ran(tdeliver)$ implies that the *mm* has been delivered to at least one process and it also implies that the total order on the message *mm* has also been constructed. Later in the refinement we show that process *pp* represents a process other than the *sequencer* process. The guard of the event ensure that message *mm* has already been delivered elsewhere and that all messages which precedes *mm* in abstract total order has also been delivered to *pp*.

5.2 Invariant Properties of Total Order

After building an abstract model of a total order broadcast(Level-0), our goal was to formally verify that our model preserves the total ordering properties defined in the Section 4. The agreement and total order requirement imply that all correct process eventually deliver all messages in the same order [20]. Thus, we add following invariant as a primary invariant to our model.

$$m1 \mapsto m2 \in delorder(p) \Rightarrow m1 \mapsto m2 \in totalorder \tag{1}$$

This invariant at (1) state that if a process delivers any two messages then their delivery order at that process corresponds to their abstract total order. Subsequently, in order to prove that total order preserves the transitivity property, we add following as a primary invariant to our model.

$$\begin{aligned} &m1 \mapsto m2 \in totalorder \wedge m2 \mapsto m3 \in totalorder \\ &\Rightarrow\ m1 \mapsto m2 \in totalorder \end{aligned} \tag{2}$$

Lastly, to verify that the abstract total order is non-symmetric and non-reflexive, we add following invariant :

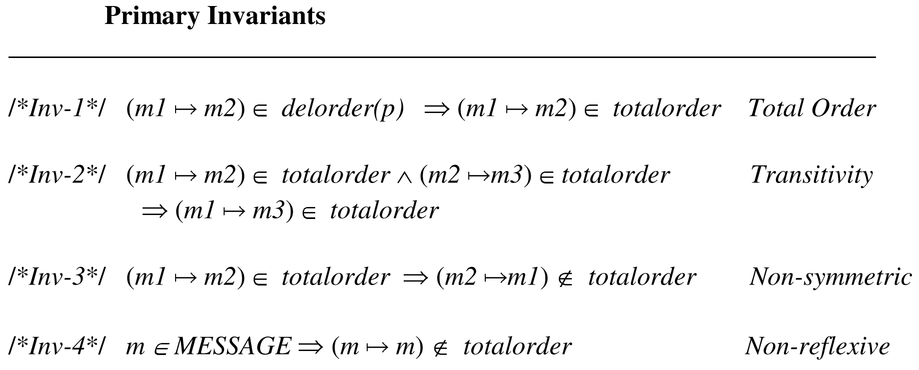
$$m1 \mapsto m2 \in totalorder \Rightarrow m2 \mapsto m1 \notin totalorder \tag{3}$$

$$m \in MESSAGE \Rightarrow m \mapsto m \notin totalorder \tag{4}$$

6 Proof Obligations and Invariant Discovery

In this section, we outline how the proof obligations generated due to the addition of the primary invariants given at (1), (2), (3) and (4) in Fig. 8 guide us discovering new invariants.

Verification of Total Ordering Property: In order to verify that our abstract model of total order broadcast satisfies the total order property, we add *Inv-1* given in Fig. 8 to our model. When we add this invariant to our model two proof obligations were generated associated with the event *Order* and *TODeliver*.



Primary Invariants

*/*Inv-1*/*	$(m1 \mapsto m2) \in delorder(p) \Rightarrow (m1 \mapsto m2) \in totalorder$	*Total Order*
*/*Inv-2*/*	$(m1 \mapsto m2) \in totalorder \wedge (m2 \mapsto m3) \in totalorder \Rightarrow (m1 \mapsto m3) \in totalorder$	*Transitivity*
*/*Inv-3*/*	$(m1 \mapsto m2) \in totalorder \Rightarrow (m2 \mapsto m1) \notin totalorder$	*Non-symmetric*
*/*Inv-4*/*	$m \in MESSAGE \Rightarrow (m \mapsto m) \notin totalorder$	*Non-reflexive*

Fig. 8. Primary Invariants-I : Level-0

Proof obligation associated with the event *Order* was discharged using interactive prover, however the proof obligation associated with *TODeliver* could not be discharged. Following is the simplified form of a proof obligation generated by the interactive prover.

$$\begin{array}{l} \mathit{TODeliver}(\mathit{PO}1) \\ \left[\begin{array}{l} p \mapsto m1 \in \mathit{tdeliver}\ \wedge \\ p \mapsto m2 \notin \mathit{tdeliver}\ \wedge \\ m2 \in \mathit{ran}(\mathit{tdeliver})\ \wedge \\ \Rightarrow\ m1 \mapsto m2 \in \mathit{totalorder} \end{array}\right] \end{array}$$

This state that if process *p* has delivered *m1* but *m2* has been delivered elsewhere then *m1* precedes *m2* in total order. In order to discharge this proof obligation, we add an invariant to our model given as *Inv-5* in Fig. 9. Addition of *Inv-5* was sufficient to discharge *PO1*, however a new proof obligation associated with *TODeliver* was generated due to the addition of *Inv-5*. Following is the simplified form of the proof obligation.

$$\begin{array}{l} \mathit{TODeliver}(\mathit{PO}2) \\ \left[\begin{array}{l} m1 \in \mathit{ran}(\mathit{tdeliver})\ \wedge \\ m2 \in \mathit{ran}(\mathit{tdeliver})\ \wedge \\ m2 \mapsto m1 \notin \mathit{totalorder}\ \wedge \\ \Rightarrow\ m1 \mapsto m2 \in \mathit{totalorder} \end{array}\right] \end{array}$$

This proof obligation require us to prove that if two messages *m1* and *m2* are delivered to any process(es) in the system then a total order exists among them, i.e., either *m1* precedes *m2* or *m2* precedes *m1* in abstract total order. In order to discharge the proof obligation we add another invariant *Inv-6* to our model. Addition of this invariant to the model further generate proof obligations.

After four round of invariant strengthening we arrive at a set of invariant given in Fig. 9 which were sufficient to discharge all proof obligations generated due to addition of invariant *Inv-1* is a primary invariant. A brief description of the properties is given below.

- If a process *p* has delivered *m1* and but not *m2*, and if *m2* was delivered to at least one process elsewhere in the system then *m1* precedes *m2* in total order(*Inv-5*).
- If two messages *m1* and *m2* has been delivered anywhere in the system then a total order exist among them, such that, either *m1* precedes *m2* or *m2* precedes *m1* in total order. (*Inv-6*)
- If a process *p* has delivered two message *m1* and *m2* then either *m1* precedes *m2* or *m2* precedes *m1* in totalorder(*Inv-7*).
- Given two processes *p1* and *p2*, then for any two messages *m1* and *m2* if the process *p2* has delivered both messages and *p1* has delivered *m1* but not *m2* then *m1* precedes *m2* in total order(*Inv-8*).

	Invariants	Required By
/*Inv-5*/	$(p \mapsto m1) \in tdeliver \land (p \mapsto m2) \notin tdeliver$ $\land\ m2 \in \mathrm{ran}(tdeliver)$ $\Rightarrow (m1 \mapsto m2) \in totalorder$	*TOdeliver*
/*Inv-6*/	$m1 \in ran(tdeliver) \land m2 \in ran(tdeliver)$ $\land\ (m2 \mapsto m1) \notin totalorder$ $\Rightarrow (m1 \mapsto m2) \in totalorder$	*Order, TOdeliver*
/*Inv-7*/	$(p \mapsto m1) \in tdeliver \land (p \mapsto m2) \in tdeliver$ $\land\ (m2 \mapsto m1) \notin totalorder$ $\Rightarrow (m1 \mapsto m2) \in totalorder$	*Order, TOdeliver*
/*Inv-8 */	$(p1 \mapsto m1) \in tdeliver \land (p1 \mapsto m2) \notin tdeliver$ $\land\ (p2 \mapsto m1) \in tdeliver \land (p2 \mapsto m2) \in tdeliver$ $\Rightarrow (m1 \mapsto m2) \in totalorder$	*Order, TOdeliver*

Fig. 9. Invariants-II : Level-0

	Invariants	Required By
/*Inv-9 */	$(m1 \mapsto m2) \in totalorder \land (p \mapsto m2) \in tdeliver$ $\Rightarrow (p \mapsto m1) \in tdeliver$	*Broadcast,Order* *TOdeliver*
/*Inv-10 */	$m \in (\ dom\ (totalorder) \cup ran(totalorder)\ )$ $\Rightarrow m \in ran(tdeliver)$	*Order*
/*Inv-11 */	$m \notin dom(\mathrm{sender}) \Rightarrow m \notin dom(totalorder)$ $m \notin dom(\mathrm{sender}) \Rightarrow m \notin ran(totalorder)$ $ran(tdeliver) \subseteq \mathrm{dom(sender)}$	*Broadcast, Order* *TOdeliver*

Fig. 10. Invariants-III : Level-0

Verification of Transitivity Property: Our next step was to verify that our model of total order broadcast also preserves transitive properties on abstract total order. In order to verify that *total order* is transitive, we add *Inv-2* given in Fig. 8 to our model. Addition of this invariant generate several proof obligations. Using the same strategy of invariants strengthening outlined in previous section, we arrive at a set of invariant that is sufficient to discharge all proof obligations generated due the addition of *Inv-2* as a primary invariants. A full set of invariant are given in the Fig. 10. A brief description of these properties are outlined below.

- For any two messages *m1* and *m2* where *m1* is *totally ordered before m2* then a process *p* who delivered *m2* has also delivered *m1*(*Inv-9*).

– The total order is built for those messages which has been delivered to at least one process(*Inv-10*).
– A total order can not be build for the messages which were not sent and each message delivered at any process must be a sent message (*Inv-11*).

Verification of Non-Symmetric and Non-Reflexive Property: In order to prove the *non-symmetric* and *non-reflexive* property on total order we add primary invariants *Inv-3* and *Inv-4* given in Fig. 8 to our model. Using process outlined in the previous section, we are able to discharge the proof obligations generated due to addition of these primary invariants without having to add a new invariant.

7 Overview of the Refinement Chain

In the previous sections we outlined abstract model of a total order broadcast and the invariant properties of abstract total order. In this section we present a overview of our refinement chain consisting of six levels. A brief outline of each refinement step is given below.

L0 This consist of abstract model of total order broadcast. In this model, abstract total order is constructed when a message is delivered to a process for the first time. At all other processes a message is delivered in the total order. We have already outlined this level in Section 5.
L1 This is a refinement of abstract model which introduces the notion of the *sequencer*. In this refinement we outline how a total order on the messages are constructed by the *sequencer*.
L2 This is a very simple refinement giving more concrete specification of *Order* event. Through this refinement we illustrate that a total order can be built using the messages delivered to the sequencer rather than all sites.
L3 In this refinement we introduce the notion of *computation* messages and *sequence numbers*. Global sequence number of the computation messages are generated by the sequencer. The delivery of the messages is done based on the sequence numbers.
L4 In this refinement we introduce notion of *control* messages. We also introduce the relationship of each *computation* message with the *control* messages.
L5 A new event *Receive Control* is introduced. We illustrate that a process other than sequencer can deliver a *computation* message only if it has received *control* message for it.

7.1 Introducing the Notion of the Sequencer : Level-1

In the first refinement, given in Fig. 11, we introduce the notion of a sequencer. The sequencer is defined as a constant for this model as $sequencer \in PROCESS$.

As shown in the refined specification of *Order* event given in Fig. 11, a message is first delivered to the sequencer process. It can be noticed that the the following guards in the abstract specification

$$mm \notin ran(tdeliver)$$
$$ran(tdeliver) \subseteq tdeliver[\{pp\}]$$

are replaced by following.

$$pp = sequencer$$
$$(sequencer \mapsto mm) \notin tdeliver$$

Due to the guard $pp \neq sequencer$ shown in the specifications of *TODeliver*, a message mm is delivered to a process other than the sequencer. The replacement of the guards in the *Order* event generate new proof obligations. Using the same approach of invariant discovery as outlined in Section 5.2, we arrived at a set of invariants that was sufficient to discharge all proof obligations. These invariants are given in Fig. 12. A brief description of these invariants are given in the following steps.

- A message not delivered to the sequencer have not been delivered elsewhere. (*Inv-12*)
- If a total order on any message m has been constructed then it must have been delivered to the sequencer.(*Inv-13,14*)

```
Order ($pp \in PROCESS, mm \in MESSAGE$) $\widehat{=}$
  WHEN    $pp = sequencer \wedge$
          $mm \in dom(sender) \wedge$
          $(sequencer \mapsto mm) \notin tdeliver$
  THEN    $tdeliver := tdeliver \cup \{pp \mapsto mm\}$ ||
          $totalorder := totalorder \cup (ran(tdeliver) \times \{mm\})$
  END;
TODeliver ($pp \in PROCESS, mm \in MESSAGE$) $\widehat{=}$
  WHEN    $pp \neq sequencer \wedge$
          $mm \in dom(sender) \wedge$
          $mm \in ran(tdeliver) \wedge$
          $pp \mapsto mm \notin tdeliver \wedge$
          $\forall m.(m \in MESSAGE \wedge (m \mapsto mm) \in totalorder$
                          $\Rightarrow (pp \mapsto m) \in tdeliver)$
  THEN    $tdeliver := tdeliver \cup \{pp \mapsto mm\}$
  END
```

Fig. 11. Total Order Broadcast : Level-1

	Invariants	Required By
/*Inv-12*/	$(sequencer \mapsto m) \notin tdeliver \Rightarrow m \notin ran(tdeliver)$	Order,TOdeliver
/*Inv-13*/	$m \in dom(totalorder) \Rightarrow (sequencer \mapsto m) \in tdeliver$	Order
/*Inv-14*/	$m \in ran(totalorder) \Rightarrow (sequencer \mapsto m) \in tdeliver$	Order

Fig. 12. Invariants-IV : Level-1

```
Order ($pp \in PROCESS$ ,$mm \in MESSAGE$ ) $\widehat{=}$
    WHEN   $pp = sequencer \ \wedge$
           $mm \in dom(sender) \ \wedge$
           $(sequencer \mapsto mm) \notin tdeliver$
    THEN   $tdeliver := tdeliver \cup \{pp \mapsto mm\}$   ||
           $totalorder := totalorder \cup ( tdeliver[\{sequencer\}] \times \{mm\})$
    END;
```

Fig. 13. Total Order Broadcast : Refined Order Event : Level-2

	Invariants	Required By
/*Inv-15*/	$ran(tdeliver) = tdeliver[\{sequencer\}]$	Order

Fig. 14. Invariants-V : Level-2

7.2 Second Refinement : Refinement of Order Event

Through this refinement we illustrate that a total order can be built using the messages delivered to the sequencer. As shown in the Fig. 11, a total order is generated as $totalorder := totalorder \cup (ran(tdeliver) \times \{mm\})$. It state that all messages delivered at any process are ordered before the new message *mm*.

In the refined *Order* event the totalorder is constructed as $totalorder := totalorder \cup (tdeliver[\{sequencer\}] \times \{mm\})$. It state that all messages delivered to the sequencer are ordered before the new message *mm*.

The specifications of this refinement are given in the Fig. 13. The replacement of the operations in the event *Order* generates proof obligations which require us to prove that the message delivered elsewhere in the system has also been delivered to the sequencer. In order to discharge the proof obligations we add the invariant *Inv-15* given in the Fig. 14. This invariant was sufficient to discharge the proof obligations.

7.3 Third Refinement : Introducing Sequence Numbers

In the third refinement, given in Fig. 15, we introduce the notion of computation message and the sequence numbers. The new variables *computation*, *seqno* and *counter* are introduced in the refinement typed as follows :

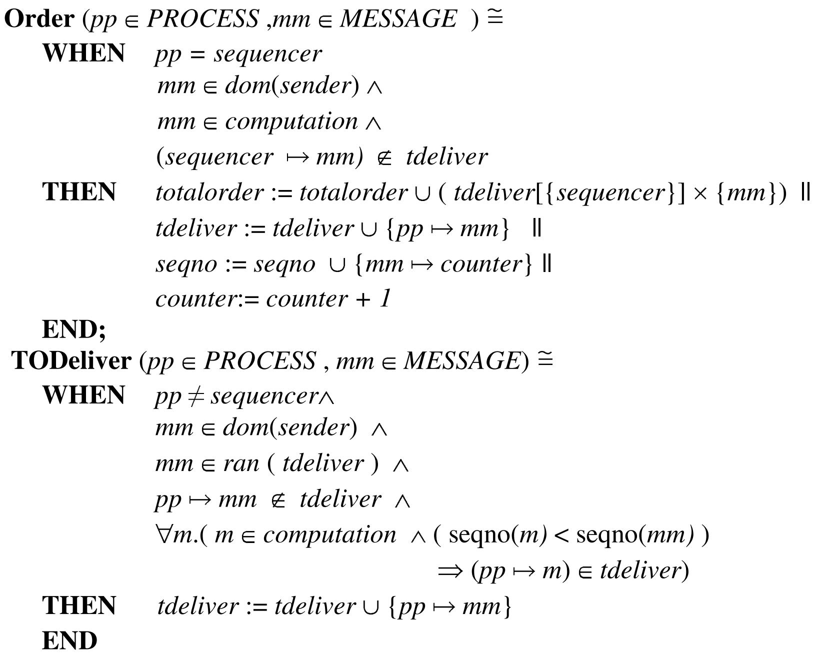
$$\begin{array}{l} computation \subseteq MESSAGE \\ seqno \in computation \rightarrowtail\!\!\!\!\!\!+\; Natural \\ counter \in Natural \end{array}$$

The variable *seqno* is used to assign sequence number to the computation messages. The *counter*, initialized with *zero*, is maintained by the sequencer process and incremented by *one* each time a control message is sent out by the *sequencer* process. It can be noted in the specification of *TODeliver* event that these message are delivered to the processes other than the sequencer in their sequence numbers. Consider the following guard of the abstract *TODeliver* event.

$$(m \mapsto mm) \in totalorder \Rightarrow (pp \mapsto m) \in tdeliver$$

The above is replaced by following guard in this refinement.

$$seqno(m) < seqno(mm) \Rightarrow (pp \mapsto m) \in tdeliver$$



```
Order ($pp \in PROCESS$ ,$mm \in MESSAGE$ ) $\widehat{=}$
    WHEN   $pp = sequencer$
           $mm \in dom(sender) \wedge$
           $mm \in computation \wedge$
           $(sequencer \mapsto mm) \notin tdeliver$
    THEN   $totalorder := totalorder \cup (tdeliver[\{sequencer\}] \times \{mm\})$ ||
           $tdeliver := tdeliver \cup \{pp \mapsto mm\}$ ||
           $seqno := seqno \cup \{mm \mapsto counter\}$ ||
           $counter := counter + 1$
    END;
 TODeliver ($pp \in PROCESS$ , $mm \in MESSAGE$) $\widehat{=}$
    WHEN   $pp \neq sequencer \wedge$
           $mm \in dom(sender) \wedge$
           $mm \in ran(tdeliver) \wedge$
           $pp \mapsto mm \notin tdeliver \wedge$
           $\forall m.(m \in computation \wedge (seqno(m) < seqno(mm))$
                                   $\Rightarrow (pp \mapsto m) \in tdeliver)$
    THEN   $tdeliver := tdeliver \cup \{pp \mapsto mm\}$
    END
```

Fig. 15. Total Order Broadcast : Level-3

	Invariants	**Required By**
*/*Inv-16*/*	$m1 \mapsto m2 \in totalorder$ $\Rightarrow seqno(m1) < seqno(m2)$	*Order,TOdeliver*
*/*Inv-17*/*	$m \in computation \wedge m \in dom(seqno)$ $\Rightarrow sequencer \mapsto m \in tdeliver$	*Order,TOdeliver*

Fig. 16. Invariants-VI : Level-3

The change of the guards in the *TODeliver* event generate new proof obligations. These proof obligations are discharged by adding new invariants given in the Fig. 16 to the model. Invariant *Inv-16* state that if *m1* precedes *m2* in abstract total order then the sequence number assigned to *m1* is less than the sequence number assigned to *m2*. The invariant *Inv-17* state that if a computation message has been assigned a sequence number then sequencer must have delivered it.

7.4 Fourth Refinement : Introducing Control Messages

In this refinement given in Fig. 17, we introduce the notion of control messages. A control message is broadcast by the sequencer process for each computation message. In this refinement, a process broadcasts a computation message *mm* to all processes including the *sequencer*. Upon delivery of this message, the sequencer assigns it a sequence number and broadcast its *control* message. All process except the *sequencer* deliver the corresponding computation messages in the order of the *sequence numbers*. This refinement consists of following new state variables typed as follows :

$$control \subseteq MESSAGE$$
$$messcontrol \in control \rightarrowtail\!\!\!\!\!\!\!+\;\; computation$$

The variables *control* and *computation* are used to represent a computation or a control message. The variable *messcontrol* is a partial injective function which defines relationship among a control message and its computation message. A mapping $(m1 \mapsto m2) \in messcontrol$ indicate that message *m1* is the *control message* related to the *computation message m2*. The set *ran(messcontrol)* contains the computation messages for which control messages has been sent by the sequencer. The guard $mm \in ran(tdeliver)$ of *TODeliver* event is replaced by the guard $mm \in ran(messcontrol)$ in this refinement. This indicate that a computation message is delivered to a process other than a sequencer only if its control message has been sent out by the sequencer. The change in the guards of *Order* and *TODeliver* events generate proof obligations which are discharged by adding following invariant to the model.

```
Order ($pp \in PROCESS ,mm \in MESSAGE,mc \in MESSAGE$ ) $\widehat{=}$
   WHEN   $pp = sequencer \wedge$
          $mm \in dom(sender) \wedge$
          $mm \in computation \wedge$
          $(sequencer \mapsto mm) \notin tdeliver \wedge$
          $mc \notin dom(messcontrol) \wedge$
          $mm \notin ran(messcontrol)$
   THEN   $totalorder := totalorder \cup ( tdeliver[\{sequencer\}] \times \{mm\})$ ||
          $tdeliver := tdeliver \cup \{pp \mapsto mm\}$ ||
          $control := control \cup \{mc\}$ ||
          $messcontrol := messcontrol \cup \{mc \mapsto mm\}$ ||
          $seqno := seqno \cup \{mm \mapsto counter\}$ ||
          $counter := counter + 1$
   END;
 TODeliver ($pp \in PROCESS , mm \in MESSAGE$) $\widehat{=}$
   WHEN   $pp \neq sequencer \wedge$
          $mm \in dom(sender) \wedge$
          $mm \in ran ( messcontrol ) \wedge$
          $pp \mapsto mm \notin tdeliver \wedge$
          $\forall m.( m \in computation \wedge ( \mathrm{seqno}(m) < \mathrm{seqno}(mm) )$
                                  $\Rightarrow (pp \mapsto m) \in tdeliver)$
   THEN   $tdeliver := tdeliver \cup \{pp \mapsto mm\}$
   END
```

Fig. 17. Total Order Broadcast : Level-4

	Invariants	**Required By**
*/*Inv-18*/*	$ran(messcontrol) \subseteq ran(tdeliver)$	*Order,TOdeliver*
*/*Inv-19*/*	$ran(messcontrol) \subseteq computation$	*Order,TOdeliver*

Fig. 18. Invariants-VII : Level-4

7.5 Fifth Refinement : Introducing Receive Control Event

A new event *ReceiveControl* is introduced in this refinement. This event model receiving a control message at a process. A new variable *receive* is also introduced in this refinement typed as $receive \in PROCESS \leftrightarrow control$. A mapping $p \mapsto m \in receive$ indicate that process p has received a control message m. The specifications of the refined events are given in Fig. 19.

```
**ReceiveControl** $(pp \in PROCESS\ ,\ mc \in MESSAGE\ ) \cong$
    WHEN  $mc \in control \wedge$
          $(pp \mapsto mc) \notin receive$
    THEN  $receive := receive \cup \{pp \mapsto mc\}$
    END
**TODeliver** $(pp \in PROCESS\ ,\ mm \in MESSAGE) \cong$
    WHEN  $pp \neq sequencer \wedge$
          $mm \in computation \wedge$
          $(pp \mapsto mm) \notin tdeliver \wedge$
          $(pp \mapsto messcontrol^{-1}(mm)) \in receive \wedge$
          $\forall m.(\ m \in computation \wedge (seqno(m) < seqno(mm)$
                  $\Rightarrow (pp \mapsto m) \in tdeliver)$
    THEN  $tdeliver := tdeliver \cup \{pp \mapsto mm\}$
    END
```

Fig. 19. Total Order Broadcast: Receive Control : Level-5

	Invariants	Required By
/*Inv-20*/	$m \in computation \wedge messcomtrol^{-1}(m) \in receive$ $\Rightarrow m \in ran(messcontrol)$	Order,TOdeliver

Fig. 20. Invariants-VIII : Level-5

As shown in the *TODeliver* event at Level-5, the guard $mm \in ran\ (messcontrol)$ is replaced by the the guard $(pp \mapsto messcontrol^{-1}(mm)) \in receive$. This guard of the *TODeliver* event ensures that a process *pp* delivers a computation message *mm* only when its corresponding control message has been received by the process *pp*. The change in the guards generate proof obligations associated with the event *TODeliver*. In order to discharge these proof obligations we add the invarinats given in Fig. 20.

8 Related Work

Distributed algorithms can be deceptive, may have complex execution paths and may allow unanticipated behavior. Rigorous reasoning about such algorithms is required to ensure that an algorithm achieves what it is supposed to do [25]. The group communication primitives has been proposed to develop fault-tolerant distributed services. The total order broadcast is one primitive that deliver messages to the sites in a distributed system in same order. The introduction of transactions based on group communication primitives represents an important step towards extending the power and generality of group communication for design and implementation of reliable fault-tolerant distributed computing applications [34]. The implementations of these group communication primitives has also been investigated for different distributed systems such Isis [10], Totem [29], Trans [27], Amoeba [36] and Transis [7]. The protocols in these systems use varying broadcast primitives and address group maintenance, fault tolerance and consistency services. The transaction mechanism in the management of replicated data is also considered in [6,8,31,32,34].

Group communication services have been studied as a basic building block for many fault tolerant distributed services, however the application of formal methods providing clear specifications and proofs of correctness is rare [16]. In [17], I/O automata are used for formal modelling and verification of a sequentially consistent shared object system in a distributed network. In order to keep the replicated data in a consistent state, a combination of total order multicast and point to point communication is used. In [18,33] the specification for group communication primitives are presented using I/O automata under different conditions such as partitioning among the group and dynamic view oriented group communication. The proof method supported in this method for reasoning about the system involves invariant assertions. An invariant assertion is defined as a property of the state of a system that is true in all execution. A series of invariants relating state variables and reachable states are proved by hand using the method of induction. In [37], a formal method is proposed to prove the total and causal order of multicast protocols. The formal results are provided in the paper that can be used to prove whether an existing system has the required property or not. Their solutions are based on the assumption that a total order is built using the service provided by a causal order protocol. In a similar work in [26], meta properties are used to express total order broadcast algorithm. The proof of correctness of the results are done by hand.

Instead, our approach of specifying the system and verification is based on the technique of abstraction and refinement. This formal approach carries a step-wise development from initial abstract specifications to a detailed design of a system in the refinement steps. Through the refinement proofs we verify that design of detailed system conforms to the abstract specifications. A refinement based approach to developing distributed systems in B is outlined in [12]. Use of refinement and decomposition rules in the development of telecommunications systems is outlined in [11]. The refinement approach of Event-B has also been used for the formal development of fault-tolerant communication systems [24] and fault-tolerant agent systems [23]. Other important work carried out using the refinement approach include verification of the IEEE 1394 tree protocol distributed algorithm [5], development of a secure communication system [13], development of a train system [2], verification of one copy equivalence criterion in a distributed database system [38]. The case study on development of Mondex purse system in Event-B [15] illustrates modelling strategies and the guidelines to achieve a high degree of automatic proofs.

9 Conclusions

In a replicated database, an update transaction modifies the requested data objects at various sites. A global update transaction may be submitted to any site and the effects of the update transaction are global, i.e., at commit all replicas at various sites must be updated. In case of abort, none of the sites update data objects. We have presented a rigorous design of distributed transactions for a replicated database in [38]. In this model, update messages from the coordinator site are assumed to broadcast using a reliable broadcast. A reliable broadcast imposes no restriction on the order in which messages are delivered to the participating sites. Unordered delivery of updates to the participating sites leads to the formation of deadlocks and the sites may abort conflicting transactions by timeouts. The failure of such transactions may be avoided if the updates are broadcast using a total order broadcast that delivers updates to the participating sites in a same order.

In this paper we have presented formal development of a system of total order broadcast. In the abstract model we outline how an abstract total order is constructed on the messages. Subsequently in a series of refinement steps we outline how an abstract total order can correctly be implemented by using the notion of control messages and sequence numbers. Instead of model checking, proving theorems by hand or proving correctness of the trace behavior, our approach consists of defining problem in the abstract model and introducing solutions or design details in the refinement steps. Through refinement checking we verify that the models in the refinement are valid refinement of abstract models. We used the *Click'n'Prove* B tool for proof management. This tool generates the proof obligations due to refinement and consistency checking, factorizes complex proof obligations in to relatively simpler proofs and helps discharge proof obligations by the use of automatic and interactive prover.

This case study illustrate how an incremental approach to system development can be used to obtain more concrete specifications. A powerful tool support helped us to discover several new invariants that helps to understand why a total order broadcast can correctly be implemented using sequence numbers. A clear relationship of computation and control message is outlined to indicate that our system generate exactly one control message for each computation message.

Table 1. Proof Statistics- Total Order Broadcast

Machine	Total POs	Completely Automatic	Required Interaction
Abstract Model	48	29	19
Refinement1	19	16	03
Refinement2	2	2	00
Refinement3	18	14	04
Refinement4	15	14	01
Refinement5	04	04	00
Overall	106	79	27

In this case study approximately 75% of the proof obligations were discharged using automatic prover. The proof obligations generated by the B tool also help discovering new system invariants. The proofs and the invariants help to precisely understand why a design decision or a solution proposed in the refinement is correct. The over all proof statistics is given in Table 1.

References

1. Abrial, J.-R.: The B-Book: Assigning programs to meanings. Cambridge University Press, Cambridge (1996)
2. Abrial, J.-R.: Train systems. In: Butler, et al. (eds.) [14], pp. 1–36
3. Abrial, J.-R., Butler, M., Hallerstede, S., Voisin, L.: An Open Extensible Tool Environment for Event-B. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 588–605. Springer, Heidelberg (2006)
4. Abrial, J.-R., Cansell, D.: Click'n Prove: Interactive Proofs within Set Theory. In: Basin, D., Wolff, B. (eds.) TPHOLs 2003. LNCS, vol. 2758, pp. 1–24. Springer, Heidelberg (2003)
5. Abrial, J.-R., Cansell, D., Méry, D.: A mechanically proved and incremental development of ieee 1394 tree identify protocol. Formal Asp. Comput. 14(3), 215–227 (2003)
6. Agrawal, D., Alonso, G., Abbadi, A.E., Stanoi, I.: Exploiting atomic broadcast in replicated databases (extended abstract). In: Lengauer, C., Griebl, M., Gorlatch, S. (eds.) Euro-Par 1997. LNCS, vol. 1300, pp. 496–503. Springer, Heidelberg (1997)
7. Amir, Y., Dolev, D., Kramer, S., Malki, D.: Membership algorithms for multicast communication groups. In: Segall, A., Zaks, S. (eds.) WDAG 1992. LNCS, vol. 647, pp. 292–312. Springer, Heidelberg (1992)
8. Babaoglu, Ö., Bartoli, A., Dini, G.: Replicated file management in large-scale distributed systems. In: Tel, G., Vitányi, P.M.B. (eds.) WDAG 1994. LNCS, vol. 857, pp. 1–16. Springer, Heidelberg (1994)
9. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley, Reading (1987)
10. Birman, K.P., Schiper, A., Stephenson, P.: Lightweigt causal and atomic group multicast. ACM Trans. Comput. Syst. 9(3), 272–314 (1991)
11. Butler, M.: Stepwise refinement of communicating systems. Science of Computer Programming 27(2), 139–173 (1996)
12. Butler, M.: An approach to the design of distributed systems with B AMN. In: Till, D., P. Bowen, J., Hinchey, M.G. (eds.) ZUM 1997. LNCS, vol. 1212, pp. 223–241. Springer, Heidelberg (1997)
13. Butler, M.: On the use of data refinement in the development of secure communications systems. Formal Aspects of Computing 14(1), 2–34 (2002)
14. Butler, M., Jones, C.B., Romanovsky, A., Troubitsyna, E. (eds.): Rigorous Development of Complex Fault-Tolerant Systems. LNCS, vol. 4157. Springer, Heidelberg (2006)
15. Butler, M., Yadav, D.: An incremental development of the mondex system in Event-B. Formal Aspects of Computing 20(1), 61–77 (2008)
16. Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Comput. Surv. 36(4), 372–421 (2004)
17. Fekete, A., Kaashoek, M.F., Lynch, N.: Implementing sequentially consistent shared objects using broadcast and point-to-point communication. Journal of the ACM 45(1), 35–69 (1998)

18. Fekete, A., Lynch, N.A., Shvartsman, A.A.: Specifying and using a partitionable group communication service. ACM Trans. Comput. Syst. 19(2), 171–216 (2001)
19. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann, San Francisco (1993)
20. Hadzilacos, V., Toueg, S.: A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR 94 -1425, Cornell University, NY (1994)
21. Kemme, B., Alonso, G.: A suite of database replication protocols based on group communication primitives. In: Proc. Intl. Conf. Distributed Computing System, Amsterdam, ICDCS, pp. 156–163 (1998)
22. Kemme, B., Pedone, F., Alonso, G., Schiper, A., Wiesmann, M.: Using optimistic atomic broadcast in transaction processing systems. IEEE Trans. Knowl. Data Eng. 15(4), 1018–1032 (2003)
23. Laibinis, L., Troubitsyna, E., Iliasov, A., Romanovsky, A.: Rigorous development of fault-tolerant agent systems. In: Butler, et al. (eds.) [14], pp. 241–260
24. Laibinis, L., Troubitsyna, E., Leppänen, S., Lilius, J., Malik, Q.A.: Formal Service-Oriented Development of Fault Tolerant Communicating Systems. In: Butler, M., Jones, C.B., Romanovsky, A., Troubitsyna, E. (eds.) Rigorous Development of Complex Fault-Tolerant Systems. LNCS, vol. 4157, pp. 261–287. Springer, Heidelberg (2006)
25. Lamport, L., Lynch, N.A.: Distributed computing: Models and methods. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B), pp. 1157–1199 (1990)
26. Liu, X., Renesse, R., Bickford, M., Krietz, C., Constable, R.: Protocol switching: Exploiting meta-properties. In: Intl. Workshop on applied reliable group communication, WARGC 2001, pp. 37–42. IEEE Computer Science, Los Alamitos (2001)
27. Melliar-Smith, P.M., Moser, L.E., Agrawala, V.: Broadcast protocols for distributed systems. IEEE Trans. Parallel Distrib. Syst. 1(1), 17–25 (1990)
28. Metayer, C., Abrial, J.R., Voison, L.: Event-B language. RODIN deliverables 3.2 (2005), http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf
29. Moser, L.E., Melliar-Smith, P.M., Agarwal, D.A., Budhia, R.K., Lingley-Papadopoulos, C.A.: Totem: A fault-tolerant multicast group communication system. Commun. ACM 39(4), 54–63 (1996)
30. Özsu, M.T., Valduriez, P.: Principles of Distributed Database Systems, 2nd edn. Prentice-Hall, Englewood Cliffs (1999)
31. Patiño-Martínez, M., Jiménez-Peris, R., Kemme, B., Alonso, G.: Middle-r: Consistent database replication at the middleware level. ACM Trans. Comput. Syst. 23(4), 375–423 (2005)
32. Pedone, F., Guerraoui, R., Schiper, A.: The database state machine approach. Distributed and Parallel Databases 14(1), 71–98 (2003)
33. Prisco, R.D., Fekete, A., Lynch, N., Shvartsman, A.: A dynamic view-oriented group communication service. In: PODC 1998: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing, pp. 227–236. ACM Press, New York (1998)
34. Schiper, A., Raynal, M.: From group communication to transactions in distributed systems. Communication of the ACM 39(4), 84–87 (1996)
35. Stanoi, I., Agrawal, D., El Abbadi, A.: Using broadcast primitives in replicated databases. In: Proc. of 18th IEEE Intl. Conf. on Distributed Computing System, ICDCS, pp. 148–155 (1998)

36. Tanenbaum, A.S., Kaashoek, M.F., van Renesse, R., Bal, H.E.: The amoeba distributed operating system - a status report. Computer Communications 14(6), 324–335 (1991)
37. Toinard, C., Florin, G., Carrez, C.: A formal method to prove ordering properties of multicast systems. ACM Operating Systems Review 33(4), 75–89 (1999)
38. Yadav, D., Butler, M.: Rigorous Design of Fault-Tolerant Transactions for Replicated Database Systems Using Event B. In: Butler, M., Jones, C.B., Romanovsky, A., Troubitsyna, E. (eds.) Rigorous Development of Complex Fault-Tolerant Systems. LNCS, vol. 4157, pp. 343–363. Springer, Heidelberg (2006)

Model-Based Testing Using Scenarios and Event-B Refinements

Qaisar A. Malik, Johan Lilius, and Linas Laibinis

Åbo Akademi University, Department of Information Technologies
Turku Centre for Computer Science (TUCS), Finland
{Qaisar.Malik,Johan.Lilius,Linas.Laibinis}@abo.fi

Abstract. In this paper, we present a model-based testing approach based on user provided testing scenarios. In this approach, when a software model is refined to add or modify features, the corresponding testing scenarios are automatically refined to incorporate these changes. The test cases, to be applied on the system under test, are generated from these scenarios. We use the Event-B formalism for software models, while user scenarios are represented as Communicating Sequential Process (CSP) expressions. The presented case study demonstrates how our approach can be used to test different features of a system such as incorporated fault-tolerance mechanisms.

1 Introduction

Testing is an important but expensive activity in the software development life cycle. With advancements in the model-based approaches for software development, new ways have been explored to generate test-cases from existing software models of the system, while cutting the cost of testing at the same time. These new approaches are usually referred to as *model-based testing.* A software model is a specification of the system which is developed from the given requirements early in the development cycle [9]. In the model-based development (MBD), this model is then refined until a required abstraction level is reached from which the implementation code can be generated, or written by hand. Model-based testing (MBT) is an approach for deriving tests from the models using automated techniques. The intended cost reductions arise because

1. the tests can be generated by tools, without hand coding,
2. changes in the model do not need extensive re-writing of test-code,
3. test coverability can be improved because we can define coverability on the model and guarantee that important parts of the system are tested.

Test selection is an active research area where both formal and informal approaches exist. In this paper, we propose a testing approach based on user-provided testing scenarios. In the Model-Based development (MBD), we start with the initial model that is created to implement the set of use cases explaining the desired behavior. The use cases are usually described in natural language and

M. Butler et al. (Eds.): Fault Tolerance, LNCS 5454, pp. 177–195, 2009.
© Springer-Verlag Berlin Heidelberg 2009

are used by requirement engineers to represent a part of the requirements given by customers. Often these use cases also form the basis for the acceptance tests of the final product. However, there is an abstraction gap between the use cases and the final acceptance tests. In order to support automatic generation of tests from given software models, we need to bridge this gap. In this paper, we study this issue in the context of formal MBD by using Event-B [5,4] as our modelling language. Event-B supports stepwise system development by refinement. We will represent formal models of software systems as Event-B specifications. On the other hand, we express the provided use-cases as scenarios in Communicating Sequential Process (CSP) [10]. Representing scenarios as CSP expressions gives us better structure and associated tool support. We show, by making *controlled* refinements of our Event-B models, that we can automatically derive the final tests from the original scenarios. An overview of the approach is given is given in Fig. 1. This work is based on our earlier approach [16] for scenario-based testing from B models.

The program refinement approach has been extensively used to model complex software systems, such as control systems, communication systems etc. It allows us to gradually incorporate implementation details and therefore, helps us to deal with overall complexity. For such systems, it is really important to be dependable i.e., to function even in the presence of faults and failures. The refinement approach allows us to gradually incorporate fault tolerance mechanisms describing how the system reacts on abnormal situations. In [11], a methodology for developing fault-tolerant systems in a stepwise manner is proposed. Our work, presented in this paper, is also based on stepwise development and thus very suitable for testing the desired properties and features of the systems that were developed in such a way.

The organisation of the paper is as follows. Section 2 gives brief introduction to Model-based testing (MBT) and details our scenario-based MBT methodology. In Section 3, we describe the formal framework for our approach and show how it can be instantiated in specific cases. In Section 4, we illustrate our approach by a case-study on the development of a fault-tolerant system. Finally, Section 5 contains related work and some concluding remarks.

2 Model-Based Testing

Model-based testing is a general notion used for testing of software systems using models of the system. In [17], model-based testing is defined as *automation of the design of black-box tests*. The system to be tested is referred as *System-Under-Test* (SUT). The SUT is an executable implementation which is considered as a black-box during the testing process, i.e., only inputs and outputs of the system are visible externally. The SUT is tested by applying *test case(s)*. A *test case* is defined as sequence of steps to test the correct behavior of a particular functionality or feature of the system [2]. In model-based testing, the test cases are generated from the given models of the system.

2.1 Scenario-Based Approach for Model-Based Testing

Our model-based testing approach is based on stepwise system development [6] using behavioral models of the system. By a behavioral model, we mean that the system behavior is modelled as states together with operations (or events) on the states. In the stepwise development process, an abstract model is first constructed and then further refined to include more details (e.g., functionalities) of the system. Generally, these models can be either formal, informal or both. In this work we only consider formal models. These models are usually created from the requirements.

In the development process, we start with an abstract model and gradually, given by a number of refinement steps, obtain a sufficiently detailed model. The final system, the system under test (SUT), is an implementation of this detailed model. Ideally, the implementation should be automatically generated, which would make it correct by construction. However, in practice, due to the abstraction gap between formal models and executable implementations, this is not always possible. As a result, an implementation is often hand-coded while consulting with the formal models. The left hand-side of the Fig.1 graphically presents this process.

The right hand side of the Fig.1 depicts a parallel process where we start from the requirements and construct an abstract scenario. This abstract scenario is a valid behavior of the abstract model present on the same level of abstraction. In short, we say that the abstract model *conforms* to the abstract scenario. In later stages, we refine our abstract scenario along the refinement chain of the system

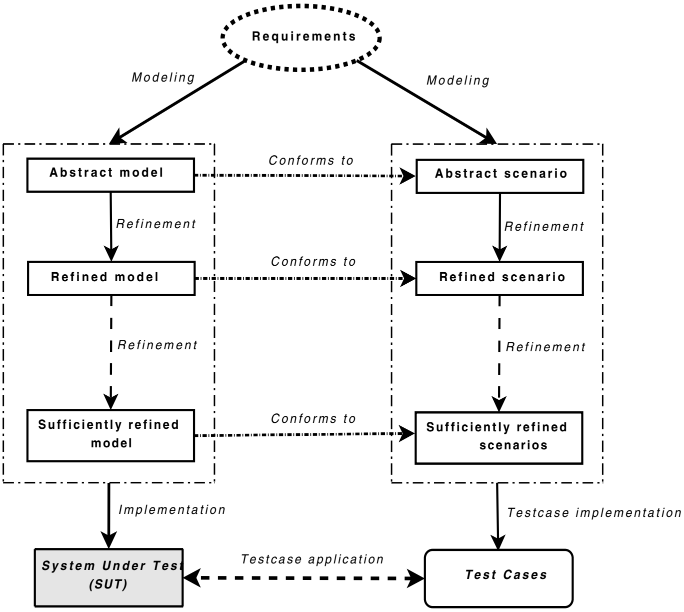


Fig. 1. Overview of our Model-based testing approach

models. Finally, the sufficiently refined scenarios are translated into executable test cases which are then applied to SUT. Later in this section, we provide more detailed information about definitions and representation of the scenarios.

In the literature, one can find several definitions of the term *scenario.* Generally speaking, as described in [1], a scenario is a description of possible actions and events in the future. In the field of software engineering, scenarios have been used to represent various concepts like system requirements, analysis, user-component interactions, test cases etc. [13]. In this work, we use the term *scenario* to represent a *test scenario* for our system under test (SUT). A test scenario is one of the possible valid execution paths that the system must follow. In other words, it is one of the expected functionalities of the system. For example, in a hotel reservation system, booking a room is one functionality, while canceling a pre-booked room is another one. In this article, we use both terms functionality and scenario interchangeably.

Each scenario usually includes more than one system-level event or procedure, which are executed in some particular sequence. Since, in a non-trivial system, there can be many possible execution sequences of the events, identifying all of the valid sequences may not be an easy task. In our approach, we deal with this complexity in a stepwise manner. On the abstract level, an initial scenario is provided by the user. The abstract model of the system *conforms to* or formally *satisfies* this scenario, meaning that the scenario is in fact a the valid behavior of the model. This scenario is a sequence of the abstract event(s) in their order of execution. Once an abstract scenario has been provided, afterwards, for each refinement step scenarios are refined automatically. Fig.2 shows the refinement process where an abstract model M_i is refined by M_{i+1} (denoted by $M_i \sqsubseteq_c M_{i+1}$). This refinement ($\sqsubseteq_c$) is a controlled refinement as will be discussed in detail in section 3.2. Scenario S_i is an abstract scenario, formally satisfiable ($\models$) by specification model M_i, is provided by the user. In the next refinement step, scenario S_{i+1} is constructed automatically from M_i, M_{i+1} and S_i in such a way that S_{i+1} is formally satisfied or conformed by the model M_{i+1}. The automatically generated scenario S_{i+1} represents functionalities, in part or whole, of the model M_{i+1}.

In some cases, the model M_{i+1} may contain some extra functionalities or features, such as incorporated fault-tolerance mechanisms, which were omitted or out of scope of scenario S_i. These *extra features*, denoted by S_{EF}, can be added in the scenario S_{i+1} manually. The modified scenario $S_{i+1} \cup S_{EF}$ must be checked (by means of available tools) to be satisfied/conformed by the model M_{i+1}. We can follow the same refinement process, now starting with $S_{i+1} \cup S_{EF}$, until we get S_{i+n}.

After the final refinement, the system is implemented from the model M_{i+n}. This implementation is called *system under test (SUT).* The scenario S_{i+n} is unfolded into the executable test cases that are then applied to SUT. In the section 3, we will demonstrate how scenarios are represented and refined. In the next section, we present some mathematical preliminaries for our model-based testing approach.

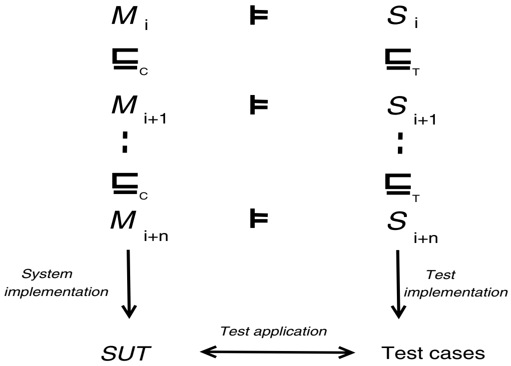


Fig. 2. Refinement of Models and Scenarios

2.2 Mathematical Preliminaries

The formal models that we use in this work are *labelled transition systems*. These are formally defined in the following:

Definition 1
A *labelled transition system* (LTS) is a 4-tuple $\langle S, L, T, s_0 \rangle$ where

- S is countable, non-empty set of *states*;
- L is a countable set of *labels*;
- $T \subseteq S \times L \times S$ is the *transition relation*
- $s_0 \in S$ is the *initial state.*

The labels in L represent the events in the system. Let $l = \langle S, L, T, s_0 \rangle$ be a label transition system with $s, s^{'}$ in S and let $\mu_i \in L$.

$$\begin{array}{lll} s \xrightarrow{\mu} s' & =_{def} & (s, \mu, s') \in T \\ s \xrightarrow{\mu_1 \cdots \mu_n} s' & =_{def} & \exists\, s_0, \ldots, s_n : s = s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} s_n = s' \\ s \xrightarrow{\mu_1 \cdots \mu_n} & =_{def} & \exists\, s' : s \xrightarrow{\mu_1 \cdots \mu_n} s' \end{array}$$

behavior of LTS is defined in terms of *traces* where a *trace* is a finite sequence of events in the system. The set of all traces over L is denoted by L^*. For an LTS $l = \langle S, L, T, s_0 \rangle$, the behavior function, denoted by *beh(LTS)*, is defined as

$$beh(l) \quad =_{def} \quad \{\sigma \in L^* \mid s_0 \xrightarrow{\sigma} \ \} \qquad \square$$

Definition 2

1. A *test sequence*, denoted by t, is a finite sequence of events, $\mu_1, \mu_2, \ldots \mu_n$, in the system defined as

$$s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} s_n$$

where $n \in \mathbb{N}$ and s_i are system states.

2. A *test scenario* , denoted by *ts*, is collection of *test sequences* present in the *behavior* of LTS *l*,

$$ts \subseteq beh(l) \qquad \square$$

Definition 3

1. The *System Under Test* (SUT) is an executable implementation of the models. Abstractly, an SUT can be viewed as a Labelled Transition System (LTS) having states and events.
2. A test case denoted as *tc*, is a finite *test sequence* to be tested on SUT. Moreover, each test case also includes the expected result(s) of the test case execution. This result is used to compute the *verdict function.*
3. A *verdict function* ν is defined, in terms of *Labelled Transition System (LTS)* with *test sequence* (ts), as

   $$\nu(LTS, ts) = Passed \qquad \textit{iff} \qquad ts \in beh(LTS)$$

   Similarly, in the context of *System Under Test* (SUT), the verdict function is used to check if the test case execution has given expected results or not.

   $$\nu(SUT, tc) = Passed \qquad \textit{iff} \qquad tc \in beh(SUT)$$
   $$Failed \text{ otherwise} \qquad \square$$

3 Formal Framework

3.1 Modeling in Event-B

The Event-B [5,4] is a recent extension of the classical B method [3] formalism. Event-B is particularly well-suited for modeling event-based systems. The common examples of event-based systems are reactive systems, embedded systems, network protocols, web-applications and graphical user interfaces.

In Event-B, the specifications are written in Abstract Machine Notation (AMN). An abstract machine encapsulates state (variables) of the machine and describes operations (events) on the state. A simple abstract machine has following general form

```
MACHINE AM
SETS TYPES
VARIABLES v
INVARIANT I
INITIALISATION INIT
EVENTS
 E_1  =  ...
 ...
 E_N  =  ...
END
```

A machine is uniquely defined by its name in the **MACHINE** clause. The **VARIABLE** clause defines state variables, which are then initialized in the **INITIALISATION** clause. The variables are strongly typed by constraining predicates of the machine invariant I given in the **INVARIANT** clause. The invariant defines essential system properties that should be preserved during system execution. The operations of event based systems are atomic and are defined in the **EVENT** clause. An event is defined in one of two possible ways

$$E = \textbf{WHEN} \;\; g \;\; \textbf{THEN} \;\; S \;\; \textbf{END}$$

$$E = \textbf{ANY} \;\; i \;\; \textbf{WHERE} \;\; C(i) \;\; \textbf{THEN} \;\; S \;\; \textbf{END}$$

where g is a predicate over the state variables v, and the body S is an Event-B statement specifying how the variables v are affected by execution of the event. The second form, with the **ANY** construct, represents a parameterized event where i is the parameter and *C(i)* contains condition(s) over i. The occurrence of the events represents the observable behavior of the system. The event guard (g or *C(i)*) defines the condition under which event is enabled.

Event-B statements are formally defined using the weakest precondition semantics [8]. The defined semantics is used to demonstrate correctness of the system. To show correctness of an event-based system it is necessary to formally prove that the invariant is true in initial state and every event preserves the invariant:

$$\begin{array}{l} wp(INIT, I) = true, \quad \text{and} \\ g_i \wedge I \Rightarrow wp(E_i, I) \end{array}$$

An Event-B machine describes a state-machine that represents particular behavior of the machine M. We can describe it as an instantiation of labelled transition system, defined in Section 2.2.

Definition 4
An Event-B machine denotes a labelled transition system $\langle S, L, T, s_0\rangle$ where

- S is set of Event-B *states*, where the state of an Event-B machine is a particular assignment of values to the Event-B variables;
- L is a set of event names;
- the transition relation T is constructed from single transitions of the form

  $$s \xrightarrow{ev} s'$$

  where s and s' are Event-B states and ev is the name of an event.
- s_0 is the state of an Event-B machine after initialization. □

3.2 Refinement of Event-Based Systems

The basic idea underlying the formal stepwise development is to design a system implementation gradually, by a number of correctness preserving steps, called *refinements*. The refinement process starts from creating an abstract, albeit implementable, specification and finishes with generating executable code. In general, the refinement process can be seen as a way to reduce non-determinism of the abstract specification, to replace abstract mathematical data structures by data structures implementable on a computer, and, hence, gradually introduce implementation decisions.

We are interested in how refinement affects the external behavior of a system under construction. Such external behavior can be represented as a trace of observable events, which then can be used to produce test cases. From this point of view, we can distinguish two different types of refinement called *atomicity* refinement and *superposition* refinement.

In ***Atomicity*** refinement, one event operation is replaced by several operations, describing the system reactions in different circumstances the event occurs. Intuitively, it corresponds to a branching in the control flow of the system. Let us consider an abstract machine AM_A and a refinement machine AM_AR given below. It can be observed that an abstract event *E* is split (replaced) by the refined events *E1* and *E2*. Any execution of *E1* and *E2* will correspond to some execution of abstract event *E1*. It is also shown graphically in Fig.3(a).

```
                            REFINEMENT AM_AR
MACHINE AM_A                REFINES AM_A
 ...                         ...
EVENTS                      EVENTS
 E = WHEN g                 $E_1$ ref E = WHEN g ∧ $g_1$ THEN $S_1$ END
       THEN S END           $E_2$ ref E = WHEN g ∧ $g_2$ THEN $S_2$ END
END                         END
```

In ***Superposition*** refinement, new implementation details are introduced into the system in the the form of new events that were invisible in the previous specification. These new events can not affect the variables of the abstract specification and only define computations on newly introduced variables. For our purposes, it is convenient to further distinguish two basic kinds of superposition refinement, where

- a non-looping event is introduced,
- a looping but terminating event is introduced.

Let us consider an abstract machine AM_S and a refinement machine AM_SR as shown below

```
                         REFINEMENT AM_SR
MACHINE AM_S             REFINES AM_S
 ...                      ...
EVENTS                   EVENTS
 E = WHEN g               E = WHEN g THEN S END
       THEN S END         $E_1$ = WHEN $g_1$ THEN $S_1$ END
END                      END
```

It can be observed that the refined specification contains both the old and the new events, *E* and *E1* respectively. To ensure termination of the new event(s), the **VARIANT** clause is added in a refinement machine. This **VARIANT** clause contains an expression over a well-founded type (e.g., natural numbers). The new events should decrease the value of the variant, thus guaranteeing that the new events will eventually return the control as the variant expression can not be decreased indefinitely. These two types of refinements are also shown graphically in Fig.3(b) and (c).

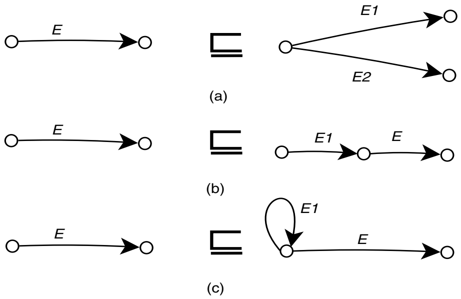


Fig. 3. Basic refinement transformations

Let us note that the presented set of refined types is by no means complete. However, it is sufficient for our approach based on user defined scenarios.

The event-based development gradually (in a controlled way) reveals more observable states of the system. Even if the idea of controlled refinement is generic, the inspiration for it came from [11], where a fault-tolerant agent system was developed, gradually incorporating fault-tolerance mechanisms. In this way, both normal and abnormal functionalities (faults/failures) can be modelled. To model abnormal behavior, the system reactions in the form of specific fault-tolerance mechanisms are added to the refined models. These mechanisms include modelling of special degraded states, recovery actions, failure modes and so on. In the development of safety/security-critical or communication systems, handling of such events often contains the most of the system's complexity.

3.3 Scenario Refinement and Representation

In section 2.1, we introduced the notion of scenarios. Each such scenario can be represented as a Communicating Sequential Process (CSP) [10] expression. Since we develop our system in a controlled way, i.e. using basic refinement transformations described in Section 3.2, we can associate these Event-B refinements with syntactic transformations of the corresponding CSP expressions. Therefore, knowing the way model M_i was refined by M_{i+1}, we can automatically refine scenario S_i into S_{i+1}. To check whether a scenario S_i is a valid scenario of its model M_i, i.e., model M_i satisfies ($\models$) scenario S_i, we use ProB [12] model checker. ProB supports execution (animation) of Event-B specifications, guided by CSP expressions. The satisfiability check is performed at each refinement level as shown in the Fig.2. The refinement of scenario S_i is the CSP trace-refinement [14] denoted by $\sqsubseteq_T$.

As we have described before, the scenarios are represented as CSP expressions. We refine our models in a controlled way targeting at individual events. We assume that the events are only executed when their guards are enabled. For simplicity, we omit the guard information from CSP expressions. Here we will discuss how individual refinement steps affect the scenarios. Let us assume we are given an abstract specification M_0 with three events, namely, `A`, `B` and `C`, and

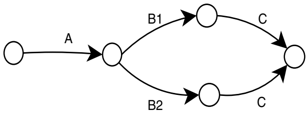


Fig. 4. Atomicity Refinement

a scenario S_0 representing the execution order of these events: first the event A, then the event B, and finally the event C. The CSP expression for scenario S_0 is given by

$$\mathtt{S_0} = \mathtt{A} \rightarrow \mathtt{B} \rightarrow \mathtt{C} \rightarrow \mathtt{SKIP}$$

In the next refinement step, the model M_0 is refined by M_1. This refinement step may involve any of three types of the supported refinements, as discussed in Section 3.2. In order to reflect the changes from the refined model into scenarios, we need to update/refine our CSP expressions accordingly. We will discuss the scenario refinement step one by one in the following.

Atomicity Refinement. Let us suppose an event B is refined using atomicity refinement. As a result, it is split into two events namely $\mathtt{B_1}$ and $\mathtt{B_2}$. It means that the older event B will be replaced by two new events $\mathtt{B_1}$ and $\mathtt{B_2}$ modelling a branching in the control flow. As a CSP expression we can represent the new refined scenario S_1 as

$$\mathtt{S_1} = \mathtt{A} \rightarrow ((\mathtt{B_1} \rightarrow \mathtt{CONT}) \ \sqcap \ (\mathtt{B_2} \rightarrow \mathtt{CONT}))$$
$$\mathtt{CONT} = \mathtt{C} \rightarrow \mathtt{SKIP}$$

where $\sqcap$ is an internal choice operator in CSP. We use internal choice operator instead of external one because all of the events in our system have guards which are strengthened in a way that at a time only one event is enabled for execution. This can also be represented graphically as shown in Fig. 4

Superposition refinement. Let us suppose we use superposition refinement to refine an event C. As a result, a new non-looping event D is introduced in the system.

The new scenario S_1 is represented as a CSP expression in the following:

$$\mathtt{S_1} = \mathtt{A} \rightarrow \mathtt{B} \rightarrow \mathtt{D} \rightarrow \mathtt{C} \rightarrow \mathtt{SKIP}$$

In the second case, let us suppose we again use superposition refinement to refine event C. However, this time a new looping event D is introduced into the system. The corresponding CSP expression is given as

$$\mathtt{S_1} = \mathtt{A} \rightarrow \mathtt{B} \rightarrow \mathtt{D} \rightarrow \mathtt{C} \rightarrow \mathtt{SKIP}$$

where D is defined as

$$\mathtt{D} = \mathtt{D} \ \sqcap \ \mathtt{SKIP}$$

The new scenario can also be represented graphically as Fig. 5.

In the next section, we outline how scenarios are unfolded into test cases.

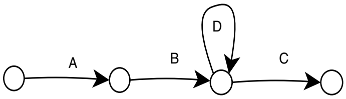


Fig. 5. Superposition refinement type II

3.4 Instantiation of Scenarios as Test Cases

Now we need to translate a scenario into a test case. The distinction between the two is the following. For a scenario, we are *mathematically* guaranteed that the model will conform to the scenario and it can be checked, e.g., by model-checking. For the SUT, we clearly can not give any such guarantee. Thus for each event in the scenario we need to check that SUT has executed the corresponding action correctly. For our approach, we use the ProB model checker, which has the functionality to animate B specifications guided by the provided CSP expression. After the execution of each event, present in the scenario, information about the changed system state is stored.

In other words, the execution trace is represented by a sequence of pairs $< e, s' >$, where e is an event and s' is a post-state (the state after execution of event e). From now on we will refer to a single pair $< e, s' >$ as an *ESPair*.

For a finite number of events $e_1, e_2 .....e_n$, present both in the model M and the System Under Test (SUT), a test case t of length n, defined in terms of test a sequence in section 2.2, consists of an initial state *INIT* and a sequence of *ESPairs*

$$t = INIT, \{< e_1, s'_1 >, < e_2, s'_2 >, ... < e_n, s'_n >\}$$

Similarly, a scenario, as formally defined in section 2.2 as finite set of related test cases, i.e., a scenario *ts* is given as

$$ts = \{t_1, t_2, .., t_n\}$$

As mentioned earlier, *ESPair* relates an event with its post-state. This information is stored during test-case generation. For SUT these stored post-states become expected outputs of the system and act as a *verdict* for the testing. After execution of each event, the expected output is compared with the output of the SUT. This comparison is done with the help of probing functions. The probing functions are such functions of SUT that at a given point of their invocation, return state of the SUT. For a test-case to pass the test, each output should match the expected output of the respective event. Otherwise, we conclude that a test case has failed. In the same way, test cases from any refinement step can be used to test implementation as long as both the implementation and the respective test cases share the same events and signatures.

4 Testing Development of a Fault-Tolerant System

In this section we will demonstrate our approach on a case study development of a fault tolerant agent system. The case study was first presented in [11]. Agent systems are examples of complex distributed systems. Though agents operate in unreliable communication environment, often such systems have high reliability requirements imposed on them. Thus, ability to operate in a volatile error prone environment and have regularly to cope with abnormal situations that are typical for agent systems is the essential requirement for designing such systems. The development of such systems should also facilitate systematic integration of the fault tolerance mechanisms into agent applications.

The most typical faults that these applications encounter are temporal connectivity losses, which can cause failures of communication between cooperating agents or between an agent and the server. In [11], the agent and server software are developed from the corresponding B specifications, where the fault tolerance features are gradually integrated into these specifications. Hence, the development of the fault-tolerance mechanisms becomes a part of the system development.

For example, while modelling collaboration of agents, we have to define the agent behavior in the presence of message losses, hardware failures, etc. Generally speaking, fault tolerance in agent systems is supported by a set of abstractions used by the application developers and a specialised middleware. The abstractions are developed to systematically separate the normal system behavior from the abnormal one. The middleware detects disconnections and, when necessary, involves agents into error recovery.

The formal development, presented in [11], is used in this section to demonstrate our model based testing approach. The main refinement steps introducing the abnormal system behavior (disconnections, hardware failures) and the corresponding system reactions (recovery actions, aborting) are reflected in the corresponding test scenarios.

We start our development with a very simple specification of a mobile agent system, where an agent performs three basic tasks when connected to the server. These basic tasks are named as *Engage*, *NormalActivity* and *Disengage*. To incorporate the fault-tolerant behavior, the system is repeatedly refined using the basic refinement types described in Section 3.3. The introduction of fault-tolerance increases the complexity of the system. Our testing methodology can be applied to test the new scenarios that result from this complexity. The initial *Event-B* machine named *AgentSystem* specifies the three basic events, mentioned above.

```
MACHINE $AgentSystem$
SETS $Agents$
VARIABLES $agents$
INVARIANT $agents \subseteq Agents$
INITIALISATION $agents := \emptyset$
EVENTS
  Engage = ANY $aa$ WHERE $aa \in Agents \wedge aa \notin agents$
          THEN $agents := agents \cup \{aa\}$ END;
  NormalActivity = ANY $aa$ WHERE $aa \in Agents \wedge aa \in agents$
          THEN skip END ;
```

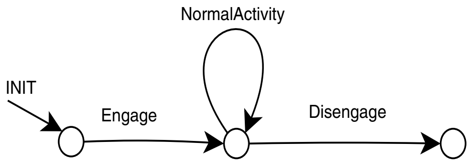


Fig. 6. Execution graph of machine *AgentSystem*

```
  Disengage = ANY aa WHERE aa ∈ Agents ∧ aa ∈ agents
          THEN agents := agents - {aa} END
END
```

In the specification *AgentSystem*, let us note that the event *NormalActivity* may happen zero or more times. The sequence of events, as determined by the specification, is shown in Fig.6. The INIT is an initialisation event.
The given scenario can be expressed as the following Communicating Sequential Process (CSP) expression

```
AgentSystem  = Engage -> Node1
Node1 = NormalActivity -> Node1
Node1 = Disengage -> SKIP
```

In the next refinement machine *AgentSystem1*, the event *Disengage* is refined into two new events in order to differentiate between leaving normally or because of a failure. This refinement step is *atomicity* refinement as discussed in Section 3.3. The other events of the specification remain the same. The execution graph for this refinement is shown in Fig.7.

```
REFINEMENT AgentSystem1 REFINES AgentSystem
  ...
EVENTS
  ...
  NormalLeaving ref Disengage = ANY aa WHERE aa ∈ Agents ∧ aa ∈ agents
          THEN agents := agents - {aa} END
  Failure ref Disengage = ANY aa WHERE aa ∈ Agents ∧ aa ∈ agents
          THEN agents := agents - {aa} END
END
```

The testing scenario for *AgentSystem1* is expressed as the following CSP expression

```
AgentSystem1 = Engage -> Node1
Node1 = NormalActivity -> Node1
Node1 = NormalLeaving -> SKIP
Node1 = Failure -> SKIP
```

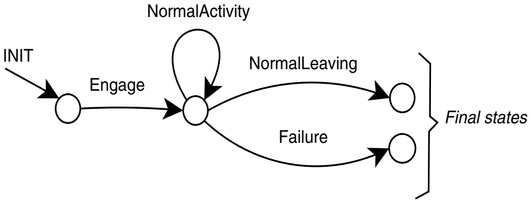


Fig. 7. Execution graph of machine *AgentSystem*1

In the next refinement machine *AgentSystem2*, we introduce temporary loss of connection for our agents. This new event is called *TempFailure*. This refinement step introduces a looping event (see superposition refinement in Section 3.3). To guarantee termination of the new event, we introduce a new variable *disconn_limit*, which is used as a variant.

```
REFINEMENT AgentSystem2 REFINES AgentSystem1
  ...
VARIABLES agents, disconn_limit
INVARIANT disconn_limit ∈ NAT
VARIANT disconn_limit
EVENTS
  ...
  NormalActivity = ANY aa WHERE aa ∈ agents
          THEN disconn_limit := Disconn_limit END;
  TempFailure = ANY aa WHERE (aa ∈ agents)
          THEN disconn_limit := disconn_limit - 1 END;
END
```

The execution flow for *AgentSystem2* is given in Fig.8. The CSP expression for the refined scenario is given as

```
AgentSystem2  = Engage -> Node1
Node1 = NormalActivity -> Node1
Node1 = TempFailure -> Node1
Node1 = NormalLeaving -> SKIP
Node1 = Failure -> SKIP
```

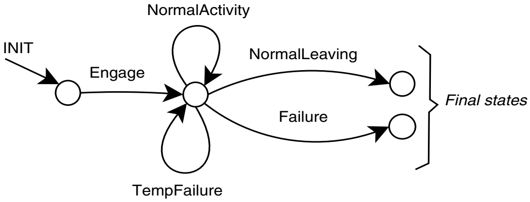


Fig. 8. Execution graph of machine *AgentSystem*2

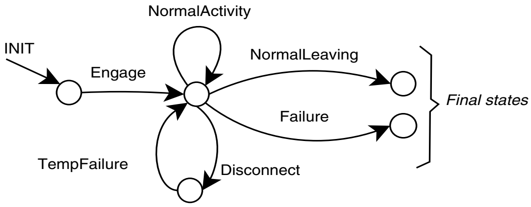


Fig. 9. Execution graph of machine *AgentSystem*3

In next refinement machine *AgentSystem3*, a new event *Disconnect* is introduced. It is the event that precedes (causes) *TempFailure* event. This refinement is a superposition refinement introducing a non-looping event. A new variable *timers* is used to ensure order of execution.

REFINEMENT *AgentSystem3* **REFINES** *AgentSystem2*

...

EVENTS

...

Disconnect = **ANY** *aa* **WHERE** $aa \in agents$

THEN $timers := timers \cup \{aa\}$ **END**

TempFailure = **ANY** *aa* **WHERE** $(aa \in agents) \wedge (aa \in timers)$

THEN $disconn\_limit := disconn\_limit - 1 \mid\mid timers := timers - \{aa\}$ **END**;

END

The execution flow for *AgentSystem*3 is shown in Fig.9. The refined scenario is represented as following CSP expression

```
AgentSystem3  = Engage -> Node1
Node1 = NormalActivity -> Node1
Node1 = Disconnect -> TempFailure -> Node1
Node1 = NormalLeaving -> SKIP
Node1 = Failure -> SKIP
```

In the final refinement step, we elaborate on error recovery and time expiration by splitting the events *TempFailure* and *Failure* by atomicity refinement.

REFINEMENT *AgentSystem4* **REFINES** *AgentSystem3*

...

EVENTS

...

TimerExpiration `ref` **Failure** = **ANY** *aa* **WHERE**

$(aa \in agents) \wedge (aa \in ex\_agents)$

THEN $agents := agents - \{aa\} \mid\mid ex\_agents := ex\_agents - \{aa\}$ **END**;

AgentFailure `ref` **Failure** = **ANY** *aa* **WHERE**

$(aa \in agents) \wedge (aa \notin timers) \wedge (aa \notin ex\_agents)$

THEN $agents := agents - \{aa\}$ **END**;

Connect `ref` **TempFailure** = **ANY** *aa* **WHERE** $(aa \in agents) \wedge (aa \in timers)$

THEN $disconn\_limit := disconn\_limit - 1 \mid\mid timers := timers - \{aa\}$ **END**;

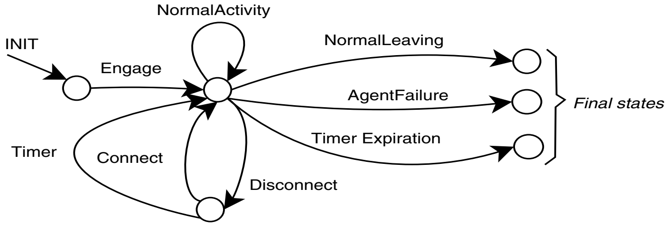
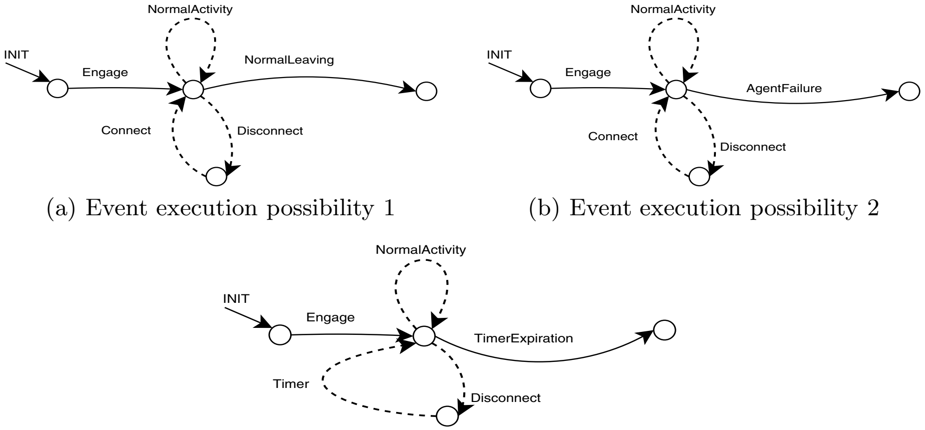


Fig. 10. Execution graph of machine *AgentSystem4*



(a) Event execution possibility 1

(b) Event execution possibility 2

(c) Event execution possibility 3

Fig. 11. All possible Event execution scenarios

```
  Timer ref TempFailure = ANY aa WHERE (aa ∈ agents) ∧ (aa ∈ timers)
        THEN disconn_limit := disconn_limit - 1 || ex_agents := ex_agents ∪ {aa} ||
        timers := timers - {aa} END
END
```

The execution graph for *AgentSystem4* is shown in Fig.10. This graph shows all the possible events with their respective states but the order of execution is controlled by their guards. In addition, the Fig.11 shows all the possible scenarios based on the information derived from events' guards and bodies. The dashed arrows represent possible loops of the event(s) during the execution. In order to generate concrete test cases from such models, the number of executions of an event in the loop can be restricted to some finite bound. The value for this bound depends on user's coverage criteria.

When testing event-based systems, the system can have demonic choice and can execute the loop-event forever. One of the possible ways to restrict such an execution is to introduce variants both in the specification and the implementation. The variant will ensure finite number of executions of that event. In the case where implementation does not has such variant and the system may continue execution forever then applying test case of finite length will detect a live-lock in the system. Here it is assumed that no valid implementation has such infinite execution.

The CSP representations of the *AgentSystem4* machine is given in the following.

```
AgentSystem4 = Engage -> Node1
Node1 = NormalActivity -> Node1
Node1 = Disconnect -> Node2
Node1 = Failure -> SKIP
Node1 = NormalLeaving -> SKIP
Node1 = TimerExpiration -> SKIP
Node2 = TempFailure -> Node1
Node2 = Timer -> Node1
```

Since in Event-B, every event is guarded, here, it is assumed that each event is enabled only when its corresponding *guard* is enabled. The guard information can also be expressed within CSP expressions as

```
(BooleanGuard & EventName)
```

These CSP expressions are finally unfolded into test cases by the methodology described in Section 3.4. These test cases are applied on the implementation to test the fault-tolerance scenarios.

In this case study, we showed how our scenario-based testing approach can be used in developing a fault-tolerant software application. We described a stepwise development approach showing how testing scenarios are refined alongside the refinements in the corresponding models. In this case study example, we used a chain of Event-B machines where the test cases are finalized from the single sufficiently refined machine. However, in practice, it is possible to decompose functionalities of the system across multiple components (machines). Our scenario-based testing approach would also work in that case provided that these components are developed from an abstract component in a consistent fashion also obeying the basic refinement types described earlier in the section 3.2 of this paper.

5 Related Work and Conclusions

The existing tools or techniques for model-based testing using model-oriented languages (e.g., see [7,15]) are based on the notion of the coverage graph, which is obtained from symbolic execution of the model. In these approaches, as a first step, the input space of an operation is partitioned into equivalent classes to create the corresponding operation instances and then the coverage graph is constructed. This coverage graph contains the sequence of operations which are then tested in the implementation. However, in these approaches, the user scenarios are not represented and tested.

In our earlier work [16], we presented the scenario-based testing approach for B models where we designed an algorithm for constructing test sequences across different refinement [6] models. However, this algorithm is exponential in nature thus limiting its practical applicability.

In this paper, we presented a model-based testing approach based on automatic refinement of test scenarios. In this work, we also described basic refinement rules, allowing us to do the development in a controlled way, and transform our testing scenarios according to those rules. This approach does not involve any exponential algorithm thus making it more applicable in practice. This methodology is well suited for development of complex systems in which the fault-tolerance mechanisms are incorporated alongside the main system functionality. However, this automatic test case generation method can also be used in formal software development process in general. The presented methodology also allows us to have multiple instances of testing scenarios to test different functionalities or features of the system.

Currently, our approach supports several basic refinement types. However, in the future, we are going to extend our method by including more refinement types. We also plan to work on building an execution environment where the test cases can be executed on the system-under-test in a controlled manner. This would enable us to interpret the test results for each test case execution. In the cases, where an error is discovered, the environment should help to trace this error to a particular place in the corresponding software model.

Acknowledgments

This work is supported by IST FP6 RODIN Project.

References

1. Cambridge Dictionary for English, `http://dictionary.cambridge.org/`
2. Software Testing Online Blog. `http://testingsoftware.blogspot.com/2006/02/test-case.html`
3. Abrial, J.-R.: The B-Book. Cambridge University Press, Cambridge (1996)
4. Abrial, J.-R.: Event Driven Sequential Program Construction (2000), `http://www.matisse.qinetiq.com`
5. Abrial, J.-R., Mussat, L.: Introducing Dynamic Constraints in B. In: Bert, D. (ed.) B 1998. LNCS, vol. 1393, p. 83. Springer, Heidelberg (1998)
6. Back, R.-J., von Wright, J.: Refinement calculus, part i: Sequential nondeterministic programs. In: REX Workshop, pp. 42–66 (1989)
7. Bernard, E., Legeard, B., Luck, X., Peureux, F.: Generation of test sequences from formal specifications: Gsm 11-11 standard case study. Softw. Pract. Exper. 34(10), 915–948 (2004)
8. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)
9. Dalal, S.R., et al.: Model Based Testing in Practice. In: Proc. of the ICSE 1999, Los Angeles, pp. 285–294 (1999)

10. Hoare, C.A.R.: Communicating sequential processes. Prentice-Hall, Inc., Englewood Cliffs (1985)
11. Laibinis, L., Troubitsyna, E., Iliasov, A., Romanovsky, A.: Rigorous development of fault-tolerant agent systems. In: RODIN Book, pp. 241–260 (2006)
12. Leuschel, M., Butler, M.: ProB: A model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003)
13. Naslavsky, L., Alspaugh, T.A., Richardson, D.J., Ziv, H.: Using Scenarios to support traceability. In: Proc. of 3rd int. workshop on Traceability in emerging forms of software engineering (2005)
14. Roscoe, A.W.: The theory and practice of concurrency. Prentice-Hall, Englewood Cliffs (1998)
15. Satpathy, M., Leuschel, M., Butler, M.J.: ProTest: An automatic test environment for B specifications. Electr. Notes Theor. Comput. Sci. 111, 113–136 (2005)
16. Satpathy, M., Malik, Q.A., Lilius, J.: Synthesis of scenario based test cases from *b* models. In: FATES/RV, pp. 133–147 (2006)
17. Utting, M., Legeard, B.: Practical Model-Based Testing. Morgan Kaufmann Publishers, San Francisco (2006)

Recording Process Documentation in the Presence of Failures

Zheng Chen and Luc Moreau

School of Electronics and Computer Science
University of Southampton
Southampton, SO17 1BJ, UK
zc05r@ecs.soton.ac.uk, L.Moreau@ecs.soton.ac.uk

Abstract. Scientific and business communities present unprecedented requirements on *provenance*, where the provenance of some data item is the process that led to that data item. Previous work has conceived a computer-based representation of past executions for determining provenance, termed *process documentation*, and has developed a protocol, PReP, to record process documentation in service oriented architectures. However, PReP assumes a failure free environment. Failures lead to process documentation unable to be recorded, losing the evidence that a process occurred. This is not acceptable in the applications relying on process documentation and would cause disastrous consequences. This paper describes our solution, F_PReP, a protocol for recording process documentation in the presence of failures. A complete formalisation of the protocol using Abstract State Machines is also presented.

1 Introduction

In scientific and business communities, a wide variety of applications have presented unprecedented requirements [20] for knowing the provenance of their data products, e.g., where they originated from and what has happened to them since creation. In chemistry experiments, provenance is used to detail the procedure by which a material is generated, allowing the material to be patented. In healthcare applications, in order to audit if proper decisions were made for a patient, there is a need to trace back the origins of these decisions. In engineering manufacturing, keeping track of the history of generated data in simulations is important for users to analyse the derivation of their data products. In finance business, the provenance of some data item establishes the origin and authenticity of the data item produced by financial transactions, enabling reviewers and auditors to verify if these transactions are compliant with specific financial regulations.

To meet these requirements, Groth et al. [15] have proposed an open architecture to record and access a computer-based representation of past executions, termed *process documentation*, which can be used for determining the provenance of data. A generic recording protocol, PReP [16], has been developed to provide interoperable means for recording process documentation in the context of service oriented architectures. In this architecture, process documentation consists

M. Butler et al. (Eds.): Fault Tolerance, LNCS 5454, pp. 196–219, 2009.
© Springer-Verlag Berlin Heidelberg 2009

of a set of assertions (termed *p-assertions*) made by *asserting actors* (i.e., either clients or services) involved in a process (i.e., the execution of a workflow). A dedicated repository, termed *provenance store*, is used to maintain p-assertions. For scalability reason, multiple provenance stores may be employed and process documentation may end up distributed, linked by pointers recorded along with p-assertions in each store. Using the pointer chain, distributed process documentation can be retrieved from one store to another.

Recording process documentation in the presence of failures is an issue that has been lacking attention so far. PReP assumes a system in which no failure occurs. However, large scale, open distributed systems are not failure-free [8,9]. For example, a service may not be available and network connection may be broken. The presence of failures may prevent process documentation from being recorded, losing the evidence that a process occurred. We now draw a parallel between the documentation of a process and a particular type of evidence in a legal setting, testimony. The absence of testimony from eyewitnesses to a crime scene would make it difficult for juries to make a judgement about whether to believe the set of claims provided by a suspect. Similarly, the unavailability of process documentation is not acceptable in the above domains that rely on process documentation to determine the provenance of their data products. It may also cause disastrous consequences as in the example of a provenance-based service billing system. In this system, users are charged according to their usage of services described by process documentation. If a user invoked a service, but documentation fails to describe this invocation, then the user will be charged too little, which must be avoided.

To address this problem, we have designed a recording protocol, F_PReP, which provides remedial actions and a novel component, Update Coordinator, to guarantee the recording of process documentation in the case of failures. The protocol has been formalised as an abstract state machine and its correctness has been proved. This paper details the protocol and presents its formalisation.

The rest of the paper is organised as follows: Section 2 introduces some terminology and identifies a set of requirements that the protocol should meet. Section 3 states our failure assumptions and defines protocol messages. In Section 4, we present a formalisation of the protocol and detail the protocol's behaviour. Then we outline the proof of the protocol's correctness in Section 5. Finally, Section 6 discusses related work, followed by a conclusion in Section 7.

2 Terminology and Requirements

2.1 Terminology

Process documentation describes a past process that led to a result. Such a process is modelled as a causally connected set of interactions between actors involved in that process [14]. An *interaction* is concerned with one application message exchanged between two actors, i.e., its sender and its receiver. An actor documents an interaction by making p-assertions to provide a sender or receiver's view of the interaction. Process documentation therefore consists of a set of p-assertions.

A p-assertion can document the application message exchanged in an interaction (*interaction p-assertion*) or the internal state of an actor (*actor state p-assertion*), such as time and memory usage, in the context of an interaction. It can also be a *relationship p-assertion*, capturing the internal causal connections between interactions within the scope of an actor, i.e., the interaction where an output message is sent (*effect interaction*) and the interaction where an input message is received (*cause interaction*).

PReP specifies that *both* actors in an interaction must make p-assertions documenting the interaction for accountability or verification purposes. For scalability reason, an actor can use various stores to record p-assertions about different interactions, though p-assertions about the same interaction must be recorded in the same place. Besides, the p-assertions made by the two actors in an interaction are also allowed to be recorded in two different stores. A notion of link, i.e., a pointer to a provenance store, has been introduced to connect distributed documentation [14].

There are two types of links, *viewlink* and *causelink*. If the two actors in an interaction use two different stores, each actor records a *viewlink* that points to the provenance store where the opposite party recorded their p-assertions about that interaction. Therefore, both views of an interaction can be retrieved by navigating from one provenance store to the other. The causelink is used in relationship p-assertions. If the p-assertions that represent a cause interaction are recorded in a different provenance store, a *causelink* is embedded in the relationship p-assertion, indicating which provenance store the p-assertions representing the cause interaction are stored in. To facilitate the description of our protocol, we define a term *ownlink* as a pointer to the provenance store where an actor records its *own* p-assertions.

Figure 1 shows an example of how links are recorded. Actor A sends an application message M2 to actor B as a consequence of message M1. A uses provenance stores P_R and P_A to record p-assertions about the interactions in which M1 and M2 are exchanged, respectively. B records p-assertions about the receipt of M2 in provenance store P_B. In order to exchange a viewlink to B, A includes its ownlink to P_A in M2. B then extracts the link and records it as its viewlink in P_B. As a result, a viewlink from P_B to P_A is created (shown by the arc VL 1). We assume that A knows from its configuration that B always stores its p-assertions in P_B. Hence, A records a viewlink to P_B in P_A. Finally, A makes a relationship p-assertion between its effect interaction containing M2 and the previous cause interaction containing M1. In the relationship p-assertion, it adds a causelink to P_R, where the p-assertions related to the cause interaction are stored. A then records the relationship p-assertion in P_A, thus connecting P_A to P_R shown by the arc CL.

By recording links, a pointer chain can be formed connecting all the provenance stores hosting the documentation of a process. Using the pointer chain, distributed documentation can be retrieved from one store to another.

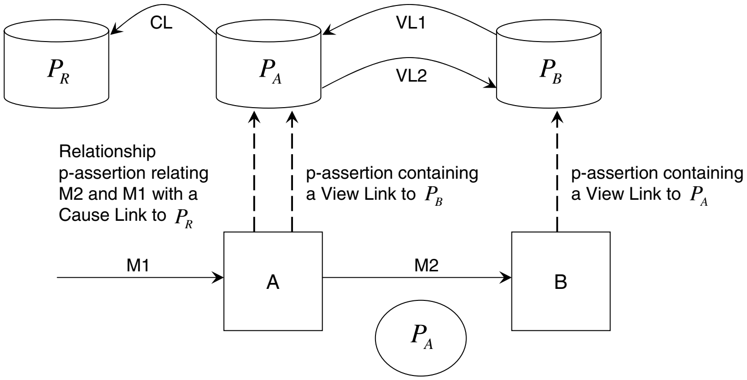

CL
VL1
VL2
P_R
P_A
P_B
Relationship p-assertion relating M2 and M1 with a Cause Link to P_R
p-assertion containing a View Link to P_B
p-assertion containing a View Link to P_A
M1
M2
A
B
P_A

Fig. 1. A example of link [14]

2.2 Requirements

Miles et al. [20] have presented requirements that a provenance system should support, such as verifiability, accountability, reproducibility, preservation, scalability, generality, customisability, non-repudiation and distribution. They have been of particular importance in motivating the design of our protocol. We now identify several new requirements that are related to failures.

PReP does not specify well-defined behaviour when recording documentation in the presence of failures. For example, it assumes an actor always obtains an acknowledgement from a provenance store for receiving a p-assertion and hence does not consider the situation where the acknowledgement is lost or provenance store crashes before storing that p-assertion. This may result in incomplete process documentation, which requires us to design a robust protocol to meet the following requirement:

Requirement 1 (Guaranteed Recording). *After a process finishes execution, the entire documentation of that process must eventually be recorded in provenance store(s).*

Distributed process documentation is connected by a chain of pointers (links) to enable retrievability. Accurate pointers must exist even in the presence of failures, leading to two requirements.

Requirement 2 (Viewlink Accuracy). *Viewlinks recorded for each interaction of a process must eventually be accurate in provenance stores. Each must point to the store where the other actor in the same interaction recorded p-assertions documenting that interaction.*

Requirement 3 (Causelink Accuracy). *Causelinks recorded during a process must eventually be accurate in provenance stores. Each must point to the store where p-assertions about the corresponding cause interaction were recorded.*

Creation and recording p-assertions have already introduced overhead into the application [13]. The remedial actions specified by the protocol may however take up computing resources and interfere with applications. In terms of recording performance, we identify another two requirements:

Requirement 4 (Efficient Recording). *Recording p-assertions and taking remedial actions should be efficient and introduce minimum overhead.*

Requirement 5 (Transparent Recording). *Recording p-assertions and taking remedial actions should be transparent to the application.*

Among the above requirements, requirements *Guaranteed Recording*, *Viewlink Accuracy* and *Causelink Accuracy* are concerned with the protocol's correctness, which are to be proved in Section 5.

3 Protocol Description

We firstly outline the design philosophy of F_PReP and state several assumptions, under which F_PReP meets the requirements identified in Section 2.2. Then, we define the protocol's messages and describe the protocol.

3.1 Design

The goal of our work is to design a *general* protocol, i.e., application and implementation independent, for recording process documentation in large, open distributed environments where a large number of provenance stores are present and failures may occur. Since PReP has provided an application independent solution to recording process documentation, we decided to derive PReP in order to inherit its generic nature.

There are several challenges in designing a distributed protocol that can cope with failures. Firstly, we need to state an appropriate failure model and systematically identify system behaviour in the case of failures. Failures are non-deterministic in nature and typically very hard to predict. Restricting our scope to particularly failures is hence necessary. Secondly, the protocol may involve the co-operation of several parties such as asserting actors, provenance stores, and if necessary, additional components. Designing such a distributed protocol is notoriously difficult, since we have to stay in control of not only the normal system behaviour when there is no failure but also of the complex situations which can occur when failures happen.

We restrict ourselves to certain failures that may occur during the recording of p-assertions into a provenance store.

Assumption 1. *Provenance stores may crash, i.e., they halt and stop any further execution, and can be restarted from their latest consistent state*[1]*.*

[1] The provenance store has been implemented as a stateless web service with a database storage system. Hence the latest consistent state refers to the initial state of the service and the latest checkpointed state of the database.

Assumption 2. *Messages to/from provenance stores can be lost, reordered but not duplicated in communication channels.*

We do not consider the failures of asserting actors and the exchange of application messages since they are application dependant. Applications should provide fault tolerance mechanisms to ensure asserting actors' availability and reliable exchange of application messages.

Assumption 3. *An asserting actor has several provenance stores to use.*

Given that we are considering an open system where there are a large number of provenance stores, it is reasonable to make use of alternative stores to provide fault tolerance.

We now analyse several failure types in a recording scenario where an asserting actor sends a p-assertion (pa) to a provenance store (PS) and PS replies the actor with an acknowledgement (ack) after recording pa in its persistent storage.

- The message pa is lost;
- PS crashes before receiving pa;
- PS crashes after receiving pa and before recording pa;
- PS crashes after recording pa and before replying ack;
- The message ack is lost.

From an asserting actor's perspective, all these failure types may lead to the incapability of receiving an ack from a provenance store. An asserting actor can set a timeout when waiting for an ack message. If the actor does not receive a response within that time, it knows failures *may* have occurred; it can then send the p-assertion again. We note that a low speed network or a provenance store experiencing slowdown can also result in an expired timeout. Since a p-assertion may be recorded in a provenance store even in the case of timeout, a provenance store should be designed to handle duplicate p-assertions due to retransmission, and always return the same acknowledgement for a specific p-assertion.

We identify several remedial actions that the protocol needs to take in the presence of failures. The primary one is to resend a p-assertion to a provenance store due to a timeout. After several reattempts, if the p-assertion still fails to be acknowledged, an actor may use alternative stores to resubmit the p-assertion until it is acknowledged. A successful receipt of an acknowledgement tells the actor that the p-assertion being acknowledged has been recorded in a provenance store.

Since distributed process documentation is connected using links to enable retrievability, the use of alternative provenance stores causes a link to the original store incorrect. Hence, an asserting actor needs to take other remedial actions. To satisfy *Causelink Accuracy*, it can maintain history information of using alternative stores during its participation in a process. The protocol checks an actor's causelinks when recording relationship p-assertions and updates them according to the history information. To achieve *Viewlink Accuracy*, we introduce a novel component, Update Coordinator, to facilitate viewlink updating. An update coordinator is only involved when an alternative store is used, which means it does not participate in every interaction, hence introducing small overhead.

Assumption 4. *The update coordinator does not fail.*

We can use the traditional fault-tolerance mechanisms such as replication to ensure its availability. This is feasible since we can have only one coordinator per process and a coordinator maintains only a small amount of information, as illustrated later. However, it is infeasible to use the replication mechanism for provenance stores for two reasons. Firstly, we have assumed an open system where there are a great number of provenance stores, it is hard to assume each is facilitated with replicated backups. Secondly, though replication is sophisticated, it comes with a significant cost due to preserving the *one-copy equivalence* property [23]. Given that the documentation produced in a process can be on the order of terabytes [11], replication becomes very expensive and time consuming. Therefore, compared with replicating provenance stores, the use of alternative stores is a more general, simple and flexible approach.

To meet requirements *Efficient Recording* and *Transparent Recording*, F-PReP is designed to be an asynchronous protocol, allowing actors to send p-assertions at any time. This means that actors can choose when to record p-assertions without delaying their execution. Secondly, all p-assertions about one interaction are submitted in a single batch and hence can be acknowledged using one acknowledgement message, saving on the overhead of establishing network connections. Thirdly, remedial actions, e.g., selecting alternative stores, are taken by the protocol irrespective of the application.

3.2 Messages

F_PReP is a distributed protocol, specifying the behaviour of actors (i.e., asserting actors, provenance stores and update coordinator) and their communications. It is defined based on *interaction* i.e., the exchange of an application message between a sender and receiver.

There are six messages in the protocol: Application Message (app), Interaction Record Message (record), Record Ack Message (ack), Repair Message (repair), Update Message (update), and Update Ack Message (uack). We now define each message with Figure 2, which provides an example of actors exchanging these messages.

Application Message. The application message app is exchanged by all application actors. It contains application specific data needing to be transferred between actors. In the context of a provenance system, the application message is adapted to include interaction contextual information: an interaction key and the sender's ownlink.

An interaction key is generated by the sender in an interaction for uniquely identifying the interaction from all other interactions. The receiver can then use the same interaction key to record p-assertions about the same interaction.

In Figure 2, we assume that the key for the interaction where the sender, a, sends an application message to the receiver, b, is i. We also assume the default provenance stores that a and b use are $PS1$ and $PS2$, respectively. In Step 1, a

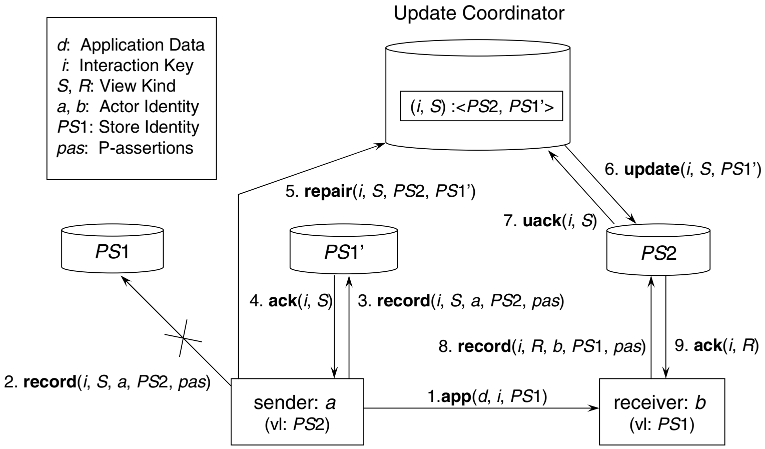


Fig. 2. Protocol Message Exchanges

sends an app to b containing application data d, interaction key i and a's ownlink to $PS1$ (Step 1). Upon receiving app, b becomes aware of its viewlink to $PS1$. We assume that a's viewlink to $PS2$ has been made available to a by means not explained in the figure; the viewlink can be built into a at deployment time or transferred to a in a response message or in an extra message from b.

Interaction Record Message. For each interaction, both actors document the interaction by asserting p-assertions and sending them in an interaction record message, record, to their respective provenance stores. The message contains: (1) an interaction key, identifying the interaction being documented; (2) a view kind, indicating the role of the asserting actor in the interaction, i.e., a sender or a receiver; (3) an actor identity, representing the asserting actor that documents the interaction, which is essential for recording attributable process documentation; (4) a viewlink of the asserting actor for that interaction; (5) a set of p-assertions that describe the interaction.

In Figure 2, both a and b create a set of p-assertions, pas, about the interaction, i, and send them in record messages with their viewlink to PS2 and PS1, respectively (Steps 3, 8). We note that the two record messages can be sent in any order, not restricted by the step numbers in the figure.

The set of p-assertions must contain an interaction p-assertion to document the exchange of an app message. If app is the consequence of receiving other messages, then the sender of app must make a relationship p-assertion to capture the causal connections between these messages.

Due to the asynchronous nature of the protocol, an asserting actor accumulates record messages in a local queue and submits them to a provenance store at its most convenient time. Before delivering a record message to a provenance store, an actor checks all the relationship p-assertions in the message and updates incorrect causelinks in order to meet *Causelink Accuracy* requirement. These actions are detailed in Section 4.3.

Record Ack Message. A provenance store acknowledges record message by means of an acknowledgement message ack, only *after* it has successfully recorded the content of record in its persistent storage. An ack message includes an interaction key and a view kind, indicating from which view of an interaction, a record is being acknowledged. Therefore, one ack can acknowledge a set of p-assertions in a record, which reduces communication overhead.

An asserting actor sets a timeout when waiting for an ack immediately after it sends a record to a provenance store. This helps the actor take remedial actions without waiting too long. If an ack is not received before the timeout, then the actor resends the same record to the actor's default store or an alternative store. Only after receiving an ack acknowledging a record can the actor eliminate the record from its local queue. An ack means that the acknowledged record message has been processed and recorded in a provenance store persistently.

In Figure 2, a sends a record to its default store $PS1$ (Step 2) but does not receive an ack before a timeout. Then it selects another store $PS1'$ to use (Step 3) and finally receives an ack (Step 4).

Repair Message. An asserting actor sends a repair message, repair, to an update coordinator to request an update of the other actor's viewlink. It consists of four elements: (1) an interaction key, indicating in which interaction the opposite actor's viewlink is to be updated; (2) the asserting actor's view kind in the interaction; (3) a pointer ($DestPS$) to the provenance store recording the opposite actor's viewlink in the interaction; (4) the actor's ownlink, pointing to the provenance store from which the actor received an ack for that interaction.

An actor issues a repair request only if it used an alternative store in an interaction, which results in the other actor's viewlink incorrect. In Figure 2, the sender sends its record to the alternative store $PS1'$ (Step 3) and receives an ack (Step 4). As a consequence, the receiver's viewlink to $PS1$ becomes incorrect, hence requiring an update. In order not to interfere with applications to support *Transparent Recording* requirement, the protocol does not allow the sender to directly inform the receiver with its new ownlink, which is now pointing to $PS1'$. Instead, the sender requests an update coordinator (Step 5) to help update the receiver's provenance store (Step 6).

An update coordinator is necessary since both sender and receiver may issue a repair request in an interaction. This cannot be achieved by direct update of the other actor's provenance store, because at that moment, one does not know which store the opposite actor is actually using. In Figure 2, if the receiver uses an alternative store to record its p-assertions, then the sender's viewlink to $PS2$ becomes incorrect as well. In that case, the receiver needs to issue another repair request to the coordinator.

Since an update coordinator is not involved in every interaction, we recommend that all the application actors participating in a process employ one coordinator. If using more than one, then any two actors exchanging an application message must share the same one in order to ensure Viewlink Accuracy requirement. The identifier of a coordinator can be built in actors or exchanged to other actors in the application message app. Figure 2 employs the former approach.

We do not consider the loss of repair messages in channel, which can be solved by using an extra acknowledgement message and retransmission actions. Assumption 4 implies that a repair request can always be processed by an update coordinator.

Update Message. The update coordinator sends an update message, update, to a provenance store in order to update a viewlink in that store. The message contains: (1) an interaction key, indicating for which interaction, the opposite actor's viewlink needs to be updated; (2) the view kind of the asserting actor that issued a repair request for that interaction; (3) the ownlink of the requesting actor. The *DestPS* field in the repair message tells the update coordinator where to send the update message.

In order to deal with the case where both actors in an interaction each issue a repair request, which can be in any order, the update coordinator maintains request information: the identity of the destination store, specified by the *DestPS* field in the repair message, and the requesting actor's ownlink. This request information is indexed by the pair of interaction key and view kind. In Figure 2, after receiving a repair request from the sender, the coordinator records a tuple $(PS2, PS1')$ indexed by the pair (i, S). Then the coordinator sends to store $PS2$ an update message containing the sender's ownlink to $PS1'$ (Step 6). Therefore, the receiver's viewlink stored in $PS2$ is replaced with $PS1'$ and hence becomes correct.

If the update coordinator receives two repair messages each from one asserting actor in an interaction, then it sends out two update messages after performing operations using the stored request information to ensure that both update messages are delivered to correct destination stores. We detail the coordinator's internal behaviour in Section 4.5.

We note that a provenance store may receive an update and a record message in any order (Steps 6, 8). The protocol specifies that the viewlink obtained from update is NOT overwritten by the one from record in order to achieve *Viewlink Accuracy* requirement.

Update Ack Message. After updating a viewlink in a provenance store, the store returns an acknowledgement message uack, containing an interaction key and a view kind, to the update coordinator acknowledging the respective update message. Since update or uack messages may be lost in channel according to Assumption 2, the coordinator sets a timeout when waiting for a uack and an expired timeout leads to resending the same update message.

4 Protocol Formalisation

F_PReP has been formalised through the use of an abstract state machine (ASM). The ASM notation we adopt has been used previously to describe a distributed reference counting algorithm [22] and a fault-tolerant directory service for mobile agents [21]. The abstract machine characterises the behaviour of actors with respect to the messages they send and receive. This behaviour is specified by the permissible transitions that the ASM is allowed to perform. Such a formalisation provides a precise, implementation-independent means of describing the system.

We begin by describing the state space of the ASM, and then proceed to discuss its transitions. Finally, we detail the behaviour of each kind of actors.

4.1 System State Space

Figure 3 shows the system state space. We identify specific subsets of actors in the system, namely, the senders, the receivers, provenance stores, and update coordinators. The set of each of protocol messages is defined formally as an inductive type. For example, the set of Application Messages is defined by an inductive type whose constructor is app and whose parameters are from the set of DATA, IK and OL. The notation DATA refers to the set of application related data. The set of all protocol messages ($\mathcal{M}$) is defined as the union of these message sets. Messages are exchanged over a set of communication channels, $\mathcal{K}$. Since no assumption is made about message order in communication channels, $\mathcal{K}$ is represented as bags of messages between pairs of actors. The power set notation ($\mathbb{P}$) denotes that there can be more than one of a given element.

We define the set of relationship p-assertions as an inductive type whose constructor is $\mathsf{rel\text{-}pa}$. The names of relationships are given in the set REL. Since a relationship p-assertion captures causal connections between effect interaction and cause interaction(s), we use the set EID and CID to index the respective interactions, each containing the interaction's key (IK) and the role (VK) that the actor plays in that interaction. With EID or CID, the p-assertions about a related interaction can be found in a local provenance store or a remote store (indicated by a causelink from the set CL). The set of interaction p-assertions can be constructed by $\mathsf{i\text{-}pa}$ whose parameter is from the set IK and application data set DATA. Since actor state p-assertions are not used in the formalisation, we do not model them to simplify the state space. The set of all kinds of p-assertions (PA) is defined as the union of these p-assertion sets.

The internal functionality of each kind of actors is modelled as follows.

Sender and Receiver State Space. An asserting actor (indexed by an asserter identity) uses various tables ($in\_T \in$ IN, $asserter\_T \in$ ASSERTER, $log\_T \in$ LOG, $queue\_T \in$ QUEUE, $lc \in$ LC and $timer\_T \in$ TIMER) to record p-assertions into a provenance store. A table maps a key to a tuple. For example, the table ($asserter\_T$) maps an interaction key ($\kappa \in$ IK) and the actor's view kind ($v \in$ VK) to a tuple of four elements: the state of an interaction record message during recording ($str \in$ STR), the actor's ownlink ($ol \in$ OL), viewlink ($vl \in$ VL) and the p-assertions created in the interaction ($\mathbb{P}\,$(PA)).

As all the data that an asserting actor works upon is located in received messages, these incoming messages and interaction keys identifying these messages are stored in a table (in_T), which is used when creating p-assertions.

$$\begin{aligned}
\mathrm{A} &= \{a_1, \ldots, a_n\} && \text{(Set of Actor Identities)}\\
\mathrm{SID} &\subseteq A && \text{(Sender Identities)}\\
\mathrm{RID} &\subseteq A && \text{(Receiver Identities)}\\
\mathrm{PID} &\subseteq A && \text{(Provenance Store Identities)}\\
\mathrm{CID} &\subseteq A && \text{(Coordinator Identities)}
\end{aligned}$$

$$\begin{aligned}
\mathcal{M} &= \mathsf{app} : \mathrm{DATA} \times \mathrm{IK} \times \mathrm{OL} \rightarrow \mathcal{M} && \text{(Set of Protocol Messages)}\\
&\quad\mid\ \mathsf{record} : \mathrm{IK} \times \mathrm{VK} \times \mathrm{A} \times \mathrm{VL} \times \mathbb{P}(\mathrm{PA}) \rightarrow \mathcal{M} &&\\
&\quad\mid\ \mathsf{ack} : \mathrm{IK} \times \mathrm{VK} \rightarrow \mathcal{M} &&\\
&\quad\mid\ \mathsf{repair} : \mathrm{IK} \times \mathrm{VK} \times \mathrm{DESTPS} \times \mathrm{OL} \rightarrow \mathcal{M} &&\\
&\quad\mid\ \mathsf{update} : \mathrm{IK} \times \mathrm{VK} \times \mathrm{OL} \rightarrow \mathcal{M} &&\\
&\quad\mid\ \mathsf{uack} : \mathrm{IK} \times \mathrm{VK} \rightarrow \mathcal{M} &&\\
\mathcal{K} &= A \times A \rightarrow Bag(\mathcal{M}) && \text{(Set of Channels)}
\end{aligned}$$

$$\begin{aligned}
\mathrm{R} &= \{m \in \mathcal{M} | m = \mathsf{record}(\kappa, v, a, vl, pas)\} && \text{(Set of Interaction Records)}\\
\mathrm{IK} &= \mathrm{SID} \times \mathrm{RID} \times N && \text{(Set of Interaction Keys)}\\
\mathrm{VK} &= \{\mathrm{S}, \mathrm{R}\} && \text{(Set of ViewKinds)}\\
\mathrm{OL} &= \mathrm{PID} && \text{(Set of Ownlinks)}\\
\mathrm{VL} &= \mathrm{PID} && \text{(Set of Viewlinks)}\\
\mathrm{DESTPS} &= \mathrm{PID} && \text{(Set of Destination Stores)}
\end{aligned}$$

$$\begin{aligned}
\mathrm{PA} &= \mathsf{rel\text{-}pa} : \mathrm{REL} \times \mathrm{EID} \times \mathbb{P}(\mathrm{CID}) \rightarrow \mathrm{PA} && \text{(Set of P-Assertions)}\\
&\quad\mid\ \mathsf{i\text{-}pa} : \mathrm{IK} \times \mathrm{DATA} \rightarrow \mathrm{PA} &&\\
\mathrm{REL} &= \{r_1, \ldots, r_n\} && \text{(Set of Business Logic Descriptions)}\\
\mathrm{EID} &= \mathrm{IK} \times \mathrm{VK} && \text{(Set of EffectIDs)}\\
\mathrm{CID} &= \mathrm{CL} \times \mathrm{IK} \times \mathrm{VK} && \text{(Set of CauseIDs)}\\
\mathrm{CL} &= \mathrm{PID} && \text{(Set of CauseLinks)}
\end{aligned}$$

$$\begin{aligned}
\mathrm{IN} &= \mathrm{A} \rightarrow \mathbb{P}(\mathrm{IK} \times \mathrm{DATA}) && \text{(Set of In Tables)}\\
\mathrm{ASSERTER} &= \mathrm{A} \rightarrow \mathrm{IK} \times \mathrm{VK} \rightarrow \mathrm{STR}_\perp \times \mathrm{OL}_\perp \times \mathrm{VL}_\perp \times \mathbb{P}(\mathrm{PA}) && \text{(Set of Asserting Actors)}\\
\mathrm{LOG} &= \mathrm{A} \rightarrow \mathrm{IK} \times \mathrm{VK} \rightarrow \mathrm{CHANGED}_\perp \times \mathrm{APS}_\perp && \text{(Set of Log Tables)}\\
\mathrm{QUEUE} &= \mathrm{A} \rightarrow Queue(\mathrm{R}) && \text{(Set of Record Queues)}\\
\mathrm{LC} &= \mathrm{A} \rightarrow \mathbb{N} && \text{(Sender's Local Counts)}\\
\mathrm{PSLIST} &= \mathrm{A} \rightarrow \mathbb{P}(\mathrm{PID}) && \text{(Set of Alternative Store Lists)}\\
\mathrm{STR} &= \{\mathsf{READY}, \mathsf{SEND}, \mathsf{SENT}, \mathsf{ACKED}, \mathsf{OK}\} && \text{(States of Interaction Record)}\\
\mathrm{CHANGED} &= \{\mathsf{TRUE}, \mathsf{FALSE}\} && \text{(Flags of using alternative PS))}\\
\mathrm{APS} &= \mathrm{PID} && \text{(Set of Alternative Stores Used)}
\end{aligned}$$

$$\begin{aligned}
\mathrm{TIMER} &= \mathrm{A} \rightarrow \mathrm{IK} \times \mathrm{VK} \rightarrow \mathrm{STATUS}_\perp \times \mathrm{TIMEOUT} && \text{(Set of Timers)}\\
\mathrm{STATUS} &= \{\mathsf{ENABLED}, \mathsf{DISABLED}\} && \text{(Set of Timer Statuses)}\\
\mathrm{TIMEOUT} &= \mathbb{N} && \text{(Set of Timeouts)}
\end{aligned}$$

$$\begin{aligned}
\mathrm{PS} &= \mathrm{PID} \rightarrow \mathrm{IK} \times \mathrm{VK} \rightarrow \mathrm{A}_\perp \times \mathrm{VL}_\perp \times \mathbb{P}(\mathrm{PA}) && \text{(Set of Provenance Stores)}
\end{aligned}$$

$$\begin{aligned}
\mathrm{C} &= \mathrm{CID} \rightarrow \mathrm{IK} \times \mathrm{VK} \rightarrow \mathrm{DESTPS}_\perp \times \mathrm{OL}_\perp && \text{(Set of Coordinators)}\\
\mathrm{UPDATE} &= \mathrm{CID} \rightarrow \mathrm{IK} \times \mathrm{VK} \rightarrow \mathrm{STATE}_\perp && \text{(Set of Update Tables)}\\
\mathrm{STATE} &= \{\mathsf{UPDATE}, \mathsf{SENT}, \mathsf{UPDATED}, \mathsf{F}\} && \text{(Set of Update States)}
\end{aligned}$$

$$\begin{aligned}
\mathrm{SC} &= \mathrm{IN} \times \mathrm{ASSERTER} \times \mathrm{QUEUE} \times \mathrm{LOG} \times \mathrm{LC} \times &&\\
&\quad \mathrm{TIMER}{\times}\mathrm{PS} \times \mathrm{C} \times \mathrm{UPDATE} \times \mathcal{K} && \text{(Set of Configurations)}
\end{aligned}$$

Characteristic Variables:
$a \in A$, $a_s \in \mathrm{SID}$, $a_r \in \mathrm{RID}$, $a_{ps} \in \mathrm{PID}$, $a_c \in \mathrm{CID}$, $m \in \mathcal{M}$, $k \in \mathcal{K}$, $d \in \mathrm{DATA}$, $\kappa \in \mathrm{IK}$, $v \in \mathrm{VK}$, $ol \in \mathrm{OL}$, $vl \in \mathrm{VL}$, $a_{dps} \in \mathrm{DESTPS}$, $pa \in \mathrm{PA}$, $pas \in \mathcal{P}(\mathrm{PA})$, $content \in \mathrm{CONTENT}$, $r \in \mathrm{REL}$, $cids \in \mathcal{P}(\mathrm{CID})$, $cl \in \mathrm{CL}$, $in\_T \in \mathrm{IN}$, $asserter\_T \in \mathrm{ASSERTER}$, $log\_T \in \mathrm{LOG}$, $queue\_T \in \mathrm{QUEUE}$, $lc \in \mathrm{LC}$, $psList \in \mathrm{PSLIST}$, $str \in \mathrm{STR}$, $changed \in \mathrm{CHANGED}$, $aps \in \mathrm{APS}$, $timer\_T \in \mathrm{TIMER}$, $status \in \mathrm{STATUS}$, $to \in \mathrm{TIMEOUT}$, $store\_T \in \mathrm{PS}$, $coord\_T \in \mathrm{C}$, $update\_T \in \mathrm{UPDATE}$, $c \in \mathrm{SC}$

Initial State of Configuration:
$c_i = \langle in\_T_i, asserter\_T_i, log\_T_i, queue\_T_i, lc_i, timer\_T_i, store\_T_i, coord\_T_i, update\_T_i, k_i \rangle$
where:
$asserter\_T_i = \lambda a\kappa v \cdot \langle \perp, \perp, \perp, \emptyset \rangle$, $log\_T_i = \lambda a\kappa v \cdot \langle \perp, \perp \rangle$, $queue\_T_i = \lambda a \cdot \emptyset$,
$lc_i = \lambda a \cdot 0$, $timer\_T_i = \lambda a\kappa \cdot \langle \perp, 0 \rangle$, $store\_T_i = \lambda a\kappa v \cdot \langle \perp, \perp, \emptyset \rangle$,
$coord\_T_i = \lambda a\kappa v \cdot \langle \perp, \perp \rangle$, $update\_T_i = \lambda a\kappa v \cdot \langle \perp \rangle$, $k_i = \lambda a_i a_j \cdot \emptyset$
$in\_T_i = \lambda a \cdot \emptyset$

Fig. 3. System State Space

The log table (log_T) maintains history information of using alternative stores in an asserting actor, used for updating causelinks. A flag ($changed \in$ CHANGED) is set to TRUE if an alternative store was used. The identifier of the final store from which an actor received acknowledgment is remembered in a field ($aps \in$ APS). So that an asserting actor knows which store recorded its p-assertions about an interaction.

After creating interaction records, an actor accumulates them in a local queue, modelled by the table ($queue\_T$), before shipping them to a provenance store. The FIFO property of the queue guarantees successful update of causelinks, detailed later. The notation (LC) defines a function mapping a sender identifier to a natural number so as to distinguish interactions between the sender and receiver. The sender needs to ensure that the natural number is locally unique on the sender side in each interaction. The list of alternative provenance stores are modelled by the set PSLIST, mapping an actor's identity to a set of store identities.

The timer table ($timer\_T$) models the timer used by asserting actors and update coordinators when waiting for acknowledgement messages. The timer's state ($status \in$ STATUS) indicates if the timer is enabled or disabled. A timeout ($to \in$ TIMEOUT) is a natural number, which counts down to zero after the timer is enabled.

PS and Coordinator State Space. The set PS models provenance stores, each containing a table ($store\_T$) indexed by a provenance store's identity. The table maps an interaction key and the view kind of the asserter that created and recorded p-assertions in the interaction to a tuple: the identity of the asserter, a viewlink and the set of p-assertions documenting the interaction. The set C models update coordinators. A update coordinator maintains repair request information in a table ($coord\_T$) and the states of updating a viewlink in another table ($update\_T \in$ UPDATE). We will further detail these tables when we describe the rules of a provenance store and update coordinator.

Given the state space, the ASM is described by an initial state and a set of transitions. A transition is the application of a rule to one configuration to achieve another configuration. Figure 3 contains the initial state ($c_i \in$ SC), which can be summarised as empty channels, empty tables and any local counters being initialised to zero in all actors. The ASM proceeds from this initial state through its execution by going through transitions that lead to new states. These transitions are defined below by the rules of the state machine.

State Machine Rules. The state machine rules are represented using the following notation.

```
$rule\_name(v_1, v_2, ...)$ :
$condition_1(v_1, v_2, ...) \wedge condition_2(v_1, v_2, ...) \wedge ...$
$\rightarrow$ {
    $pseudo\_statement_1$;
    ...
    $pseudo\_statement_n$;
  }
```

Rules are identified by their name and a number of parameters that the rule operates over. Any number of conditions must be met for a rule to fire. Once a rule's conditions are met, the rule fires. The execution of a rule is atomic, so that no other rule may interrupt or interleave with an executing rule. This maintains the consistency of the ASM. A new state is achieved after applying all the rule's pseudo-statements to the state that met the conditions of the rule.

We use *send* and *receive* and table update pseudo-statements. Informally, $send(m, a_1, a_2)$ inserts a message m into the communication channel from actor a_1 to actor a_2, and $receive(m, a_1, a_2)$ removes m from the channel. The table update operation puts a message into a table or changes content state in a table. We use the notation $table\_T$ to refer to any table in the system state space. Formally, *send*, *receive* and table update pseudo-statements act as state transformers and are defined as follows.

- If k is the set of message channels of a state $\langle \ldots, k\rangle$, then the expression $send(m, a_1, a_2)$ denotes the state $\langle \ldots, k'\rangle$, where[2] $k'(a_1, a_2) = k(a_1, a_2) \oplus m$, and $k'(a_i, a_j) = k(a_i, a_j), \forall(a_i, a_j) \neq (a_1, a_2)$.
- If k is the set of message channels of a state $\langle \ldots, k\rangle$, then the expression $receive(m, a_1, a_2)$ denotes the state $\langle \ldots, k'\rangle$, where $k'(a_1, a_2) = k(a_1, a_2) \ominus m$, and $k'(a_i, a_j) = k(a_i, a_j), \forall(a_i, a_j) \neq (a_1, a_2)$.
- If $table\_T$ is a component of state $\langle \ldots, table\_T, \ldots\rangle$, then the expression $table\_T(\ldots).y := V$ denotes the state $\langle \ldots, table\_T', \ldots\rangle$, where $table\_T'(\ldots).x = table\_T(\ldots).x$ if $x \neq y$, and $table\_T'(\ldots).y := V$.

To manipulate an asserting actor's queue, which is used for accumulating interaction records, we define the following operations: $head(q)$, $enqueue(m, q)$ and $dequeue(q)$.

- The expression $head(q)$ returns the head element of queue q.
- The expression $enqueue(m, q)$ denotes $q := q \| m$, which means m is added at the tail of queue q.
- The expression $dequeue(q)$ denotes $q := tail(q)$, which means the head of queue q is removed.

For convenience, we use notation $a \leftarrow b$ to bind a local variable a to a value b. We then define an assignment operator $:=$ for tables. It can assign a value to a field of a table, or assign a tuple to a table as in the following example. In this example, the second field of $asserter\_T(a, \kappa, v)$, i.e., the ownlink ol, is not assigned when $*$ is present.

$$asserter\_T(a, \kappa, v) := \langle \mathsf{OK}, *, PS_2, pas\rangle \equiv \begin{cases} asserter\_T(a, \kappa, v).str := \mathsf{OK} \\ asserter\_T(a, \kappa, v).vl := PS_2 \\ asserter\_T(a, \kappa, v).pas := pas \end{cases}$$

[2] We use the operators $\oplus$ and $\ominus$ to denote union and difference on bags.

Having defined the system state space and ASM rules, we now introduce the rules for asserters (the senders and receivers), provenance stores and update coordinators. These rules precisely define these actors' internal behaviour.

4.2 Asserter Rules in Exchanging Phase

An asserting actor's behaviour can be summarised as two phases: Exchanging and Recording. We firstly describe the Exchanging phase and then introduce the rules of the Recording phase in Section 4.3.

The sender and receiver in an interaction have different rules in the Exchanging phase (Figure 4 and Figure 5). The sender exchanges to the receiver an application message app including application data (d), an interaction key (κ) and the sender's ownlink (i.e., the receiver's viewlink, vl). After receiving an app message, the receiver adds κ and d into table (in_T). Both actor document the exchange of app and an interaction record message is then produced and accumulated in a queue ($queue\_T$). This buffering of interaction records is designed to meet *Transparent Recording* and *Efficient Recording* requirements. It reduces the performance penalty upon the application by allowing the actor to send interaction records when convenient. An asserting actor also initialises several tables, used in the Recording phase.

```
$send\_app(a_s, a_r, a_{ps}, vl, d, r)$ :
//triggered when d, produced by a function
//described by r, is to be sent by $a_s$ to $a_r$,
//and when the viewlink, vl, is available.
$\rightarrow \{$
    $\kappa \leftarrow newIdentifier(a_s, a_r)$;
    $send(\mathsf{app}(d, \kappa, a_{ps}), a_s, a_r)$;
    $pas \leftarrow createPA(a_s, \kappa, d, r)$;
    $enqueue(\mathsf{record}(\kappa, S, a_s, vl, pas), queue\_T(a_s))$;
    $asserter\_T(a_s, \kappa, S) := \langle \mathsf{READY}, a_{ps}, vl, pas \rangle$;
    $log\_T(a_s, \kappa, S) := \langle \mathsf{FALSE}, \perp \rangle$ ;
$\}$
```

Fig. 4. The Sender's Rules (Exchanging Phase)

```
$receive\_app(a_s, a_r, a_{ps}, d, \kappa, vl)$ :
    $\mathsf{app}(d, \kappa, vl) \in k(a_s, a_r)$
$\rightarrow \{$
    $receive(\mathsf{app}(d, \kappa, vl), a_s, a_r)$;
    $in\_T(a_r) := in\_T(a_r) \oplus \langle \kappa, d \rangle$;
    $pas \leftarrow createPA(a_s, \kappa, d, \perp)$;
    $enqueue(\mathsf{record}(\kappa, R, a_r, vl, pas), queue\_T(a_r))$;
    $asserter\_T(a_r, \kappa, R) := \langle \mathsf{READY}, a_{ps}, vl, pas \rangle$ ;
    $log\_T(a_r, \kappa, R) := \langle \mathsf{FALSE}, \perp \rangle$ ;
    // business logic
$\}$
```

Fig. 5. The Receiver's Rules (Exchanging Phase)

The function $newIdentifier(a_s, a_r)$ creates a globally unique interaction key, as defined by the following pseudo function. This function requires that senders are responsible for creating interaction keys. This function takes the identities of the sender and the receiver as inputs. It then obtains the local counter of the sender and increases it by one. Finally, a new interaction key using the two actor identities and the local counter is constructed and returned.

```
Definition
$newIdentifier : \mathrm{SID} \times \mathrm{RID} \rightarrow \mathrm{IK}$
$newIdentifier(a_s, a_r)$ :
    $lc(a_s) := lc(a_s) + 1$;
    $return\ \langle a_s, a_r, lc(a_s) \rangle$ ;
```

In order to define function $createPA(a, \kappa, d, r)$, we firstly define a function $cause(a, d, r)$. It takes an actor identity (a), application data (d), and a business description (r) as input and finds interaction keys of all causes that are related to the production of d.

```
Definition
$cause : \mathrm{A} \times \mathrm{DATA} \times \mathrm{REL} \rightarrow \mathbb{P}\,(\mathrm{IK})$
$cause(a, d, r)$ :
    $let\ f_r\ be\ a\ function\ described\ by\ r,\ such\ that\ f_r(args) = d,$
    $where\ args = \{\langle \kappa, d' \rangle \mid \langle \kappa, d' \rangle \in in\_T(a)\};$
    $return\ \{\kappa \mid \langle \kappa, d' \rangle \in args\};$
```

Recall that in Figure 5, the data received from application messages is stored in table in_T. In $cause(a, d, r)$, we assume there exists a function f_r that takes some data d' from in_T as input and produces a result data d. Then $cause(a, d, r)$ returns the keys of interactions where all the input data is received.

The $createPA(a, \kappa, d, r)$ function is defined as follows. It takes an actor identity (a), an interaction key (κ), application data (d), and a business logic description (r) to create a set of p-assertions documenting the interaction (indexed by κ) in which d is transferred.

```
Definition
$createPA : \mathrm{A} \times \mathrm{IK} \times \mathrm{DATA} \times \mathrm{REL} \rightarrow \mathbb{P}\,(\mathrm{PA})$
$createPA(a, \kappa, d, r)$ :
    $pas \leftarrow if\ \ r = \perp$
                    $\{\text{i-pa}(\kappa, d)\}$ ;
                    $\{\text{i-pa}(\kappa, d), \text{rel-pa}(r, \langle \kappa, S \rangle, cids)\}$ ,
$where\ cids = \{\langle cl, \kappa', R \rangle \mid \kappa' \in cause(a, d, r)\ and\ cl = asserter\_T(a, \kappa', R).ol\};$
  $return\ pas;$
```

In $createPA(a, \kappa, d, r)$, the created p-assertions must at least include an interaction p-assertion documenting the exchange of an application message that contains κ and d. If d is the consequence of receiving other messages, i.e., $r \neq \perp$, then the sender must make a relationship p-assertion[3] to capture the causal connections between these messages. Function $cause(a, d, r)$ is used here to find keys of cause interactions when creating a relationship p-assertion. An asserting actor may create other application dependent p-assertions, which are not shown in the definition.

4.3 Asserter Rules in Recording Phase

In Recording phase, an asserting actor sends queued record messages to a provenance store and takes remedial actions in response to timeouts. To facilitate presentation, we assume each asserting actor employs a Recording Manager (RM), which monitors the actor's queue and submits record messages to a provenance store. The behaviour of RM is specified in Figure 6 and summarised now.

[3] A relationship p-assertion is always created and recorded in the context of its effect interaction.

```
$pre\_check(a, \kappa, v, vl, pas)$ :
    $queue\_T(a) \neq \emptyset \wedge \mathsf{record}(\kappa, v, a, vl, pas) = head(queue\_T(a)) \wedge asserter\_T(a, \kappa, v).str = \mathsf{READY}$
$\rightarrow$ {
      for $each\ pa \in pas,\ such\ that\ pa = \mathsf{rel\text{-}pa}(r', \langle \kappa, v \rangle, cids')$
            do for $each\ cid \in cids'$
                  do $\langle cl', \kappa', v' \rangle \leftarrow cid$;
                      $if\ (log\_T(a, \kappa', v').changed),\ then$
                              $cid' \leftarrow \langle log\_T(a, \kappa', v').aps, \kappa', v' \rangle$;
                              $cids'' \leftarrow cids' \ominus cid \oplus cid'$;
                $pa' \leftarrow \mathsf{rel\text{-}pa}(r', \langle \kappa, v \rangle, cids'')$;
                $pas' \leftarrow pas \ominus pa \oplus pa'$;
      $if\ pas' \neq \perp$
            $asserter\_T(a, \kappa, v) := \langle *, *, *, pas' \rangle$;
      $asserter\_T(a, \kappa, v).str := \mathsf{SEND}$;
    }

$send\_record(a, \kappa, v, vl, pas, to)$ :
    $queue\_T(a) \neq \emptyset \wedge \mathsf{record}(\kappa, v, a, vl, pas) = head(queue\_T(a)) \wedge asserter\_T(a, \kappa, v).str = \mathsf{SEND}$
$\rightarrow$ {
      $a_{ps} \leftarrow asserter\_T(a, \kappa, v).ol$;
      $send(\mathsf{record}(\kappa, v, a, vl, pas), a, a_{ps})$;
      $timer\_T(a, \kappa, v) := \langle \mathsf{ENABLED}, to \rangle$ ;
      $asserter\_T(a, \kappa, v).str := \mathsf{SENT}$;
    }

$timer\_click(a, \kappa, v)$ :
    $timer\_T(a, \kappa, v).status = \mathsf{ENABLED}$
$\rightarrow$ {
      $timer\_T(a, \kappa, v).to := timer\_T(a, \kappa, v).to - 1$;
    }

$timeout\_ack(a, \kappa, v)$ :
    $timer\_T(a, \kappa, v).status = \mathsf{ENABLED} \wedge timer\_T(a, \kappa).to \leq 0$
$\rightarrow$ {
      $a'_{ps} \leftarrow random(psList(a))$;
      $log\_T(a, \kappa, v).changed := \mathsf{TRUE}$;
      $timer\_T(a, \kappa, v) := \langle \mathsf{DISABLED}, 0 \rangle$ ;
      $asserter\_T(a, \kappa, v) := \langle \mathsf{SEND}, a'_{ps}, *, * \rangle$;
    }

$receive\_ack(a, a_{ps}, \kappa, v)$ :
    $\mathsf{ack}(\kappa, v) \in k(a, a_{ps})$
$\rightarrow$ {
      $receive(\mathsf{ack}(\kappa, v), a, a_{ps})$;
      $ol \leftarrow asserter\_T(a, \kappa, v).ol$;
      $if\ (timer\_T(a, \kappa, v).to > 0 \wedge a_{ps} = ol \wedge asserter\_T(a, \kappa, v).str = \mathsf{SENT}),\ then$
            $dequeue(queue\_T(a))$;
            $timer\_T(a, \kappa, v) := \langle \mathsf{DISABLED}, 0 \rangle$ ;
            $asserter\_T(a, \kappa, v).str := \mathsf{ACKED}$;
    }

$post\_check(a, a_c, \kappa, v)$ :
    $asserter\_T(a, \kappa, v).str = \mathsf{ACKED}$
$\rightarrow$ {
      $if\ (log\_T(a, \kappa, v).changed),\ then$
            $a_{ps} \leftarrow asserter\_T(a, \kappa, v).ol$;
            $a_{dps} \leftarrow asserter\_T(a, \kappa, v).vl$;
            $send(\mathsf{repair}(\kappa, v, a_{dps}, a_{ps}), a, a_c)$;
            $log\_T(a, \kappa, v).aps := a_{ps}$;
      $asserter\_T(a, \kappa, v).str := \mathsf{OK}$;
    }
```

Fig. 6. Asserter rules in Recording phase

- *Updating causelinks.* Given a record message from the queue ($queue\_T(a)$), RM checks and updates causelinks in all relationship p-assertions included in the message (rule *pre_check*). A log table (log_T) maintains a history of the use of alternative provenance stores for each interaction. If the log table shows that an alternative store was used to record p-assertions about a cause interaction, then the corresponding causelink is updated.
- *Submitting a record message.* RM sends a record message to a provenance store and sets timeout when waiting for an ack message (rule *send_record*).
- *Resubmitting a record message.* If RM does not receive an ack when the timeout expires (rule *timeout_ack*), then it infers that failures may have occurred. In this case, RM may resend the record to the same store or use an alternative store if retry attempts to the old store also failed. In order to simplify rules, we do not formalise resending messages to a same provenance store; instead, a new store is selected once a timeout expires. The function $random(psList(a))$ returns an alternative store's identity, selected from a list of candidates. There can be various ways of selecting a store from a list of stores. Here we randomly select one to use.

  Only after an ack is received, can RM eliminate the acknowledged record from the queue (rule *receive_ack*). Checking the state *str* as well as the identifier of the provenance store from which an ack is received help detect duplicate acknowledgements, preserving the correcness of the protocol (rule *receive_ack*).
- *Requesting to update viewlinks.* If an alternative store was used to record a record message, the actor's ownlink known to the opposite actor in an interaction becomes invalid, since the actor's store has changed. Therefore, RM requests a update coordinator to update the opposite actor's viewlink by sending a repair message (rule *post_check*). We note that for a given interaction, an asserting actor at most sends one repair request, which minimises the overhead of taking remedial actions.
- *Updating log table.* If an alternative store was used to record a record message, RM sets $log\_T(a, \kappa, v).changed$ to TRUE (rule *timeout_ack*). After a record message is successfully recorded in a provenance store, RM remembers the provenance store's identity in the log table if $log\_T(a, \kappa, v).changed$ is TRUE (rule *post_check*). This information is to be used for updating causelinks as described above.

We note that the FIFO property of the queue guarantees successful update of causelinks. This is because rule *send_app* and rule *receive_app* enforce that an actor always makes p-assertions about a cause interaction, i.e., where it receives a message, before an effect interaction, i.e., where it sends another message as consequence of received messages. This implies that the record messages about cause interactions are always placed into the queue before that about the effect interaction. Therefore, by monitoring the use of alternative stores when sending record messages, causelinks can be updated successfully. Although current modelling indicates that there is only one queue per asserting actor, which is highly sequential, it can be relaxed by adding a process identifier to $queue\_T(a)$. Then, each process that an actor participates in can utilise a queue, which enables parallel recording.

4.4 Provenance Store Rules

Figure 7 gives provenance store's rules. A provenance store replies an ack message only *after* it has processed a record message (rule *receive_record*). A store checks if p-assertions about a given interaction exists before processing a record message. This prevents resubmitted record messages from being recorded multiple times.

```
$receive\_record(a, a_{ps}, \kappa, v, vl, pas)$ :
  $\mathsf{record}(\kappa, v, a, vl, pas) \in k(a, a_{ps})$
$\rightarrow$ {
    $receive(\mathsf{record}(\kappa, v, a, vl, pas), a, a_{ps})$;
    $if\ (store\_T(a_{ps}, \kappa, v).pas = \emptyset),\ then$
        $store\_T(a_{ps}, \kappa, v) := \langle a, *, pas \rangle$;
    $if\ (store\_T(a_{ps}, \kappa, v).vl = \bot),\ then$
        $store\_T(a_{ps}, \kappa, v).vl := vl$;
    $send(\mathsf{ack}(\kappa, v), a_{ps}, a)$;
  }

$receive\_update(a_{ps}, a_c, \kappa, v, \overline{v}, ol)$ :
  $\mathsf{update}(\kappa, \overline{v}, ol) \in k(a_c, a_{ps})$
$\rightarrow$ {
    $receive(\mathsf{update}(\kappa, \overline{v}, ol), a_c, a_{ps})$;
    $store\_T(a_{ps}, \kappa, v).vl := ol$;
    $send(\mathsf{uack}(\kappa, \overline{v}), a_{ps}, a_c)$;
  }
```

Fig. 7. Provenance Store rules

Since a provenance store may receive an update and a record message related to a same interaction in any order, to achieve requirement *Viewlink Accuracy*, the viewlink obtained from record must NOT overwrite any existing one which may come from an update.

The notation $\overline{v}$ in rule *receive_update* stands for the opposite view in an interaction. For example, if v is the view of the sender, then $\overline{v}$ represents the view of the receiver.

An actor selects an alternative store to record p-assertions if an ack is not received within a timeout. However, it may be the case that the original store still receives and records those p-assertions. This may lead to duplicate information in several stores though, it does not affect the correctness of the protocol, since only the p-assertions successfully acknowledged by an ack can be retrieved using the links updated by the protocol.

4.5 Coordinator Rules

The update coordinator's rules are shown in Figure 8. Upon receiving a repair request (rule *receive_repair*), if there exists request information from the opposite view with regard to the same interaction, which means the coordinator has received a repair message from the other actor, then the coordinator replaces one actor's destination store with the other's ownlink, thus making each actor's destination store correct. Then the update coordinator dispatches two update messages to their respective new destination stores by setting update status to UPDATE (rule *send_update*).

Since a crashing provenance store can be restarted, resending update messages to a same provenance store can be eventually successful (rule *timeout_uack*

```
$receive\_repair(a_{ps}, a_{dps}, a_c, \kappa, v, \overline{v}, ol)$ :
  $\mathsf{repair}(\kappa, v, a_{dps}, ol) \in k(a_{ps}, a_c)$
$\rightarrow$ {
    $receive(\mathsf{repair}(\kappa, v, a_{dps}, ol), a_{ps}, a_c)$;
    $if\ (coord\_T(a_c, \kappa, v) = \perp),\ then$
        $coord\_T(a_c, \kappa, v) := \langle a_{dps}, ol\rangle$;
        $update\_T(a_c, \kappa, v) := \mathsf{UPDATE}$;
        $if\ (coord\_T(a_c, \kappa, \overline{v}) \neq \perp),\ then$
            $a'_{dps} \leftarrow coord\_T(a_c, \kappa, \overline{v}).ol$;
            $coord\_T(a_c, \kappa, v) := \langle a'_{dps}, *\rangle$;
            $coord\_T(a_c, \kappa, \overline{v}) := \langle ol, *\rangle$;
            $update\_T(a_c, \kappa, \overline{v}) := \mathsf{UPDATE}$;
  }
```

```
$send\_update(a_c, \kappa, v, to)$ :
  $update\_T(a_c, \kappa, v) = \mathsf{UPDATE}$
$\rightarrow$ {
    $\langle a_{dps}, ol\rangle \leftarrow coord\_T(a_c, \kappa, v)$;
    $send(\mathsf{update}(\kappa, v, ol), a_c, a_{dps})$;
    $timer\_T(a_c, \kappa, v) := \langle \mathsf{ENABLED}, to\rangle$ ;
    $update\_T(a_c, \kappa, v) := \mathsf{SENT}$;
  }
```

```
$timer\_click(a_c, \kappa, v)$ :
  $timer\_T(a_c, \kappa, v).status = \mathsf{ENABLED}$
$\rightarrow$ {
    $timer\_T(a_c, \kappa, v).to := timer\_T(a_c, \kappa, v).to - 1$;
  }
```

```
$timeout\_uack(a_c, \kappa, v)$ :
  $timer\_T(a_c, \kappa, v).status = \mathsf{ENABLED} \wedge$
  $timer\_T(a_c, \kappa, v).to \leq 0$
$\rightarrow$ {
    $update\_T(a_c, \kappa, v) := \mathsf{UPDATE}$;
    $timer\_T(a_c, \kappa, v) := \langle \mathsf{DISABLED}, 0\rangle$ ;
  }
```

```
$receive\_uack(a_{ps}, a_c, \kappa, v)$ :
  $\mathsf{uack}(\kappa, v) \in k(a_{ps}, a_c)$
$\rightarrow$ {
    $receive(\mathsf{uack}(\kappa, v), a_{ps}, a_c)$;
    $if\ (timer\_T(a_c, \kappa, v).to > 0) then$
        $if\ (a_{ps} = coord\_T(a_c, \kappa, v).a_{dps}),\ then$
            $timer\_T(a_c, \kappa, v) := \langle \mathsf{DISABLED}, 0\rangle$ ;
            $update\_T(a_c, \kappa, v) := \mathsf{UPDATED}$;
  }
```

Fig. 8. Coordinator rules

sets $update\_T(a_c, \kappa, v)$ to UPDATE, which will resend update message in rule *send_update.*). This ensures that all requested viewlinks in provenace stores can be updated.

In the current design, we do not specify removing request information maintained in an update coordinator. Request information with regard to an interaction can only be eliminated *after* the coordinator *successfully* updates the provenance store in *each view* of the interaction. If there exists request information for only one view, then the coordinator cannot delete it since it may receive another repair request from the other view. Given that an actor sends out a repair message for an interaction within finite time (due to the use of timeouts in Figure 6), the coordinator can remove any request information with corresponding update status being UPDATED after a reasonably long period of time since the information is recorded. As illustrated above, request information with an update status F cannot be removed.

5 Protocol Analysis

Based on the ASM above, we now analyse F_PReP. The requirements *Guaranteed Recording*, *Viewlink Accuracy* and *CauseLink Accuracy*, identified in Section 2.2, are concerned with the protocol's correctness. We prove that the three requirements are satisfied when the protocol terminates in each interaction. Given that a process consists of a set of interactions, if the protocol can ensure that for each interaction, the three requirements are supported, then the documentation of the whole process is guaranteed to be recorded and retrievable. We have proved the protocol terminates under the assumptions stated in Section 3.1. We now formalise the three requirements as properties and outline the proof of these properties.

Theorem 1 (Guaranteed Recording). *When the protocol terminates, the documentation produced by each asserting actor about the interaction is recorded in provenance stores.*
For any reachable configuration c and for any a, κ, v, the following implication holds when the ASM terminates:

If $asserter\_T(a, \kappa, v).str \neq \bot$, *then*

$$store\_T(a_{ps}, \kappa, v) = \langle a, vl, asserter\_T(a, \kappa, v).pas \rangle,$$

such that $vl \neq \bot$ *and* $a_{ps} = asserter\_T(a, \kappa, v).ol$. □

Theorem 2 (Viewlink Accuracy). *When the protocol terminates, each asserter's viewlink of an interaction is accurate in its provenance store. The viewlink points to the store where the other actor in the interaction recorded p-assertions about the same interaction.*
For any a, a', κ, v, then the following implication holds when the ASM terminates:
if $asserter\_T(a, \kappa, v).str \neq \bot$, *then*

$$store\_T(a_{ps}, \kappa, v).vl = asserter\_T(a', \kappa, \overline{v}).ol,$$

such that $a_{ps} = asserter\_T(a, \kappa, S).ol$. □

Theorem 3 (Causelink Accuracy). *When the protocol terminates, an asserter's causelinks are accurate in its provenance store. Each points to the store where p-assertions about the corresponding cause interaction are recorded.*
For any a, κ, v, then the following must hold when the protocol terminates:

if $asserter\_T(a, \kappa, v).str \neq \bot$, *then*

$$\begin{aligned} &\textit{for any } pa \in store\_T(a_{ps}, \kappa, v).pas, \textit{ such that } pa = \textsf{rel-pa}(rel, \langle \kappa, v \rangle, cids), \\ &\quad \textit{for any } c \in cids, \textit{ let } \langle cl', \kappa', v' \rangle = c, \\ &\qquad cl' = asserter\_T(a, \kappa', v').ol. \end{aligned}$$

such that $a_{ps} = asserter\_T(a, \kappa, v).ol$. □

Due to space restriction, we now outline our proof of these properties. Given an arbitrary valid configuration of the ASM, our proofs typically proceed by induction on the length of the transitions that lead to the configuration, and by a case analysis on the kind of transitions. We show that a property is true in the initial configuration of the machine and remains true for every possible transition. This kind of proof is systematic, less error prone and avoids the complications of temporal reasoning.

6 Related Work

Much research has been seen to support recording process documentation, such as Chimera [10], myGrid [26], Karma [24], Kepler [3]. All these systems rely on their execution environment or specific technologies. The drawback is that the recorded documentation lacks interoperability and hence cannot be shared by different organisations. To promote interoperability, Groth et al. [15] proposed an application and technology independent approach to modelling process documentation in the context of SOAs and developed a generic recording protocol, PReP. All the surveyed systems however do not deal with failures. F_PReP preserves the application and technology independent nature and provides well-defined behaviour while recording documentation in the case of failures.

Redundancy has been widely used to provide fault-tolerance for distributed systems [4]. It involves replicating data or system functionalities, and repeating messages or operations. We adopt the redundancy mechanism in our work, e.g., replicating update coordinators and retransmitting messages. Provenance stores are not suitable to be replicated due to the complexity and significant cost of replicating documentation as explained before.

Atomic transactions typically requires all-or-nothing property to maintain system consistency [12]. It can be an alternative solution to our work. We now discuss a scenario where atomic transaction is applied. We assume that an asserting actor and its provenance store are the two participants in a transaction of recording p-assertions. If the provenance store fails, the actor is notified that the transaction is aborted. Then the actor can select another store to use until the transaction is complete. In this case, the use of atomic transaction provides similar functionality as the remedial actions in our protocol, i.e., selecting an alternative store upon an expired timeout. This approach however is too complicated to be adopted by the fact that each interaction leads to two transactions (sender/receiver).

Formal methods are mathematically-based techniques for the specification, development and verification of software and hardware systems. There are three rigorous methods, Abstract State Machines (ASM) [17], B[2] and Z[1], that share a common conceptual foundation and are widely used in both academia and industry for the design and analysis of hardware and software systems.

Applying formal methods to the design and reasoning of fault-tolerance has been studied in distributed systems, e.g., distributed database systems[25], control systems[18], and mobile agent systems[19]. The ASM notation we adopt has been used previously to describe a fault-tolerant directory service for mobile agents[21] and PReP. Our proof follows a systematic procedure based on mathematical induction. While done by hand, we believe it is sufficient to provide confidence that the protocol does conform to the properties in Section 5. Previous experience has shown that the ASM formalism is suitable for mechanical proof derivations, and several algorithms[21] have been carried out using a Coq theorem prover[5].

7 Conclusion

In this paper, we have presented a generic protocol, F_PReP, for recording process documentation in the presence of failures. By deriving PReP, F_PReP not only keeps the generic nature, but also guarantees that process documentation is recorded in the presence of failures. Also, it enables the retrievability of distributed documentation in large scale distributed environments where failures may occur. The protocol is systematically designed and meets the requirements, identified in Section 2, under the assumptions we state on failures.

Our ASM-based formalisation provides a precise and implementation independent means of specifying the protocol. Firstly, it sketches the essence of the protocol and accurately defines required actor's behaviour with unnecessary message fields or messages removed. Secondly, it promotes a rigorous design of the protocol and helps us better understanding the complex behaviour of actors in the presence of failures. With such a formal description, we have successfully identified several deficiencies in the early design of the protocol. Thirdly, the code-like specification is independent of any given programming language or implementation. This enables our protocol to be implemented using different languages and technologies. In summary, the use of a formal notation has significantly improved the design of F_PReP.

F_PReP has been implemented in Java and integrated into a client side p-assertion recording library developed by the University of Southampton. Its performance has been evaluated and the result reveals that it introduces acceptable overhead [7]. We are currently investigating how to create process documentation when an application has its own fault tolerance schemes to tolerate application level failures. In future work, we plan to make use of the process documentation recorded in the presence of failures to diagnose failures.

References

1. Abrial, J., Schuman, S., Meyer, B.: A specification language. On the Construction of Programs, 343–410 (1980)
2. Abrial, J.R.: The B-Book. Cambridge University Press, Cambridge (1996)
3. Altintas, I., Barney, O., Jaeger-Frank, E.: Provenance collection support in the kepler scientific workflow system. In: Moreau, L., Foster, I. (eds.) IPAW 2006. LNCS, vol. 4145, pp. 118–132. Springer, Heidelberg (2006)
4. Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C.E.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Sec. Comput. 1(1), 11–33 (2004)
5. Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J., Gimenez, E., Herbelin, H., Huet, G., Munoz, C., Murthy, C., Parent, C., Paulin-Mohring, C., Saibi, A., Werner, B.: The Coq proof assistant reference manual: Version 6.1. Technical Report RT-0203 (1997)
6. Butler, M., Jones, C.B., Romanovsky, A., Troubitsyna, E. (eds.): Rigorous Development of Complex Fault-Tolerant Systems. LNCS, vol. 4157, pp. 241–260. Springer, Heidelberg (2006)
7. Chen, Z., Moreau, L.: Implementation and evaluation of a protocol for recording process documentation in the presence of failures. In: Freire, J., Koop, D., Moreau, L. (eds.) IPAW 2008. LNCS, vol. 5272, pp. 92–105. Springer, Heidelberg (2008)
8. Coulouris, G., Dollimore, J., Kindberg, T.: Distributed Systems: Concepts and Design, 4th edn. Addison-Wesley, Reading (2005)

9. Deelman, E., et al.: Managing large-scale workflow execution from resource provisioning to provenance tracking: The cybershake example. In: e-Science, p. 14. IEEE Computer Society, Los Alamitos (2006)
10. Foster, I.T., Vöckler, J., Wilde, M., Zhao, Y.: The virtual data grid: A new model and architecture for data-intensive collaboration. In: CIDR (2003)
11. Gagliardi, F., Jones, B., Grey, F., Bgin, M.E., Heikkurinen, M.: Building an infrastructure for scientific grid computing: Status and goals of the egee project. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 363(1833), 1729–1742 (2005)
12. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers, San Francisco (1993)
13. Groth, P., Miles, S., Fang, W., Wong, S.C., Zauner, K.-P., Moreau, L.: Recording and using provenance in a protein compressibility experiment. In: 14th IEEE International Symposium on HPDC 2005: Proceedings of the High Performance Distributed Computing, 2005. HPDC-14. Proceedings, pp. 201–208. IEEE Computer Society, Washington (2005)
14. Groth, P.: The origin of data: Enabling the determination of provenance in multi-institutional scientific systems through the documentation of processes. Phd thesis, University of Southampton (2007)
15. Groth, P., Jiang, S., Miles, S., Munroe, S., Tan, V., Tsasakou, S., Moreau, L.: An architecture for provenance systems. Technical Report D3.1.1, University of Southampton (February 2006)
16. Groth, P., Luck, M., Moreau, L.: A protocol for recording provenance in service-oriented grids. In: Higashino, T. (ed.) OPODIS 2004. LNCS, vol. 3544, pp. 124–139. Springer, Heidelberg (2005)
17. Gurevich, Y., Kutter, P.W., Odersky, M., Thiele, L. (eds.): ASM 2000. LNCS, vol. 1912. Springer, Heidelberg (2000)
18. Laibinis, L., Troubitsyna, E.: Refinement of fault tolerant control systems in B. In: Heisel, M., Liggesmeyer, P., Wittmann, S. (eds.) SAFECOMP 2004. LNCS, vol. 3219, pp. 254–268. Springer, Heidelberg (2004)
19. Laibinis, L., Troubitsyna, E., Iliasov, A., Romanovsky, A.: Rigorous development of fault-tolerant agent systems. In: Butler, et al. (eds.) [6], pp. 241–260
20. Miles, S., Groth, P., Branco, M., Moreau, L.: The requirements of using provenance in e-science experiments. Journal of Grid Computing 5(1), 1–25 (2007)
21. Moreau, L.: A Fault-Tolerant Directory Service for Mobile Agents based on Forwarding Pointers. In: The 17th ACM Symposium on Applied Computing (SAC 2002) — Track on Agents, Interactions, Mobility and Systems, Madrid, Spain, pp. 93–100 (March 2002)
22. Moreau, L., Dickman, P., Jones, R.: Birrell's Distributed Reference Listing Revisited. ACM Transactions on Programming Languages and Systems (TOPLAS) 27(4), 52 (2005)
23. Ozsu, M.T., Valduriez, P.: Principles of Distributed Database Systems, 2nd edn. Prentice Hall, Englewood Cliffs (1999)
24. Simmhan, Y.L., et al.: Performance evaluation of the karma provenance framework for scientific workflows. In: Moreau, L., Foster, I. (eds.) IPAW 2006. LNCS, vol. 4145. Springer, Heidelberg (2006)
25. Yadav, D., Butler, M.: Rigorous design of fault-tolerant transactions for replicated database systems using event b. In: Butler, et al. (eds.) [6], pp. 343–363.
26. Zhao, J., Wroe, C., Goble, C.A., Stevens, R., Quan, D., Greenwood, R.M.: Using semantic web technologies for representing e-science provenance. In: International Semantic Web Conference, pp. 92–106 (2004)

DREP: A Requirements Engineering Process for Dependable Reactive Systems

Sadaf Mustafiz and Jörg Kienzle

School of Computer Science, McGill University
Montreal, Quebec, Canada
sadaf@cs.mcgill.ca, joerg.kienzle@mcgill.ca

Abstract. Discovering and documenting potential abnormal situations and irregular user behavior that can interrupt normal system interaction is of tremendous importance in the context of dependable systems development. Exceptions that are not identified during requirements elicitation might eventually lead to an incomplete system specification during analysis, and ultimately to an implementation that lacks certain functionality, or even behaves in an unreliable way. This paper presents a requirements engineering process, DREP, that systematically guides the developer to consider reliability and safety concerns of reactive systems. After the discovery of normal system behavior by means of use cases, the developer is lead to explore exceptional situations arising in the environment that change the context in which the system operates and service-related exceptional situations that threaten to fail user goals. The process requires the developer to specify means that detect such situations, and to define the recovery measures that attempt to put the system in a reliable and safe state. The process is iterative, and refinements are carried out, if necessary, to achieve desired quality levels. To conclude the requirements phase, an extended use case diagram summarizes the normal interactions, exceptions, handlers and their relationships. The proposed process is demonstrated with the 407 Express Toll Route System case study.

1 Introduction

Complex computer systems are increasingly built for highly critical tasks, from military and aerospace domains to industrial and commercials areas. Failures of such systems may have severe consequences ranging from loss of business opportunities, physical damage, to loss of human lives. Systems with such responsibilities should be highly *dependable*.

On the software developer's part, this involves acknowledging that many exceptional situations may arise during the execution of an application, and providing measures to handle such situations. When using a standard software development process to develop systems, there is no guarantee that such situations are considered during the development. Whether the system can handle these situations or not depends highly on the imagination and experience of the developers. In addition, even if the application can actually deal with these special sitrutations, the particular way that the developer chose to address that situation might not be the one that a typical user of the system would expect if it was not explicitly agreed upon and documented in the requirements. As a

M. Butler et al. (Eds.): Fault Tolerance, LNCS 5454, pp. 220–250, 2009.
© Springer-Verlag Berlin Heidelberg 2009

result, the final application might not function correctly in all possible situations or react in unexpected ways. This can at best annoy or confuse the user, but can also have more severe repercussions.

When developing dependable systems, nothing should be left to chance. Following the idea of integrating exception handling into the software life cycle [1,2], this paper describes an extension to standard *use case*-based requirements elicitation that leads the developers to consider dependabaility issues early on. Our approach focusses in particular on reliability and safety concerns. We believe that thinking about behaviour or events that affect the reliability or safety of the system has to start at the requirements phase, because it is up to the stakeholders of the system to decide how they expect the system to react to exceptional situations. Only with exhaustive and detailed user feedback is it possible to discover and then specify the complete system behavior in a subsequent analysis phase, and decide on the need for employing fault masking and fault tolerance techniques for achieving run-time dependability during design.

This paper describes *a use case-driven requirements engineering and analysis process*, DREP, that leads the developers to consider dependability issues early on during software development. Our approach focuses in particular on *reliability* and *safety* concerns. This paper focuses on the process itself, and hence complements the papers [3,4], which describe the exceptional use case notation used in the process, and papers [5,6], which describe our model-driven approach on mapping exceptional use cases to DA-Charts and Markov chains to perform dependability analysis.

The paper is structured as follows: Section 2 introduces the dependability attributes and gives a brief overview of exceptions, handlers, and use cases. Section 3 describes our proposed process, and the ideas are illustrated by means of the 407 highway toll route case study in Section 4. Section 5 presents DREP in the context of model-driven engineering. Section 6 describes an academic experiment we conducted with 40 software engineering graduate students to validate the applicability and effectiveness of our proposed process. Section 7 presents related work in this area and Section 8 discusses future work and draws some conclusions.

2 Background

2.1 Requirements Engineering

Requirements engineering can be categorized as *requirements development* and *requirements management*. Requirements development involves several activities: *discovery* and *elicitation* of the system functionality, properties and qualities, *definition* and *specification* of the requirements and precise definition of the system boundary, and *analysis* of the requirements to ensure that they are correct, complete and that they meet the stakeholders expectations. If the analysis reveals undesired properties or flaws, the specification has to be refined. Once the system is implemented, the running system can be validated against the requirements.

2.2 Use Cases

Use cases are a widely used formalism for discovering and recording behavioral requirements of software systems [7]. A use case describes, without revealing the details of the system's internal workings, the system's responsibilities and its *interactions* with its environment as it performs work in serving one or more requests that, if successfully completed, satisfy a goal of a particular stakeholder. A use case can contain several scenarios including the main success scenario and alternate scenarios. The external entities in the environment that interact with the system are called *actors*.

Use cases are stories of actors using a system to *meet goals*. The actor that interacts with the system in the pursuit of a well defined goal is referred to as the *primary actor*. External entities that are required by the system in order to achieve its functionality are called *secondary actors*. Secondary actors include software or hardware that is out of our control. The system, on the other hand, is the software that we are developing and which is under our control.

2.3 Dependability

Systems are developed to satisfy a set of requirements that meet a need. A requirement that is important in mission- and safety-critical systems is that they be highly dependable. *Dependability* [8] is that property of a computer system such that reliance can justifiably be placed on the service it delivers. Dependability involves satisfying several requirements: availability, reliability, safety, maintainability, confidentiality, and integrity. The dependability requirement varies with the target application, since a constraint can be essential for one environment and not so much for others. In this paper, we focus on the *reliability* and *safety* attributes of dependability.

Reliability. The *reliability* of a system measures its aptitude to provide service and remain operating as long as required [9]. Reliability of a service is typically measured in *probability of success* of the service, once requested, or else in *mean time to failure*. If the average time to complete a service is known, it is possible to convert between the two values.

Safety. The *safety* of a system is determined by the lack of catastrophic failures it undergoes [9]. The seriousness of the consequences of the failure on the environment can range from benign to catastrophic. Seriousness of consequences can be measured with a safety index. For instance, the DO-178B standard for civil aeronautics defines safety index values from 0 to 4 with the following meaning:

0. *Without effects*;

1. *Minor effects* lead to upsetting the stakeholders or increasing the system workload;

2. *Major effects* lead to minor injuries of users, or minor physical damage or monetary loss;

3. *Dangerous effects* lead to serious injuries of users, or serious physical damage or monetary loss;

4. *Catastrophic effects* lead to loss of human lives, or destruction of the system.

Each application has different safety requirements. It is now up to the developer in consultation with all the stakeholders to define the number of safety levels to consider, and their exact definitions.

Fault tolerance is a means of achieving system dependability. As defined in [10], fault tolerance includes error detection and system recovery. Error detection involves identification of erroneous state in the system by means of acceptance tests or active redundancy. Late or dead processes can be detected using timers that sound an alert when a deadline for a specific interaction or functionality expires. The error might lead to a permanent or transient failure. *Permanent failures* are failures that persist, and lead to a loss of service until appropriate recovery measures are taken. *Transient failures* are failures which disappear over time. System recovery involves correcting such problems to ensure that the system continues to deliver its services. Forward error recovery techniques restore the system to a new, possibly degraded, state. This approach requires knowledge of the errors and hence is application-specific, but is efficient and suitable in cases of anticipated faults and missed deadlines. A popular forward error recovery technique, exception handling, is discussed in Section 2.4.

At the use case level, error detection involves detection of exceptional situations by means of secondary actors such as sensors and time-outs. Recovery at the use case level involves describing the interactions with the environment that are needed to continue to deliver the current service, or to offer a degraded service, or to take actions that prevent a catastrophe. The former two recovery actions increase reliability, whereas the latter ensures safety.

2.4 Exceptions and Handlers

An exceptional situation, or short *exception* [1], describes a situation that, if encountered, requires something exceptional to be done in order to resolve it. Hence, an *exception occurrence* during a program execution is a situation in which the standard computation cannot pursue. For the program execution to continue, an atypical computation is necessary [11].

A programming language or system with support for exception handling allows users to signal *exceptions* and to define *handlers* [12]. To *signal* an exception amounts to detecting the exceptional situation, interrupting the usual processing sequence, looking for a relevant handler, and then invoking it.

Handlers are defined on (or attached to) entities, such as data structures, or *contexts* for one or several exceptions. According to the language, a context may be a program, a process, a procedure, a statement, an expression, etc. Handlers are invoked when an exception is signaled during the execution or the use of the associated context or nested context. To *handle* means to put the system to a coherent state, i.e. to carry out forward error recovery, and then to take one of these steps: transfer control to the statement following the signaling one (*resumption model* [1]); or discard the context between the signaling statement and the one to which the handler is attached (*termination model* [1]); or signal a new exception to the enclosing context.

3 A Dependability-Focused Requirements Engineering Process

Our **D**ependability-focused **R**equirements **E**ngineering **P**rocess (DREP) targets the development of dependable reactive systems. It defines detailed steps or *tasks* that lead the

[1] It should be noted that the terms *exception* and *handler* are used in this paper at a higher level of abstraction and does not necessarily map to programming language exceptions.

developer to pay particular attention to system safety and reliability when performing requirements elicitation, specification and analysis. The following subsections describe the different activities in detail.

The basic tasks carried out as part of these activities in DREP are outlined in a hierarchical manner in Fig. 1. To clearly illustrate our extensions, the tasks that are part of standard use case analysis are shown in boxes with dashed line borders.

3.1 Requirements Elicitation and Discovery

Task 1: Discovering Actors, Goals, and Modes The first task can be divided into several sub-tasks.

1.1 Brainstorm services/goals and outcomes
1.2 Brainstorm actors
1.3 Classify services/goals and actors
1.4 Decompose services into subgoals
1.5 Brainstorm modes

The first activity in use case based requirements elicitation consists in establishing a list of actors and stakeholders, with a special emphasis on *primary actors*, i.e. external entities in the environment that interact with the system in the pursuit of a well defined goal. Secondary actors are also documented during this brainstorming activity, if their use is indeed part of the requirements and is not already part of the solution domain.

For each of the discovered goals, a *use case outline* is written. This outline consists in a textual summary of the goal, an explanation of the context in which the primary actor wants to achieve the goal, and a clear description of the value or service that the system has to provide to satisfy the primary actor. Complex goals can be split into several subgoals to form a hierarchy [13], in which case a use case outline is written for each of the subgoals. The goals are further be classified as normal services or other special services.

The brainstorming activity can also lead the developer to discover that a given service might have several acceptable *outcomes*, i.e., the system can satisfy the goal of the primary actor in multiple ways.

Finally, during this task, the developer should also consider possible modes of operation to be offered by the system. An operation mode is defined by the set of services that the system offers when operating in that mode[2]. During normal operation, a system should try and provide all of the services it is intended to provide at any given time. There is no need to artificially create different normal modes of operation. Some systems, however, need more than one normal mode of operation, and allow the user to switch between these modes by request or to accommodate changes in the environment. For example, a cell-phone can be put into a child-safe mode, in which the only service offered is to place local calls.

[2] For each service provided in a mode, reliability and safety levels have to be specified as explained in task 4.

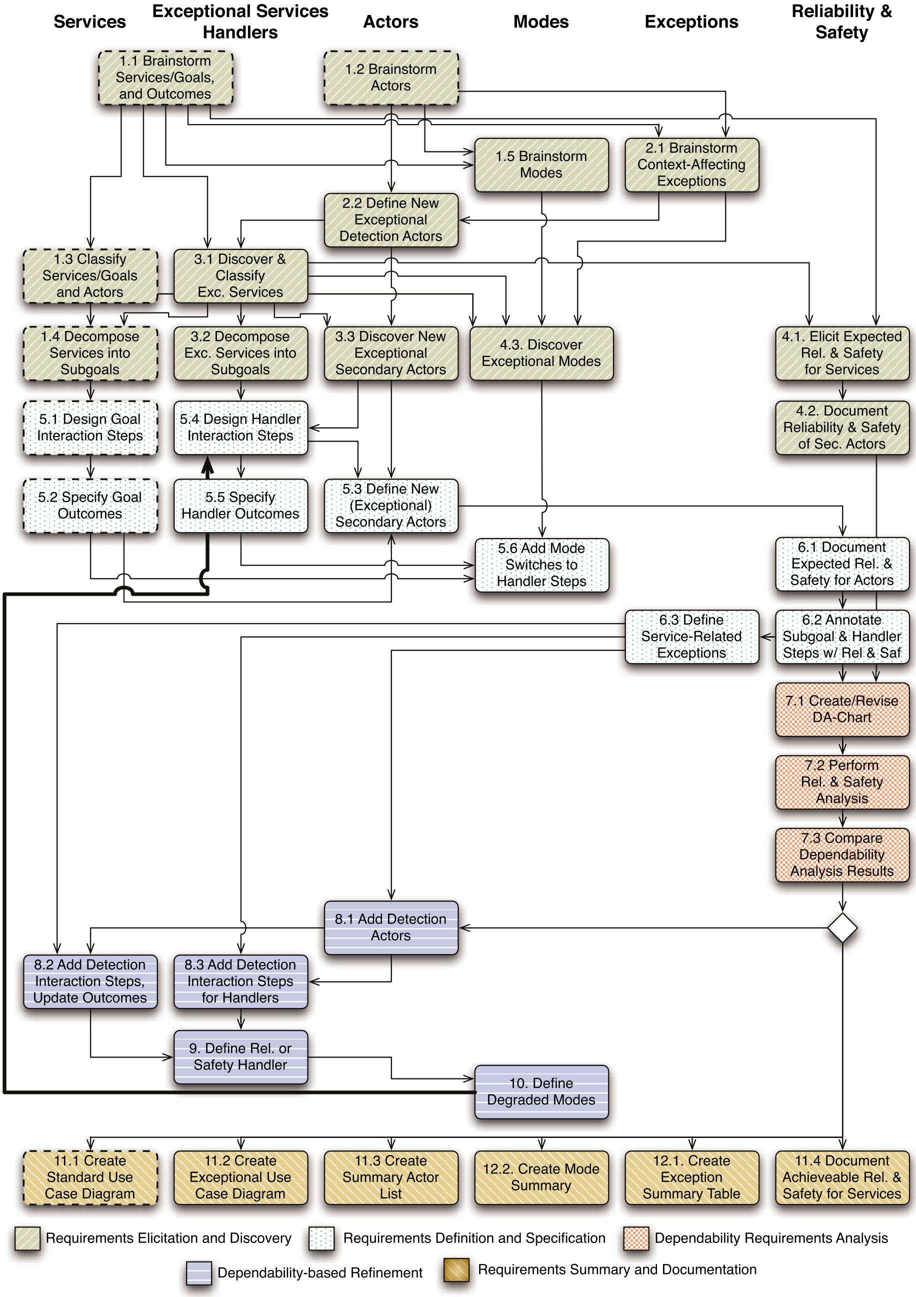

Services
Exceptional Services Handlers
Actors
Modes
Exceptions
Reliability & Safety
1.1 Brainstorm Services/Goals, and Outcomes
1.2 Brainstorm Actors
1.5 Brainstorm Modes
2.1 Brainstorm Context-Affecting Exceptions
2.2 Define New Exceptional Detection Actors
1.3 Classify Services/Goals and Actors
3.1 Discover & Classify Exc. Services
1.4 Decompose Services into Subgoals
3.2 Decompose Exc. Services into Subgoals
3.3 Discover New Exceptional Secondary Actors
4.3. Discover Exceptional Modes
4.1. Elicit Expected Rel. & Safety for Services
4.2. Document Reliability & Safety of Sec. Actors
5.1 Design Goal Interaction Steps
5.4 Design Handler Interaction Steps
5.2 Specify Goal Outcomes
5.5 Specify Handler Outcomes
5.3 Define New (Exceptional) Secondary Actors
5.6 Add Mode Switches to Handler Steps
6.1 Document Expected Rel. & Safety for Actors
6.3 Define Service-Related Exceptions
6.2 Annotate Subgoal & Handler Steps w/ Rel & Saf
7.1 Create/Revise DA-Chart
7.2 Perform Rel. & Safety Analysis
7.3 Compare Dependability Analysis Results
8.1 Add Detection Actors
8.2 Add Detection Interaction Steps, Update Outcomes
8.3 Add Detection Interaction Steps for Handlers
9. Define Rel. or Safety Handler
10. Define Degraded Modes
11.1 Create Standard Use Case Diagram
11.2 Create Exceptional Use Case Diagram
11.3 Create Summary Actor List
12.2. Create Mode Summary
12.1. Create Exception Summary Table
11.4 Document Achieveable Rel. & Safety for Services
Requirements Elicitation and Discovery
Requirements Definition and Specification
Dependability Requirements Analysis
Dependability-based Refinement
Requirements Summary and Documentation

Fig. 1. Task Structure of DREP

Task 2: Discovering Context-Affecting Exceptions Task 2 involves carrying out the following two sub-tasks.

2.1 Brainstorm context-affecting exceptions
2.2 Define new exceptional detection actors

In this step, the developer has to focus on *context-affecting exceptional situations*, i.e., situations that change the context in which the system operates. Certain context changes might require a dependable system to adapt in order to continue to provide reliable and safe service. To help discover these situations, the following questions should be answered:

- What situations / conditions / changes in the environment make it impossible for the entire system to provide safe service? In such situations, should the system provide some other service?
- What situations / conditions / changes in the environment prevent the system from satisfying a primary actor's goal (or subgoal)? In such situations, can the system partially fulfill the service?
- What situations take priority over the primary actor's goal?
- What situations / conditions / changes in the environment could make the primary actor change his goal? In such situations, how can the primary actor inform the system of the goal change?

For each exceptional situation that is discovered, a *named exception* is defined, together with a small text that describes the situation in more detail. All discovered exceptions are documented in an exception table.

This activity typically leads to the discovery of new *exceptional goals*. Often, the occurrence of the situation cannot be detected by the system without help from the environment, which means that new *exceptional actors* have to be introduced. For example, in an elevator system where safety is the main concern, in case of a fire outbreak in the building, the elevator operator or a smoke detector, both exceptional actors, should activate the fire emergency mode of the elevator control software.

Task 3: Eliciting Handlers for Context-Affecting Exceptions This task can be split into the following sub-tasks.

3.1 Discover and classify exceptional services
3.2 Decompose exceptional services into subgoals
3.3 Discover new exceptional secondary actors

For each identified exception, a *handler use case* outline has to be established describing how the system is supposed to react or *recover* from that situation. A handler can be further classified as a *safety* or *reliability* handler depending on the concern it attempts to satisfy. A handler can also be linked to one or several contexts, i.e. use cases during which the exceptional situation can occur. Upon occurrence of the exception, the current interaction is interrupted and the exceptional interaction begins. In an elevator system, for example, in case of a fire outbreak signalled by a smoke detector, standard elevator operation is interrupted. To ensure safety, the elevators are brought to the ground floor.

Task 4: Eliciting Dependability Expectations and Discovering Exceptional Modes This task can be split into the following sub-tasks.

4.1 Eliciting dependability expectations for each service
4.2 Document provided reliability and safety of mandatory secondary actors
4.3 Discover exceptional modes of operation

For dependable systems, it is at this phase important to discover the requirements with respect to safety and reliability for each service that the system provides. Mission and safety-critical systems often have to comply with *safety standards*, but even if the requirements do not require compliance with a standard, stakeholders and primary actors explicitly or implicitly *expect a certain degree* of safety and reliablility from a dependable system. To document the desired dependability, reliability and safety annotations have to be added to the use case outlines that, for each goal, specify the desired probability of successful achievement of the goal, as well as the maximum tolerable probability of occurrence of a safety violation.

The desired safety and reliability values should be elicited not onlyfor the normal services of the system, but also for the new *exceptional goals* discovered in task 2.

It is important to note here that in the real world, 100 percent dependability is never achievable. If the specified safety and reliability are too high, then it might be impossible (or too expensive) to implement a system that fulfills the requirements. It is hence important that the stakeholders decide on acceptable risks at this point.

Next, for each service offered by a mandatory secondary actor, the developer has to document the service's reliability and safety properties. If the secondary actor is a piece of hardware, then the reliability can be found in the specification manual.

Whenever a dependable system has encountered difficulties performing a requested service due to some exceptional situation, the effect of the encountered problem on future service provision of the system has to be evaluated. If the reliability or safety of future service provision is threatened, then a *mode switch* is necessary. Switching to a different operation mode (an exceptional mode) allows the system to signal to the environment that the services offered by the system have changed, and reject any requests for services that cannot be performed with sufficient reliability or safety. We have addressed exceptional modes of operation in the behavioural models used in DREP, and details can be found in [14].

While normal modes of operation have been discovered in task 1.5, this task concentrates on the discovery of emergency and restricted modes. In an *emergency mode*, normal services are suspended and only emergency services, possibly initiated by a new exceptional actor, are available. The system is in a state in which it cannot provide any of its normal services anymore, not even in a degraded form. This is usually due to safety reasons. For example, in case of a fire alarm, an elevator system does not handle user requests anymore, but moves all the elevator cabins to the ground floor. In an *restricted mode*, a combination of emergency services and normal services are offered. The system is in an exceptional state in which only a subset of the normal services are available and the functions of particular emergency services are also required.

To discover emergency and restricted modes, all context-affecting exceptions identified in task 2.1 need to be considered. In a new context, some of the services provided under normal circumstances might not be adequate anymore. Therefore, the developer should reflect on the impact of the context change on each of the services provided by the system. If the safety of normal services is threatened by a situation, then an appropriate exceptional or emergency mode should be defined.

If during this task new modes of operation have been defined, then it is important to specify the expected reliability and safety of each service (see task 4) provided in each mode.

It is important to note that mode definitions are not based on the developer's creativity. Each mode has to be validated with the stakeholders to check if, according to them, the services provided in the mode form a coherent set, and that the provided levels of reliability and safety for each service are sufficient.

3.2 Requirements Definition and Specification

Now that the goals and subgoals have been identified, detailed use case descriptions have to be elaborated for each of them. We suggest to describe use cases with a predefined template as done by others [15], which forces the developer to explicitly document all relevant features.

Reactive systems only perform work or produce output after they receive an input event. Therefore, the main parts of our use case template consist of a numbered list of individual base interaction *steps*, each one describing either an *input interaction* – an external actor decides to send a message/data/event to the system – or an *output interaction* – the system sends a message/data/event to an external actor. If a use case is decomposed into subfunction-level use cases, a step can also be a reference to a lower level use case, which in turn describes the base interaction steps that leads to the completion of the subgoal. In any case however, a use case describing a user goal could be flattened into a sequence of base interaction steps, if needed.

Task 5: Designing Interactions Interaction design in DREP is comprised of several sub-tasks.

5.1 Design goal interaction steps
5.2 Specify goal outcomes
5.3 Define new (exceptional) secondary actors
5.4 Design handler interaction steps
5.5 Specify handler outcomes
5.6 Add mode switches to handler steps

The standard way of achieving a goal is described in the *main success scenario* part of the template. The ordering of the individual interaction steps are often dictated by logic, by required usage patterns, by user interfaces, or by protocols enforced by secondary actors interacting with the system. Where flexibility exists, the stakeholders should be consulted to choose the most adequate interaction pattern.

When designing the goal interaction steps, it is also necessary to define the service outcome. The main success scenario of a user goal can end in only one possible way, and the use case should clearly show this. From the users' perspective, the goal outcome can be one of the following: $\ll$ *success* $\gg$, $\ll$ *failure* $\gg$, or $\ll$ *goal abandoned* $\gg$[3]. If alternate scenarios are available, it is also necessary to specify the outcome of all such alternate paths, and to document them in the use case extension section.

When the requested service cannot be provided, a dependable system should strive to handle the current situation and attempt to provide partial service, if possible. Partial service, or *degraded service outcome* as we call it, happens when a service does not deliver what initially promised, but yet provides something that potentially satisfies the requester of the service. A degraded outcome is better than a complete failure to deliver the service.

Intuitively, a service provision can only result in a degraded outcome when an exceptional situation has occurred. Reacting to such an exceptional situation, and providing a well-defined outcome can only be done within a handler use case. Therefore, after the detailed interaction steps of a handler have been designed, the outcome of the handler should be clearly defined. Handlers can end in $\ll$ *success* $\gg$, $\ll$ *degraded* $\gg$ or $\ll$ *failure* $\gg$.

Whenever an exception has lead to the definition of a new mode, then the steps of the handler that addresses the exception have to be updated to indicate a mode switch. In general, mode switches should be performed as soon as possible, i.e. as soon as it becomes apparent that the provision of the current services at the required reliability and safety level can not be sustained.

Task 6: Defining Service-Related Exceptions and Effects on System Reliability and Safety This task focuses on discovering service-related exceptions and documenting dependability values.

6.1 Document expected reliability and safety for actors
6.2 Annotate subgoal and handler steps with reliability and safety
6.3 Define service-related exceptions

The successful completion of a user goal may be threatened due to service-related exceptional situations. Service-related exceptions have many natures:

- The system state makes the provision of a service impossible[4],
- Failure of secondary actors that are necessary for the completion of the user goal,
- Failure of communication links between the system and important secondary actors,
- Actors violate the system interaction protocol, i.e. they invoke system services in the wrong order, or at the wrong time.

Possible service-related exceptions can be discovered most effectively following a bottom-up approach. DREP requires the developer to examine each individual base step

[3] To correctly calculate reliability, it is important to separate the situations in which the user voluntarily abandons the goal from the situations in which the service fails. A service that is successfully cancelled upon user request represents a correct and reliable system behavior.

[4] Addressing these situations is of course not new to our approach. Standard use case driven requirements engineering techniques usually specify the handling of such situations in an extension section of the use case.

of a use case, sub-use case or handler, and reflect on the consequences that a failure of the step has on reliability, i.e. the achievement of the goal, and on system safety. The developer should answer the following questions:

- If this step is omitted, will the goal fail? If yes, the step should be annotated with a *reliability tag*, together with the probability of success of the step.
- If this step is omitted, is the safety of the system threatened? If yes, the step should be annotated with a *safety tag*, together with the corresponding safety level and the probability of success of the step.

A *named exception* should be defined for each service-related exceptional situation, together with a small text that describes the situation in more detail. For example, in an elevator system, a motor failure, i.e. the situation in which the motor does not react to commands anymore because of a hardware or communication failure, is a serious threat to safety and reliability. The identified exception is then added to the exception table of environmental exceptions for documentation reasons.

3.3 Dependability Requirements Analysis

Task 7: Assessing Safety and Reliability The sub-taks involved in the assessment phase are listed here.

7.1 Create/revise DA-Chart
7.2 Perform reliability and safety analysis
7.3 Compare dependability analysis results with expected dependability values

In [16], we proposed a model-based approach for analyzing the safety and reliability of our use cases. Since each interaction step in a use case is annotated with a probability reflecting its chances of success, and a safety tag if the failure of the step hampers the system safety, it is possible to map the use case to a formalism that is well-suited for dependability analysis. For this purpose, we developed the DA-Charts formalism [16] which is a probabilistic extension of part of the statecharts formalism. We have implemented our formalism in the AToM^3 tool [17] to provide support for automatic dependability analysis. The tool allow a developer to create a DA-Chart that corresponds to the use cases established in tasks 1 - 6. The tool also verifies the formalism constraints and ensures that the mapping rules are adhered to. Based on path analysis of the DA-Charts, the tool quantitatively determines probabilities of reaching safe or unsafe states, or achieving the goal, providing a degraded success, or failing. For details on the DA-Charts formalism and the dependability analysis see [18].

The dependability determined by the tool can now be compared with the dependability required by the stakeholders as determined in task 4. If the analysis reveals an acceptable level of reliability and safety, then the requirements engineering process is complete, and a summary specification can be established (see tasks 11 and 12). Otherwise, the requirements need to be refined with handler use cases that address the service-related exceptions, which is described in tasks 8, 9 and 10.

3.4 Dependability-Based Refinement

Based on the output of our analysis tool, the service-related exceptions that have significant negative effect on the system's reliability and safety can be identified. To improve the situation, the following tasks should be performed for *each one of them*.

Task 8: Specifying Detection Mechanisms Once possible exceptional situations have been elicited, it is important to carry out the following tasks.

8.1 Add detection actors
8.2 Add detection interaction steps and revisit goal outcomes
8.3 Add detection interaction steps for handlers

Before any recovery actions can be taken by the system, the exceptional situation has to be *detected*. The developer should investigate if the current actors and their interactions make the detection possible, and if not, adapt the interaction pattern or even add secondary detection actors to the system's environment.

Detection is usually done differently for *input* and *output* interactions. Omission of input to the system can usually be detected using timeouts. Invalid input data can be detected with checksums, etc. In both cases, no additional detection actors have to be introduced. The use case has to be updated by adding the discovered exception to the extension section of the template as an alternative to the essential input step. If necessary, new use case outcomes might have to be defined.

Output failure is more difficult to handle. Whenever a system output triggers a critical action of an actor, then the system must make sure that it can detect eventual communication problems or failure of an actor to execute the requested action. This very often requires additional hardware, e.g. a sensor, to be added to the system. The job of this new actor is to inform the system that the essential actor successfully executed the system's request. This new acknowledgement step has to be added to the main success scenario after the essential output step in the use case or handler, and an exception representing the failure of the output, detected by a timeout while waiting for the acknowledgement, is added to the extension section as an alternative to the acknowledgement step. For example, an elevator control software might request the motor to stop, but a communication failure or a motor misbehaviour might keep the motor going. Additional hardware, for instance, a sensor that detects when the cabin stopped at a floor, might be necessary to ensure safety or reliability.

Task 9: Specifying Handler Use Cases If the exception puts the user in danger, then measures must be taken to put the system in a safe state. If the exception threatens the successful completion of the user goal, reliability is at stake. It should then be investigated if the system can recover and meet the user goal in an alternative way.

In any case, exceptional interaction steps with the environment are performed during recovery, and hence must be specified in a separate reliability or safety handler use case[5]. Very often, actors – especially humans – are "surprised" when they encounter an

[5] Separation of handlers also enables subsequent reuse of handlers. Just like a subfunction-level use case can encapsulate a subgoal that is part of several user goals, a handler use case can encapsulate a common way of handling exceptions that might occur while processing different user goals. Sometimes even, different exceptions can be handled in the same way.

exceptional situation, and are subsequently more likely to make mistakes when interacting with the system. Exceptional interactions must therefore be as intuitive as possible, and respect the actor's needs.

If the goal of the primary actor cannot be achieved, then it is of paramount importance to inform him of the situation by an appropriate output interaction. In some cases, it might not be possible to satisfy a user's goal completely, but a dependable system can instead offer a degraded form of service. For example, a user might order a product online and request for delivery before a certain date. If the system is unable to satisfy this request, the user might be offered the option to pick-up the order at the store instead. The handler use case should then define a new *degraded outcome* for this situation.

If omission of input from an actor can cause the goal to fail, then, once the omission has been detected, different options of handling the situation have to be considered. For instance, prompting the actor for the input again after a given time has elapsed, or using default input are possible options. Safety considerations might make it even necessary to temporarily shutdown the system in case of missing input.

Invalid input data is another example of input problem that might cause the goal to fail. Since most of the time the actors are aware of the importance of their input, a reliable system should also acknowledge input from an actor, so that the actor realizes that he is making progress in achieving his goal.

Task 10: Defining Degraded Modes For each of the service-related exceptions identified in task 6.3 and handled in task 9, the developer should evaluate the effects that the service-related exception has on future requests for the same service or other services that could be affected by the exception. In the case where these effects lower the reliability and safety of the service below the required level specified by the current mode, then a degraded mode should be defined.

A *degraded mode* of operation (of a normal mode) offers only limited services. Some services of the normal mode are still provided as is. Some services are provided, but with a lower degree of reliability and safety. In this case, the service is said to be offered with *degraded quality of service* (QoS). For example, a web browser running low on memory might switch into a mode where only textual elements from webpages are displayed and graphical elements and other media are suppressed to save memory.

Iteration To complete this iteration, every interaction step of the newly defined handler of task 9 must again be elaborated (task 5.4), the outcomes must be specified (task 5.5), and the essential steps tagged with reliability probabilities and safety information (task 6.2). Finally, the developer can re-analyze the updated use cases (task 7) to determine if the required safety and reliability requirements can now be satisfied.

3.5 Requirements Summary and Documentation

To begin with the requirements documentation, the following tasks are suggested.

Task 11: Use Case Summary

11.1 Create standard use case diagram
11.2 Create exceptional use case diagram

11.3 Create summary actor list
11.4 Document achievable reliability and safety for services

Whereas individual use cases are text-based, the UML use case diagram provides a concise high level view of the use cases of a system. It allows developers to graphically depict the use cases, the actors that interact with the system, and the relationships between actors and use cases. To begin with in this phase, a standard use case diagram based on the actors and goals defined earlier should be created.

In a use case diagram, standard use cases appear as ellipses, associated to the actors whose goals they describe. In [19] we extended use case diagrams and proposed to identify handler use cases with a < <handler> > stereotype or even a different graphical symbol in order to differentiate them from standard use cases. We also suggest classifying the handlers as a < <safety handler> > or a < <reliability handler> >. Our notation for handlers is illustrated in Fig. 8. Having different type of handlers enables quick identification of functionality that affects safety or reliability of a system, as well as identification of safety-critical parts of the system. It allows the developer in collaboration with the stakeholders to decide, for instance, how much resources should be allocated to the development of the functionality defined in the handler use cases, or to prioritize between safety and reliability in case of conflict.

Handler use cases are associated to a base use case, which may be any standard use case or other handler use case. We suggest to depict this association in the use case diagram by a directed relationship (dotted arrow) linking the handler use case to its base use case. This relationship is very similar to the standard UML < <extends> > relationship. It specifies that the behavior of the base use case may be affected by the behavior of the handler use case in case an exception is encountered.

In case of an occurrence of an exceptional situation, the base behavior is put on hold or terminated, and the interaction specified in the handler is started. A handler can temporarily take over the system interaction, for instance to perform some compensation activity, and then switch back to the normal interaction scenario. In this case, the relationship is tagged with a < <interrupt& continue> > stereotype. Some exceptional situations, however, cannot be handled smoothly, and cause the current goal to fail. Such dependencies are tagged with < <interrupt & fail> >. The exceptions that activate the handler use case are added to the interrupt relationship in a UML comment, similar to what is done for extension points.

In addition, a list of all primary and secondary actors both normal and exceptional should be developed. For each service to be provided by the system, it is necessary to document the reliability and safety that can be achieved.

Task 12: Summary Tables

12.1 Exception summary table
12.2 Mode summary table

For traceability and documentation reasons, all discovered environmental and service-related exceptions are recorded in a table during tasks 2 and 6. As a summary, the entries in this table already contains a small textual description of the exceptional situation should be complemented with the exception contexts in which the exception can occur, the associated handler(s), and the mechanism for detecting the exception.

Finally, the **mode table** is created to summarize all modes of the system. The mode table can of course also be created earlier and updated iteratively whenever a new mode is defined. For each mode the table includes a *mode name*, a *description* of the mode, followed by a list of services that are provided in the mode. For each service, the *service name*, the *expected minimal reliability*, and the *expected minimal safety* are given.

4 Case Study: 407 Express Toll Route System

We illustrate the process described in Section 3 with the 407 ETR (Express Toll Route) System. The 407 ETR is a highway that runs east-west just North of Toronto, and was one of the largest road construction projects in the history of Canada. The road uses a highly modern Electronic Toll Collection system that allows motorists to pass through toll routes without stopping or even opening a window. The ETR system is a hard real-time application requiring high levels of dependability.

Vehicles can be registered with the 407 ETR system, in which case the driver is issued a small electronic tag, called a transponder, to be attached to the windshield. When the vehicle enters the highway, it passes under the overhead gantry. The hardware devices of a gantry include a vehicle detector, a locator antenna, a read/write antenna, cameras, lights, and a laser scanner. The locator antenna determines if the vehicle is equipped with a transponder. Next, the read/write antenna reads the account number from the transponder and the point of entry, time and date is recorded. In addition, the system uses laser scanners to determine the class of vehicle. The same process occurs when the vehicle exits the highway. The entry and exit data are then matched and the transponder account holder is debited.

Unregistered vehicles are identified by their license plate number. The system triggers cameras and lights to take pictures of the rear number plate. At the same time, the laser scanners are activated to classify the vehicle in order to determine the trip charge. The owner of the vehicle is identified by electronic access to government records. If the video correlation and image processing fails to determine the license plate with sufficient probability, a human operator has to look at the pictures to make the call.

4.1 Elicitation and Discovery

Task 1: Discovering actors, goals, and modes. In the ETR system there is initially only one primary actor, the *Driver*. The summary-level use case *UseHighway* is shown in Fig. 2. When interacting with the system, the driver has the goal of registering (*RegisterVehicle*), taking the highway (*TakeHighway*), payment of bills (*PayBill*), and cancelling the registration (*CancelRegistration)*. These goals can be further split into subgoals as summarized in Fig. 2.

The 407 ETR system only has one normal mode of operation since there is only one primary goal, using the highway, that needs to be satisfied at all times.

Task 2: Discovering Context-Affecting Exceptions. In the ETR system, an accident on the highway or extreme weather leading to critical road conditions, would require the highway operator, an exceptional actor, to temporarily close parts of the highway. Activating the emergency behavior is an *exceptional goal* for the operator, since this happens

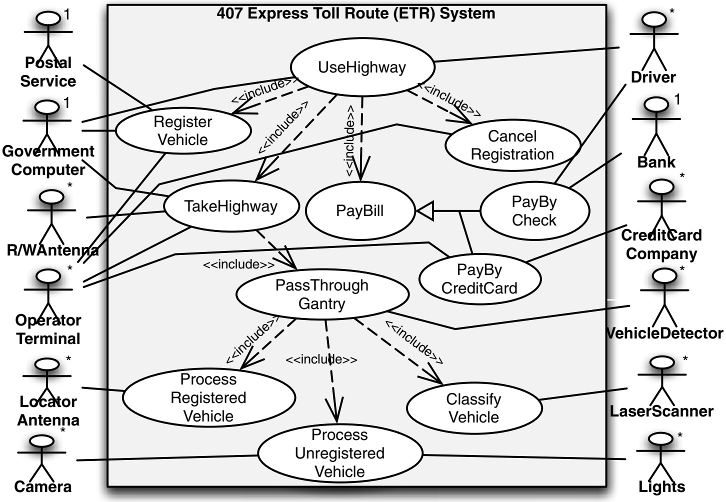


Fig. 2. 407 Standard Use Case Diagram

only in rare occasions. In this task, we identified an environmental exception: *HighwayUnavailable* which is signaled by the *HighwayOperator* exceptional actor when he receives the request from the Roadside Motorist Assistant Patrol.

Task 3: Eliciting Handlers for Context-Affecting Exceptions. After a discussion with the stakeholders it has been decided that closing the highway in case of emergencies is done by activating barriers that prevent new vehicles from entering the highway at the closed sections. The vehicles on the highway are to be informed of the situation by displaying messages on message boards. We therefore identified new secondary actors, the *Barrier*, and the *Message Board*.

Task 4. Eliciting Dependability Expectations. We define 3 safety levels for the 407 ETR: level 0 - without effects, level 1 - cars end up in a traffic jam / increased load on operators, level 2 - cars are damaged / system components are damaged. The reliability of *TakeHighway* and *PayBill* should be high, e.g. 0.999 (one of 10000 cars can fail to pay for a trip). Other services, e.g. registering and cancelling, require a reliability of 0.995. During an emergency, the chances of a level 2 safety violation should be very small, e.g. 0.00001, and a level 1 safety violation should be rare, 0.001. The required reliability of the emergency behavior should be very high, e.g. 0.99999.

Task 5: Discovering and Classifying Normal and Exceptional Modes.

As mentioned earlier, the system only offers one normal mode which includes all services associated to taking the highway, i.e. *TakeHighway*, *RegisterVehicle*, *PayBill*, and *CancelRegistration*.

In task 3, the context-affecting exception *HighwayUnavailable* was identified since critical road conditions might require the system to restrict vehicles from entering the highway. Emergency services such as activating the road barriers are carried out. However, the exit service needs to be active to allow the vehicles already on the highway

Use Case: TakeHighway
Level: User Goal
Primary Actor: Driver
Main Success Scenario:

1. Driver enters highway, passing through gantry.
2. Driver exits highway, passing through gantry.
3. System retrieves the driver's vehicle record based on trip information*.
4. System determines the amount owed based on the trip information and adds the transaction to the vehicle's records.
5. System informs Driver by sending a signal to the RWAntenna of successful completion of transaction.

Extensions:

3a. Vehicle is unregistered and does not have a record yet.
  3a.1. System sends licence plate information to GovernmentComputer.
  3a.2. GovernmentComputer sends vehicle information and owner's address to System.
  3a.3. System creates a new vehicle record. Use case continues at step 4.
3b. Vehicle is unregistered and licence plate is unrecognizable.
  3b.1. System displays pictures on OperatorTerminal.
  3b.2. OperatorTerminal sends licence plate information to System. Use case continues at step 3.
5a. Vehicle is not registered. Use case ends in $\ll$ *success* $\gg$.

Fig. 3. *TakeHighway* Use Case

to continue. Therefore, when such an exceptional situation arises the system needs to switch to a *restricted mode ExitOnly*.

4.2 Requirements Definition and Specification

Due to space constraints, we only discuss the user-goal level *TakeHighway*, and the subgoals *PassThroughGantry* and *ProcessRegisteredVehicles*.

Task 6: Designing Interactions

The interaction steps required for *TakeHighway* are detailed in Fig. 3. To take the highway, the *Driver* enters the highway, and then exits it by passing through gantries. If the vehicle has a transponder, then the device beeps and blinks green after the driver exits the highway. The *ProcessRegisteredVehicle* use case, shown in Fig. 4, describes how the system communicates with the transponder, and verifies the class of the vehicle. The sub-functional level uses cases that describe the processing of unregistered vehicles and the classification of vehicles are not shown here for space reasons.

The extension section of *TakeHighway* on purpose describes only alternative ways to achieve the goal, since exceptional interaction will be shown later.

Fig. 5 shows the handler *ActivateBarrier* that handles the exception *HighwayUnavailable*. Handler use cases have an additional field in the use case template named *Contexts & Exceptions* that is used to document by which exception and in what context the handler is triggered. In our case, a *HighwayUnavailable* exception occurring puts the system in a restricted mode, at which time the system only allows exits but does not allow new goals to start. As a first handling step, the emergency road barriers are activated. Then the message boards are updated with a warning message. Subsequently when the road conditions improve, the *Operator* can deactivate the barriers and grant access to the highway once again. Since the handler attempts to satisfy user safety, we

Use Case: ProcessRegisteredVehicle
Level: Sub-Function
Primary Actor: N/A
Main Success Scenario:

1. LocatorAntenna notifies System that it detected an approaching vehicle with transponder.
2. System asks R/WAntenna to obtain account information from transponder.
3. RWAntenna informs System of account information.
4. System records account information for the trip.
5. System turns on the Lights.
6. System triggers the Cameras.
7. Cameras send images to System.
8. System determines licence plate information based on images.

Extensions:
1a. The approaching vehicle does not have a transponder. Use case ends in $\ll failure \gg$.

Fig. 4. *ProcessRegisteredVehicle* Use Case

Handler Use Case: ActivateBarrier
Handler Class: Safety
Context & Exception: TakeHighway{HighwayUnavailable}
Level: Usergoal
Primary Actor: Operator
Main Success Scenario:
System switches into restricted mode ExitOnly.

1. System activates barriers at entry gantries.
2. System displays "Highway Unavailable" at message boards on highway.
3. Operator informs System that highway is accessible again.
4. System deactivates barriers.
5. System clears message boards.

System switches back to normal mode.

Fig. 5. *ActivateBarrier* Handler Use Case

label the handler as a *safety handler*. This is shown in the *Handler Class* field in the template.

Task 7: Defining Service-Related Exceptions and Effects on System Reliability and Safety. We begin in a bottom-up way by examining each step in the *ProcessRegisteredVehicle* use case to determine how essential it's contribution is in order to achieve the goal. For example, step 1 involves the locator antenna notifying the system that a vehicle with a transponder is passing by. This is an input interaction, and its omission leads to an exceptional situation. An antenna defect would cause vehicles to be incorrectly identified as unregistered vehicles. Therefore, the step is annotated with a *reliability* tag together with the failure probability. Next, if the read/write antenna malfunctions, the registration information associated with the transponder would be inaccessible. Malfunctioning lights or cameras also hinder the success of the goal, so they are tagged as well. The exceptions that arise are defined as *LocatorAntennaFailure*, *RWAntennaFailure*, *LightFailure*, and *CameraFailure*.

In *TakeHighway*, the government computer might fail to send the requested information back to the system. The operator might fail to respond when a picture is sent to him. The transponder might not react to the acknowledgement signal sent by the read/write antenna. The service-related exceptions identified in this task that occur in the *TakeHighway* context are named as *GovernmentComputerUnavailable*, *OperatorFailure*, and *TransponderUnreachable*. *Reliability* tags are attached to each of these steps. None of the steps is safety-critical.

We also need to consider the possibility of the handlers failing and the consequences of such failures. While handling the *HighwayUnavailable* exception, the barrier might fail to get activated resulting in a highly unsafe condition. Step 3 in Fig. 5 is therefore annotated with a *safety* tag and corresponding probability, and the extensions section is appended with a *BarrierFailure* exception.

4.3 Requirements Analysis

Task 8: Assessing Safety and Reliability. The probabilistic analysis of the system is not elaborated here for space reasons. The interested reader is referred to [16] for details. It reveals that system safety and reliability cannot be met with the current interaction: *BarrierFailure* has to be handled in order to improve safety, *GovernmentComputerUnavailable*, *OperatorFailure* and *TransponderUnreachable* have to be addressed in order to improve reliability.

4.4 Dependability-Based Refinement and Iteration

We use the *TakeHighway* use case and the exceptions occurring in it to illustrate the tasks in this section.

Task 9: Specifying Detection Mechanisms. We first address the reliability issues in the *TakeHighway* use case. To begin with, detecting unavailability of the government records (exception *GovernmentComputerUnavailable*) can be done by using a timeout (the lack of reception of a message), as discussed in Section 3.4. The *OperatorFailure* exception can also be detected in a similar manner. To detect the exception *TransponderUnreachable*, we need to know whether the read/write antenna was able to reach the transponder. Hence, an additional acknowledgement step is needed. Detecting a failed entry is done when an exit is detected. Detecting a failed exit is done using a timeout. The updated use case is shown in Fig. 6.

To increase safety, we need to find a mechanism to detect the failure of the barrier. To this intent, we introduced an additional sensor which detects when a barrier is closed. A malfunctioning barrier can therefore be detected by the absence of the acknowledgement. The *ActivateBarrier* handler use case is updated with the detection and acknowledgement step (not shown for space reasons).

Task 10: Specifying Handler Use Cases. In the case where an exit or entry of a vehicle is not detected, it is impossible to determine the length of the vehicle's trip. Therefore, the driver is billed for a minimal charge, shown in the *TakeHighway* use case in Fig. 6 with the degraded outcome *MinimalTrip* (steps 4a.1a and 4b.1a).

If the transponder is unreachable, the driver can not be notified of the success of the transaction. Therefore the use case ends in the degraded outcome *DriverNotNotified* (step 6a).

We know that the government computer is highly reliable and available, and therefore failures reaching the government computer are probably of temporary nature. Therefore

Use Case: TakeHighway
Level: User Goal
Primary Actor: Driver
Main Success Scenario:

1. Driver enters highway, passing through gantry.
2. Driver exits highway, passing through gantry.
3. System retrieves the driver's vehicle record based on trip information*.
4. System determines the amount owed based on the trip information and adds the transaction to the vehicle's records.
5. System informs Driver by sending a signal to the RWAntenna of successful completion of transaction. *reliability*
6. System receives confirmation from RWAntenna that the driver was notified.

Extensions:

3a. Vehicle is unregistered and does not have a record yet.
 3a.1. System sends license plate information to GovernmentComputer.
 3a.2. GovernmentComputer sends vehicle information and owner's address to System. *reliability*
  3a.2a. Exception{GovernmentComputerUnavailable}: use case ends in $\ll failure \gg$.
 3a.3. System creates a new vehicle record. Use case continues at step 4.
3b. Vehicle is unregistered and license plate is unrecognizable.
 3b.1. System displays pictures on OperatorTerminal.
 3b.2. OperatorTerminal sends license plate information to System. Use case continues at step 3. *reliability*
  3b.2a. Exception{OperatorFailure}: use case ends in $\ll failure \gg$.
4a. Exit unsuccessful.
 4a.1a. If entry was successful, minimum trip charge is added to vehicle's records. Use case ends in $\ll$ *degraded* $\gg$ *MinimalTrip*.
 4a.1b. If entry was unsuccessful as well, use case ends in $\ll failure \gg$.
4b. Entry unsuccessful.
 4b.1a. If exit was successful, minimum trip charge is added to vehicle's records. Use case continues in $\ll$ *degraded* $\gg$ *MinimalTrip* at step 4.
5a. Vehicle is not registered. Use case ends in $\ll$ *success* $\gg$.
6a. Exception{TransponderUnreachable}: use case ends in $\ll$ *degraded* $\gg$ *DriverNotNotified*.

Fig. 6. Updated *TakeHighway* Use Case

the service-related exception *GovernmentComputerUnavailable* is handled by resending the request. The handler defined for this task is shown in Fig. 7. The *OperatorFailure* exception can be handled in a similar manner, resending the request to the operator again or by trying another operator terminal.

In case of a malfunctioning transponder, the client is notified of the problem. He is given a grace period within which to service the transponder, and during which time he will not be charged a video toll charge. The handler defined for this purpose, *WarnClients*, is shown in Fig. 7. The handlers defined in this task are accordingly labelled as *reliability handlers*.

In case of the *BarrierFailure* exception, the system should immediately notify an operator. The operator can then evaluate the situation and, if necessary, call a service person and inform the patrol officers. This functionality is described in the safety handler use case *CallHighwayPatrol* (not shown here for space reasons).

Task 11: Defining Degraded Modes. In the 407 ETR system, even if the hardware of some entry or exit gantries are malfunctioning, it was decided that the highway should

Handler Use Case: RetryGovtComp
Handler Class: Reliability
Context & Exception: TakeHighway{GovernmentComputerUnavailable}
Primary Actor: N/A
Secondary Actor:
Main Success Scenario:

1. System resends license plate information to the government computer.
   Step 1 is repeated 2 times.
2. Government computer sends vehicle information.

Handler Use Case: WarnClients
Handler Class: Reliability
Context & Exception: TakeHighway{TransponderUnreachable}
Primary Actor: N/A
Main Success Scenario:

1. System ascertains that the transponder is out of order.
2. System notifies operator that the transponder is not responding.
3. System flags transponder account as temporarily unavailable and cancels the video toll charge.
4. Operator issues a warning letter to the vehicle owner.
5. Owner brings transponder to the office for service.
6. Operator changes status of the account after transponder is serviced.

Extensions:
5a.1 System warns that grace period is over.
5a.2 System cancels discount of video toll charge.

Fig. 7. *RetryGovtComp* and *WarnClients*

continue to operate (and charge minimal trips for vehicles that enter or exit through malfunctioning gantries).

Bad weather conditions might prevent a video camera from capturing clear pictures, or transmission problems might prevent the captured images from reaching the central computer. In this case, the detection and recognition services of a video surveillance system might temporarily be less reliable. In such a situation, the system would switch to a *degraded mode DegradedReliability*.

4.5 Requirements Summary

Task 12: Use Case Summary. Fig. 8 shows the use cases, exceptions and handlers related to *TakeHighway* by means of an extended use case diagram. All exceptional interactions are tagged with the < <handler> > stereotype along with the handler class *safety* or *reliability*, and all exceptional situations that trigger these interactions are documented using notes attached to the < <interrupt> > relationships. For space reasons, the secondary actors have been omitted from the diagram.

Task 13: Exception Summary Table. The exception table is very straightforward to create (as described in Section 3.5) and is presented in Table 1.

Task 14: Mode Summary Table. The mode summary table only contains three modes: the normal operation mode, the restricted mode *ExitOnly*, and the degraded mode *DegradedReliability*.

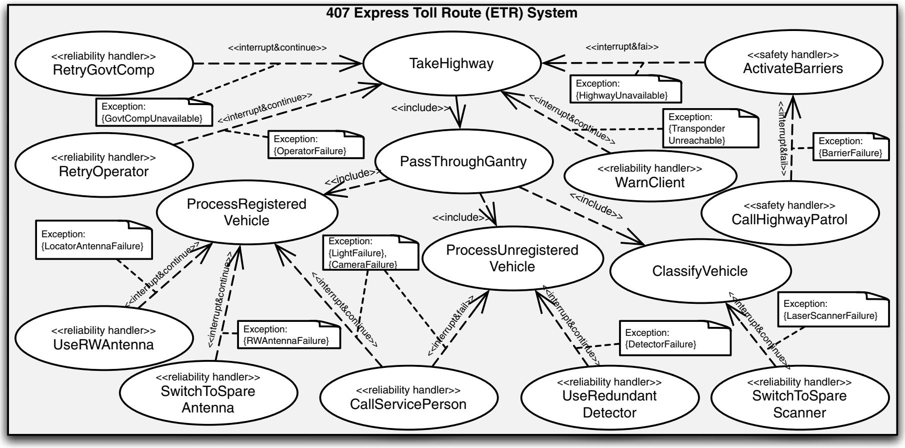


Fig. 8. Reliable and Safe 407 ETR Use Case Diagram

5 DREP and Model-Driven Engineering

The left hand side of Fig. 9 shows a summary of the tasks of our dependability-aware requirements engineering process.

The arrows illustrate that the developer is expected to go through several *iterations* of the process, refining the use cases and handlers if the analysis task reveals that the required system reliability or safety cannot be achieved within the current environment with the chosen interactions.

The iteration proceeds as follows. The analysis tool determines how severely each service-related exception affects system safety and service reliability. Among the service-related exceptions, the developer should start by addressing an exception that can be detected (with reasonable effort / costs), and for which a handling strategy that ensures safety or reliability can be envisioned. The exception to begin with might be one which requires immediate attention, or is safety-critical (in cases where safety is a priority). Then tasks 8 and 9 are executed, adding detection capabilities and handlers to the system. To complete this iteration, every interaction step of the newly defined handler of task 9 must again be elaborated (task 5.4), the outcomes must be specified (task 5.5), and the essential steps tagged with reliability probabilities and safety information (task 6.2)[6]. Finally, the developer can re-analyze the updated use cases (task 7) to determine if the required safety and reliability requirements can now be satisfied.

It should be noted here that the calculated dependability numbers do not represent the final system safety and reliability. They have to be interpreted as the *maximal dependability of the system if it were to be implemented without any flaws* (see section 5.2). Hence the calculated numbers should be *higher* than the required ones specified for each service in task 4.

[6] If new exceptional goals have been discovered, the developer might even be required to go back to task 1 to brainstorm new secondary actors that are needed to achieve the new exceptional goals.

Table 1. 407 ETR System: Exception Table

Exception	Description	Context	Handler	Detection
Government Computer Unavailable	System unable to access vehicle record	TakeHighway	Retry Govt Comp	Timeout on government computer
Operator Terminal Failure	Operator unable to communicate with government computer	TakeHighway	Retry Operator	Timeout on operator
Transponder Unreachable	System cannot communicate with the transponder	TakeHighway	WarnClient	Lack of acknowledgement / Timeout
Detector Failure	System does not get info on incoming vehicles from the vehicle detector	Process Unregistered Vehicle	Use Redundant Detector	Locator antenna detects vehicles
Locator Antenna Failure	System receives no message from locator antenna; unable to identify transponders	ProcessRegistered Vehicle	Use RW Antenna	Timeout on antenna
RW Antenna Failure	System does not receive transponder account information from the RW antenna	ProcessRegistered Vehicle	SwitchTo Spare-Antenna	Timeout on RW antenna
Light Failure	Lights failed to turn on when taking images	ProcessRegistered Vehicle, Process UnregisteredVehicle	Call Service Person	Bad images
Camera Failure	System does not receive images from the camera	Process Registered Vehicle, Process Unregistered Vehicle	Call Service Person	Timeout on camera device
Laser Scanner Failure	System unable to classify due to lack of message from scanner	Classify Vehicle	SwitchTo SpareScanner	Timeout on scanner
Highway Unavailable	System attempts to block access to parts of the highway	TakeHighway	ActivateBarriers	Operator request
Barrier Failure	The barrier fails to get activated	ActivateBarriers	Call Highway Patrol	Timeout on barrier

The difference between the calculated and the required values determines how much effort has to be put into the design and implementation phases. If the difference is small, then stringent quality assurance, such as formal methods and proofs, extensive testing, or fault tolerance techniques, has to be employed by the implementors in order to assure that the internal flaws of the system are minimal. Refinement, i.e. defining new detectors and handlers, therefore has to continue until the calculated dependability numbers are sufficiently higher than the required ones.

5.1 Tool Support

In order to use our dependability-aware requirements engineering process efficiently, tool support is necessary. This is especially true for the probabilistic analysis of system reliability and safety. DREP relies heavily on the idea of model-driven engineering (as defined by OMG [20]), in which models of the system under development are built, and then incrementally modified and transformed as the development progresses from requirements elicitation to analysis, design and implementation. At each phase, our process uses the modelling formalisms and notations that are most appropriate to express the concern at hand. The different modelling formalisms used in our requirements

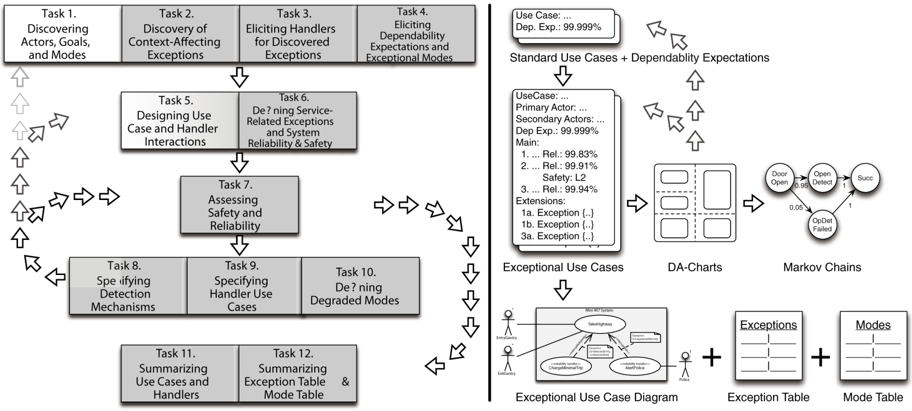

Task 1. Discovering Actors, Goals, and Modes
Task 2. Discovery of Context-Affecting Exceptions
Task 3. Eliciting Handlers for Discovered Exceptions
Task 4. Eliciting Dependability Expectations and Exceptional Modes
Task 5. Designing Use Case and Handler Interactions
Task 6. Defining Service-Related Exceptions and System Reliability & Safety
Task 7. Assessing Safety and Reliability
Task 8. Specifying Detection Mechanisms
Task 9. Specifying Handler Use Cases
Task 10. Defining Degraded Modes
Task 11. Summarizing Use Cases and Handlers
Task 12. Summarizing Exception Table & Mode Table
Use Case: ...
Dep. Exp.: 99.999%
Standard Use Cases + Dependablity Expectations
UseCase: ...
Primary Actor: ...
Secondary Actors: ...
Dep Exp.: 99.999%
Main:
1. ... Rel.: 99.83%
2. ... Rel.: 99.91%
Safety: L2
3. ... Rel.: 99.94%
Extensions:
1a. Exception {..}
1b. Exception {..}
3a. Exception {..}
Exceptional Use Cases
DA-Charts
Door Open
Open Detect
Succ
OpDet Failed
0.95
0.05
1
1
Markov Chains
Exceptional Use Case Diagram
Exceptions
Exception Table
Modes
Mode Table

Fig. 9. Dependable Requirement Engineering Process Summary and Used Modelling Formalisms

engineering process are shown in the right hand side of Fig. 9. The arrows depict the model transformations that occur when moving from one phase of the process to the next.

Currently, our tool [6] supports the creation of DA-Charts. The mapping to Markov chains and the dependability analysis is automated. We are working on the automated mapping of use cases to DA-Charts, and are even planning on providing automated support to map DA-Charts back to use cases. This would allow developers who are used to the DA-Chart formalism to apply reliability and safety increasing modification directly to the DA-Charts.

5.2 Discussion and Limitations

Our process helps the developer to discover potential exceptional situations that the system under development might be exposed to, and then guides the developer to investigate together with the stakeholder how the system should react in order to provide its services in the most reliable and safe way. Our approach is based on use cases that even non-technical people can read and understand, which makes getting feedback from all concerned stakeholders very easy.

As a result, however, our approach is also limited by the expressiveness of use cases. Use cases focus strictly on the interactions between the system and the environment. Hence, our dependability analysis only takes into account how the failures of actors and communication links affect the reliability and safety of the system under development. It does *not* consider failures internal to the system. The calculated dependability numbers represent the *best achievable safety and reliability of the system if it were implemented without any flaws*. If the numbers are too low, the developer should refine the interactions between the system and the actors, or even add new actors to the environment, to increase the achievable dependability of the system under development.

At this level of abstraction our approach cannot address internal flaws of the system. Use cases treat the system under development as a black box, and therefore no internal details are defined yet. Hence it is impossible to reason about conceptual system state, and even less to define invariants or pre- and postconditions on that state for system services.

To do this, a domain model describing conceptual system state must be created, and the use case models produced using our approach have to be mapped to operations that work with that system state. Popular development processes, e.g. the Unified Process [21], suggest to use graphical modelling formalisms such as activity diagrams or sequence diagrams for this purpose, together with OCL [22] constraints to express invariants, pre- and postconditions. If formal techniques are to be used to analyze system properties, a detailed system specification should be derived from our models using an appropriate formalism, e.g. B [23].

6 Validation

We conducted an empirical study in an academic environment using the 407 ETR case study to evaluate the applicability and effectiveness of our proposed process. We ran two separate experiments, both as part of an assignment in an undergraduate/graduate object-oriented software development course.

In the first experiment, a 3 hour lesson introduced a group of 20 students to use cases, after which they were asked to write use cases for the 407 ETR system using the standard use case template described in [15]. This first set of use cases is labelled *standard* in the following evaluation. Following the completion of this task, the students were presented with our exceptional use case notation as proposed in [19] in a one hour lecture. As a second part of the assignment, the students had to develop and extend their use cases by analyzing the cases for exceptional situations. They were required to document their results in an exception table, listing all exceptions discovered, their contexts, possible detection mechanisms, and handlers. This second set of use cases is labelled *EUC Notation* in the following evaluation.

The second experiment was carried out using the same case study, but with a different group of undergraduate/graduate software engineering students. In this case, the group was again introduced to use cases first (3 hours), but then presented with our dependability-driven requirements engineering process described in this paper (1 hour). Subsequently the students were asked to apply our task-based process to the 407 ETR system and elaborate use cases, handlers and a summarizing exception table following the guidelines outlined in Section 3.

As illustrated in the previous section, our specification of the 407 ETR included 10 exceptional scenarios. We went over the students submissions and analyzed the exceptions discovered by the members of each group. The results of the study are shown in Fig. 10 and 11.

Fig. 10 illustrates the measured improvements when using our proposed process as opposed to the standard use cases approach. In the first experiment using the standard use cases, about 67% of the students did not identify any exceptional situations. None of the students were able to identify context-affecting exceptions. Using DREP, students of the second experiment were able to discover an additional 64.45% exceptions on average, and 96% discovered *HighwayUnavailable*. It is not surprising that none of the students using standard use cases discovered the *BarrierFailure* exception, since this exception is revealed only after revisiting and analyzing a handler use case.

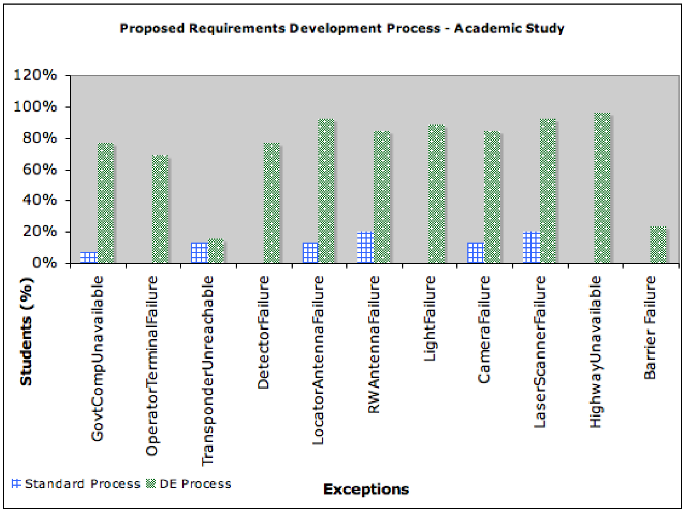


Fig. 10. Validation Results: Standard Use Cases versus RE Process

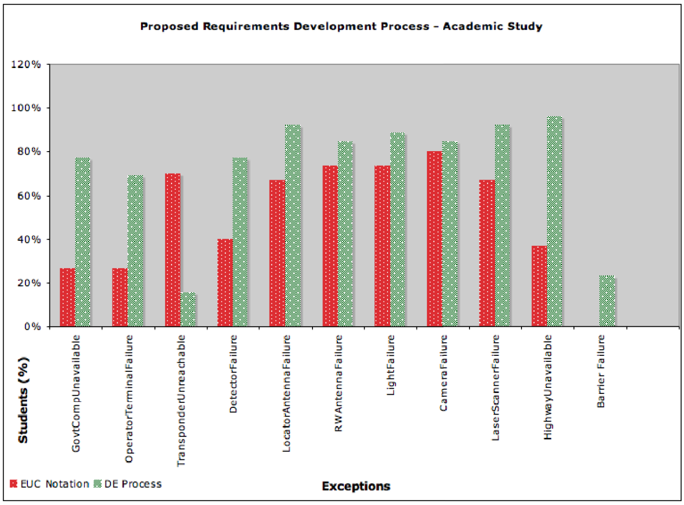


Fig. 11. Validation Results: Exceptional Use Cases versus RE Process

Fig. 11 shows that students who applied our proposed process achieved even better results than those who just used the exceptional use case notation. Using DREP, the students of the second experiment were able to discover an additional 21.3% of the exceptions compared to the students in the first experiment using the exceptional use case notation only. It is to be noted that, using the exceptional use case notation, the designer is not lead to carry out the refinement and iteration tasks, and hence the results obtained by students of the first experiment were based on one iteration only. Iterations (as suggested in task 8 and 9) would probably have improved the use cases further.

There was one surprise in the experiment results: using our process, fewer number of students were able to discover the exceptional situation *TransponderUnreachable*. We believe that this is due to the fact that in our process exceptions are discovered by analyzing the use case interactions step by step. If the standard use case is incomplete to begin with, i.e. if the analyst forgets a normal interaction, this omission propagates further on and associated exceptions do not appear. A careful inspection of main success scenario of the *TakeHighway* use case of the students of the second experiment revealed that indeed many students had forgotten to include the acknowledgment step where the system informs the driver of the successful completion of the transaction.

The successful classroom study can be taken as a good indication that our process has potential. It would of course be useful to conduct a similar experiment in an industrial setting. Organizations are often willing to test new ideas in small, low risk projects. But in our case we propose a process to aid in the development of dependable systems, or in other words high risk projects, and it seems to be more difficult to convince companies to try new development techniques.

7 Related Work

We have carried out an extensive literature overview of specialized software development methods, domain-specific frameworks and general-purpose middleware that address dependability, timeliness, adaptability, or other QoS requirements [24]. To the best of our knowledge, mainstream development methods currently address such concerns only at the late design and implementation phases. However, several specialized approaches have been proposed that consider such issues at the early phases.

The frameworks TIRAN and DepAuDE [24] are two significant contributions to the development of dependable systems, but cater to a specific domain. The TARDIS project [24] provides a general framework that addresses various non-functional requirements, but does not define a step-by-step development process.

Some approaches have also been proposed that consider exceptions or non-functional requirements during requirements elicitation, and they are briefly discussed here.

De Lemos et al. [2] emphasize the separation of the treatment of requirements-related, design-related, and implementation-related exceptions during the software lifecycle by specifying the exceptions and their handlers in the context where faults are identified. The description of exceptional behavior is supported by a cooperative object-oriented approach that allows the representation of collaborative behavior between objects at different phases of the software development.

Alexander [25] proposes using *misuse cases* to document and analyze negative scenarios, for example scenarios that threaten the security or safety of the system. The paper describes concepts and modelling constructs along with tool support that can be used for this purpose. However, the support for eliciting and analyzing exceptional situations is quite minimal, and requires much imagination and experience. Sindre et al. [26] define *misuse cases*, and provide methodological support for eliciting security requirements. Ebnenasir et al. [27] propose an approach based on *misuse cases* for modelling of failsafe fault tolerance. The approach introduces faults in a model and defines *unsafe* and *at risk* use cases to allow analysis.

Lamsweerde [28] proposes the KAOS method, which is a goal-oriented approach for requirements modelling, specification, and analysis. It addresses quality-of-service issues, and present a high-level approach for specifying requirements and deriving the design based on refinements. Exceptional behaviour, defined as *obstacles*, is also addressed during requirements engineering [29]. To begin with goals are elaborated using goal graphs, from which the functional requirements are derived. Obstacles are generated from the goal specifications. The obstacles are then analyzed and refined if needed. Strategies for resolving the obstacles are then defined, and the goal structure is updated with the newly introduced goals. Goals and obstacles are expressed in a formal temporal language, and thus it requires time and expertise to develop correct and complete specifications. [29] briefly discusses informal obstacle identification but detailed guidelines are not provided, and informal or semi-formal techniques for resolution and elimination of obstacles are not offered.

Laibinis et al. [30] uses redundancy patterns as support for integrating fault tolerance into use cases. They suggest refining use case diagrams with recovery measures using the standard UML notation. They only focus on error recovery mechanisms, and do not discuss detection techniques or requirements elicitation methods that need to be used.

Rubira et al. [31] present an approach that incorporates exceptional behavior in component-based software development by extending the Catalysis method. The requirements phase of Catalysis is also based on use cases, and the extension augments them with exception handling ideas.

Whittle et al. [32] focuses on representing cross cutting concerns (including non-functional concerns) during requirements development. The issues of aspect modelling in scenario-based requirements elicitation are addressed.

Leveson [33] presents the hazard analysis approach which is used as part of the safety-life-cycle process. The risks are realized by considering different failures classes, and then discovering the failures in the context of the system under development based on experience and domain knowledge. The causes of hazards are identified and analyzed by using techniques such as fault trees. Each hazard is then assigned a criticality level and a probability to enable risk assessment. In comparison to hazard analysis, we believe that our approach leads to more complete specifications with respect to safety and reliability concerns.

Fault trees are also part of the safety-life-cycle process which comprises of several phases starting from specification of safety requirements, to design and implementation of safety concerns of critical systems. The initial phase, hazard analysis and risk assessment [33], has goals similar to our exceptional use cases method. Hazard analysis is carried out to identify the risks, to determine the causes, and then to assess and mitigate the risks. The analysis is based on fault trees. The risks are realized by considering different failures classes, and then discovering the failures in the context of the system under development. The hazards are then associated with a criticality level and with the likelihood of occurrence. Such a technique requires much experience and expertise on the developers part. In comparison, our use-case based approach is intuitive and provides a systematic process that allows developers to identify exceptional situations by analysing the set of interactions between the actors and the system.

Our approach is different from the above for several reasons. Firstly, we help the requirements engineers to elicit, specify, analyze, and refine dependability issues, exceptions and handlers with a well-defined *process* that they can follow. DREP focuses on reliability and safety concerns specifically, and guides analysts to develop a requirements specification document that exhaustively addresses dependability expectations of the stakeholders. Without a process, the only way a developer can discover exceptions and define recovery measures is based on only his imagination and experience. Secondly, our process increases dependability by helping the developers detect the need for adding "feedback" and "acknowledgement" interaction steps to counter communication problems. Additionally, the process recommends adding hardware to monitor request execution of secondary actors when necessary. Our handler use cases are stand-alone, clearly separate exceptional behavior from standard behavior, and can be associated with multiple exceptions and multiple contexts. DREP also gives support for automatic dependability analysis with tool support. The process is based on semi-formal constructs, and developers do not require expertise in formal specification languages to define or determine the quality of their requirements. In addition, communicating with end-users is simpler with use cases.

8 Conclusion

In most software systems today, it is crucial to guarantee that dependability requirements are successfully achieved. The discovery of all reliability and safety concerns is essential for the development of dependable systems. Exceptional situations are less common and the required behavior of the system in such situations is less obvious, and hence detailed user feedback on expected system behavior in such situations is very important. Also, users are more likely to make mistakes when exposed to exceptional situations, and therefore system interaction during handling of an exceptional situation is to be designed with great care. Early discovery of dependability concerns allow developers to discover and then document how the users of the system expect the system to react in every situation, which ultimately results in a more dependable system and saves considerable development costs. To this aim, we propose an approach that extends use case-based requirements elicitation, focussing on system reliability and safety.

Our task-based process begins with eliciting user goals and dependability expectations, and then discovering context-affecting exceptions. Recovery goals that dictate the system behaviour in such situations are defined in *handler use cases*. The process then goes on to give guidance on designing interactions to satisfy each of the discovered user and handler goals. The defined use cases are then examined step-by-step for reliability and safety related issues.The identified issues are labelled and documented as exceptions. As the next phase, the process suggests to analyze the use cases by using a probabilistic dependability analysis technique. The assessment results might show the need for increasing the dependability, and hence lead to further refinement. The refinement tasks require revisiting the use cases and integrating appropriate exception detection and recovery means. In the final task, the use cases, exceptions, and handlers are summarized in graphical and textual forms.

Based on our dependability focused use cases, a specification that considers all exceptional situations and user expectations can be elaborated during a subsequent analysis phase. This specification can then be used to decide on the need for employing fault masking and fault tolerance techniques when designing the software architecture and during detailed design of the system.

For future work, we intend to extend DREP to address other dependability constraints like availability and timeliness. We also plan to continue the development of our tool analysis tool to support DREP by providing a visual modelling environment for our dependability-focused use cases, and allow automatic mapping to analysis models for dependability assessment.

References

1. Goodenough, J.B.: Exception handling: Issues and a proposed notation. Communications of the ACM 18(12), 683–696 (1975)
2. de Lemos, R., Romanovsky, A.: Exception handling in the software lifecycle. IJCSSE 16(2), 167–181 (2001)
3. Shui, A., Mustafiz, S., Kienzle, J.: Exceptional use cases. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 568–583. Springer, Heidelberg (2005)
4. Shui, A., Mustafiz, S., Kienzle, J.: Exception-Aware Requirements Elicitation with Use Cases. In: Dony, C., Knudsen, J.L., Romanovsky, A., Tripathi, A.R. (eds.) Advanced Topics in Exception Handling Techniques. LNCS, vol. 4119, pp. 221–242. Springer, Heidelberg (2006)
5. Mustafiz, S., Sun, X., Kienzle, J., Vangheluwe, H.: Model-Driven Assessment of Use Cases for Dependable Systems. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 558–573. Springer, Heidelberg (2006)
6. Zia, M., Mustafiz, S., Vangheluwe, H., Kienzle, J.: A Modelling and Simulation Based Process for Dependable Systems Design. In: Software and Systems Modeling, pp. 437–451 (April 2007)
7. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd edn. Prentice Hall, Englewood Cliffs (2002)
8. Laprie, J.C., Avizienis, A., Kopetz, H. (eds.): Dependability: Basic Concepts and Terminology. Springer, New York (1992)
9. Geffroy, J.C., Motet, G.: Design of Dependable Computing Systems. Kluwer Academic Publishers, Dordrecht (2002)
10. Avizienis, A., Laprie, J., Randell, B.: Fundamental concepts of dependability (2001)
11. Knudsen, J.L.: Better exception-handling in block-structured systems. IEEE Software 4(3), 40–49 (1987)
12. Dony, C.: Exception handling and object-oriented programming: Towards a synthesis. In: Meyrowitz, N. (ed.) 4th ECOOP 1990. ACM SIGPLAN Notices, vol. 25, pp. 322–330. ACM Press, New York (1990)
13. Cockburn, A.: Writing Effective Use Cases. Addison-Wesley, Reading (2000)
14. Mustafiz, S., Kienzle, J., Berlizev, A.: Addressing degraded service outcomes and exceptional modes of operation in behavioural models. In: Proceedings of the International Workshop on Software Engineering for Resilient Systems (SERENE 2008). ACM, New York (2008)
15. Sendall, S., Strohmeier, A.: UML-based fusion analysis. In: France, R.B., Rumpe, B. (eds.) UML 1999. LNCS, vol. 1723, pp. 278–291. Springer, Heidelberg (1999)

16. Mustafiz, S., Sun, X., Kienzle, J., Vangheluwe, H.: Model-driven assessment of use cases for dependable systems. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 558–573. Springer, Heidelberg (2006)
17. de Lara, J., Vangheluwe, H.: AToM^3: A tool for multi-formalism and meta-modelling. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 174–188. Springer, Heidelberg (2002)
18. Mustafiz, S., Sun, X., Kienzle, J., Vangheluwe, H.: Model-driven assessment of system dependability. In: Software and Systems Modeling (SoSym) (March 2007)
19. Shui, A., Mustafiz, S., Kienzle, J., Dony, C.: Exceptional use cases. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 568–583. Springer, Heidelberg (2005)
20. Mukerji, J., Miller, J.: Mda guide v1.0.1 (2003)
21. Jacobson, I., Rumbaugh, J., Booch, G.: The Unified Software Development Process. Object Technology Series. Addison–Wesley, Reading (1999)
22. Warmer, J., Kleppe, A.: The Object Constraint Language, 2nd edn. Object Technology Series. Addison–Wesley, Reading (2003)
23. Abrial, J.R.: The B Book - Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
24. Mustafiz, S., Kienzle, J.: A survey of software development approaches addressing dependability. In: Guelfi, N., Reggio, G., Romanovsky, A. (eds.) FIDJI 2004. LNCS, vol. 3409, pp. 78–90. Springer, Heidelberg (2005)
25. Alexander, I.F.: Misuse cases: Use cases with hostile intent. IEEE Software 20(1), 58–66 (2003)
26. Sindre, G., Opdahl, A.L.: Eliciting security requirements with misuse cases. Requir. Eng. 10(1), 34–44 (2005)
27. Ebnenasir, A., Cheng, B.H.C., Konrad, S.: Use case-based modeling and analysis of failsafe fault-tolerance. In: RE, pp. 336–337. IEEE Computer Society, Los Alamitos (2006)
28. van Lamsweerde, A.: Goal-oriented requirements engineering: A guided tour. In: RE, p. 249. IEEE Computer Society, Los Alamitos (2001)
29. van Lamsweerde, A., Letier, E.: Handling obstacles in goal-oriented requirements engineering. IEEE Trans. Software Eng. 26(10), 978–1005 (2000)
30. Laibinis, L., Troubitsyna, E.: Fault tolerance in use-case modeling. In: Proceedings of RHAS 2005 (September 2005)
31. Rubira, C.M.F., de Lemos, R., Ferreira, G.R.M., Fliho, F.C.: Exception handling in the development of dependable component-based systems. Software – Practice & Experience 35(3), 195–236 (2004)
32. Whittle, J., Araújo, J.: Scenario modelling with aspects. IEE Proceedings - Software 151(4), 157–172 (2004)
33. Leveson, N.G.: SAFEWARE: System Safety and Computers. Addison-Wesley Publishing Company, Reading (1995)

Documenting the Progress of the System Development*

Marta Pląska[1], Marina Waldén[1], and Colin Snook[2]

[1] Åbo Akademi University/TUCS, Joukahaisenkatu 3-5A, 20520 Turku, Finland
[2] University of Southampton, Southampton, SO17 1BJ, UK

Abstract. While UML gives an intuitive image of the system, formal methods provide the proof of its correctness. We can benefit from both aspects by combining UML and formal methods. Even for the combined method we need consistent and compact description of the changes made during the system development. In the development process certain design patterns can be applied. In this paper we introduce progress diagrams to document the design decisions and detailing of the system in successive refinement steps. A case study illustrates the use of the progress diagrams.

Keywords: Progress diagram, Statemachines, Stepwise development, Refinement, Refinement Patterns, UML, Event-B, Action Systems, Graphical representation.

1 Introduction

For complex systems the stepwise development approach of formal methods is beneficial, especially considering issues of ensuring the correctness of the system. However, formal methods are often difficult for industrial practitioners to use. Therefore, they need to be supported by a more approachable platform. The Unified Modelling Language (UML) is commonly used within the computer industry, but currently, mature formal proof tools are not available. Hence, we use formal methods in combination with the semi-formal UML.

For a formal top-down approach we use the Event B formalism [11] and associated proof tool to develop the system and prove its correctness. Event-B is based on Action Systems [4] as well as the B Method [1], and is related to B Action Systems [22]. With the Event-B formalism we have tool support for proving the correctness of the development. In order to translate UML models into Event B, the UML-B tool [18, 19] is used. UML-B is a specialisation of UML that defines a formal modelling notation combining UML and B.

The first phase of the design approach is to state the functional requirements of the system using natural language illustrated by various UML diagrams, such as statechart diagrams and sequence diagrams that depict the behaviour of the system. The system is built up gradually in small steps using superposition refinement [3, 10]. We rely on patterns in the refinement process, since these are the cornerstones for creating *reusable* and *robust* software [2, 7]. UML diagrams and corresponding Event B code are developed for each step simultaneously. To get a better overview of the design process, we introduce the *progress diagram*, which illustrates only the refinement-affected parts of the system and is based on statechart diagrams. Progress diagrams support the

* Work done within the RODIN-project, IST-511599.

M. Butler et al. (Eds.): Fault Tolerance, LNCS 5454, pp. 251–274, 2009.
© Springer-Verlag Berlin Heidelberg 2009

construction of large software systems in an incremental and layered fashion. Moreover, they help to master the complexity of the project and to reason about the properties of the system. We illustrate the use of the diagrams with a case study.

The rest of the paper is organised as follows. In Section 2 we give an overview of our case study, Memento, from a general and functional perspective. An abstract specification is presented as a graphical, as well as a formal representation in Section 3. Section 4 describes stepwise refinement of systems and gives refinement patterns and Section 5 introduces the idea of progress diagrams. The system development is analysed and illustrated with the progress diagrams relying on the case study in Section 6. The related work is presented in Section 7. We conclude with some general remarks and our future work in Section 8.

2 Case Study – Memento Application

The *Memento* application [14] that is used as a case study in this paper is a commercial application developed by *Unforgiven.pl*. It is an organiser and reminder system that has evolved into an internet-based application. Memento is designed to be a framework for running different modules that interact with each other.

In the distributed version of Memento every user of the application must have its own, unique identifier, and all communication is done via a central application server. In addition to its basic reminder and address book functions, Memento can be configured with other function modules, such as a simple chat module. Centralisation via the use of a server allows the application to store its data independently of the physical user location, which means that the user is able to use his own Memento data on any computer that has access to the network.

The design combines the web-based approach of internet communicators and an open architecture without the need for installation at client machines. During its start-up the client application attempts to *connect to a central server*. When the connection is established, the *preparation phase* begins. In this phase the user provides his/her unique identifier and password for authorisation. On successful login the server responds by sending the data for the account including a list of contacts, news, personal files etc. Subsequently the *application searches for modules* in a working folder and attempts to initialise them, so that the user is free to run any of them at any time. During execution of the application, *commands from the server and the user are processed* at once. Memento translates the requested actions of the user to internal commands and then handles them either locally or via the server. Upon a termination command Memento *finalises all the modules*, saves the needed data on the server, logs out the user and *closes the connection*. To minimise the risk of data loss, in case of fatal error, this termination procedure is also part of the fatal exception handling routine.

3 Abstract Specification

3.1 UML-Models

We use the Unified Modelling Language™ (UML) [5], as a way of modelling not only the application structure, behaviour, and architecture of a system, but also its data structure. UML can be used to overcome the barrier between the informal industry world and the formal one of the researchers. It provides a graphical interface and documentation for every stage of the (formal) development process. Although UML offers miscellaneous diagrams for different purposes, we focus on two types of these in our paper: sequence diagrams and statechart diagrams.

The sequence diagram is used within the development of the system to show the interactions between objects and the order in which these interactions occur. The diagram can be derived directly from the requirements. Furthermore, it may give information on the transitions of the statemachines. The interaction between entities in the sequence diagram can be mapped to self-transitions on the statechart diagram to model communication between the modelled entity and its external entities.

In our case study the external entities are the server and the users interacting with the modelled entity Memento. An example of a sequence diagram for the application is given in Fig. 1, where part of the requirements (the emphasized text in Section 2) concerning the server connection and the program preparation phase is shown. In the diagram we describe the initialisation phase of the system, which consists of establishing a connection (in the connection phase) and then preparing the program (in the preparation phase). The first of these actions requires the interaction with the server through an internet connection. The second action requires communication with user as well. The described interaction (in Fig. 1) is transferred to a statechart diagram as transition *tryInit* (to later be refined to the transitions *tryConn* and *tryPrep* as explained in Section 6).

In statechart diagrams objects consist of states and behaviours (transitions). The state of an object depends on the previous transition and its condition (guard). A statechart diagram provides a graphical description of the transitions of an object from one state to another in response to events [12, 13]. The diagram can be used to illustrate the behaviour of instances of a model element. In other words, a statechart diagram shows the possible states of the object and the transitions between them.

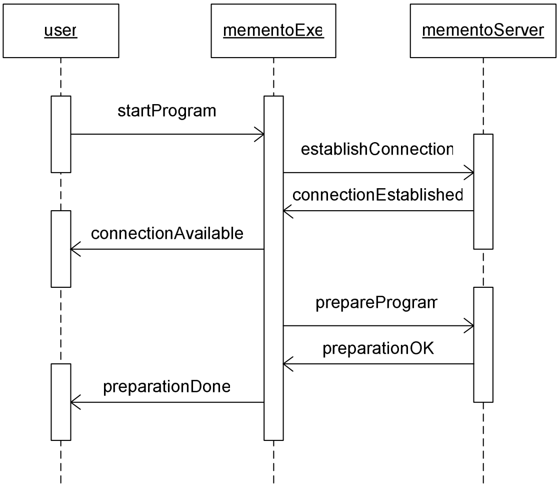


Fig. 1. Sequence diagram presenting the object interaction in the *initialisation phase*

The statemachine depicting the abstract behaviour of Memento is shown in Fig. 2. The first phase is to initialise the system by communicating with the server. It is modelled with the event *tryInit*. When initialisation has been successfully completed, the transition *goReady* brings the system to the state ready, where it awaits and processes the user and server commands. Upon the command *close*, the system enters the finalisation phase, which leads to the system cleanup and proper termination.

The detection of errors in each phase is taken into consideration. In the model, the errors are captured by transitions targeting the suspended state (*susp*), where error handling (rollback) takes place. The system may return to the state where the error was detected, if the error happens to be recoverable. If the error is non-recoverable, the fatal termination action is taken and the system operation finishes. Any error detected during or after finalisation phase is always non-recoverable.

We use the following notation for the transitions in statechart diagrams and in the Event-B code in the rest of the paper. The symbols ‘::’ and ‘:∈ ’ stand for non-deterministic assignment and are applied interchangeably, in the diagrams and the code, respectively. The symbol ‘:=’ is used in assignments, whereas ‘||’ symbol denotes that the operands are executed concurrently. All of the mentioned symbols are placed in the statement parts. By the use of a junction pseudo state [20] (marked with angled brackets ‘<>’) that denotes the old, refined transition, we indicate the refinement relation between new transitions and previous abstract ones.

3.2 Formal Specification

In order to be able to reason formally about the abstract specification, we translate it to the formal language Event B [11]. An Event-B specification consists of a model and its context that depict the dynamic and the static part of the specification, respectively. They are both identified by unique names. The context contains the sets and constants of the model with their properties and is accessed by the model through the SEES relationship [1]. The dynamic model, on the other hand, defines the state

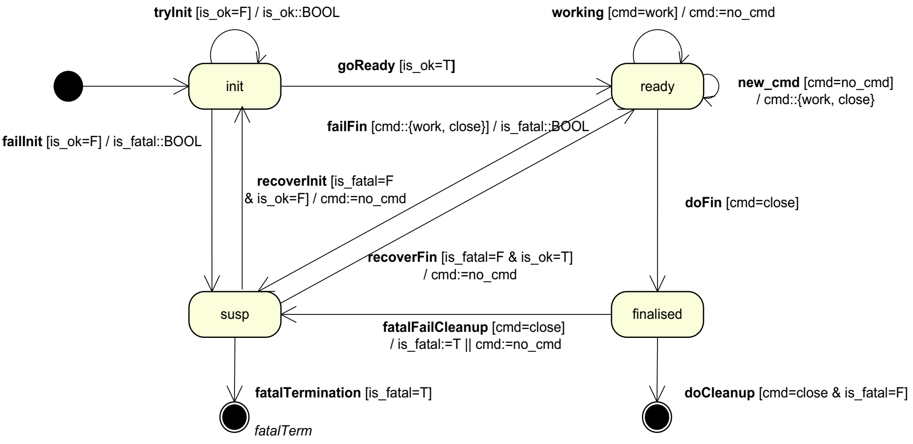

tryInit [is_ok=F] / is_ok::BOOL
working [cmd=work] / cmd:=no_cmd
init
goReady [is_ok=T]
ready
new_cmd [cmd=no_cmd] / cmd::{work, close}
failFin [cmd::{work, close}] / is_fatal::BOOL
failInit [is_ok=F] / is_fatal::BOOL
recoverInit [is_fatal=F & is_ok=F] / cmd:=no_cmd
doFin [cmd=close]
recoverFin [is_fatal=F & is_ok=T] / cmd:=no_cmd
susp
finalised
fatalFailCleanup [cmd=close] / is_fatal:=T || cmd:=no_cmd
fatalTermination [is_fatal=T]
doCleanup [cmd=close & is_fatal=F]
fatalTerm

Fig. 2. The abstract statemachine of Memento

variables, as well as the operations on these. Types and properties of the variables are given in the invariant. All the variables are assigned an initial value according to the invariant. The operations on the variables are given as events of the form **WHEN** guard **THEN** substitution **END** in the Event-B specification. When the guard evaluates to true the event is said to be enabled. If several events are enabled simultaneously any one of them may be chosen non-deterministically for execution. The events are considered to be atomic, and hence, only their pre and post states are of interest. In order to be able to ensure the correctness of the system, the abstract model should be consistent and feasible [11].

Each transition of a statechart diagram is translated to an event in Event-B. Below we show the Event B-translation of the statemachine concerning the initialisation (state *init*) of the cooperation with the server in Fig. 2:

```
    MACHINE          Memento
    SEES             Data
    VARIABLES        is_fatal, is_ok, cmd, state
    INVARIANT        is_fatal ∈ BOOL ∧ is_ok ∈ BOOL ∧ cmd ∈ CMD ∧ state ∈ STATE ∧
                     (state=init ⇒ cmd=no_cmd) ∧ ...
    INITIALISATION   is_fatal:=FALSE || cmd:=no_cmd || is_ok:=FALSE || state:=init
    EVENTS
        tryInit =    WHEN state=init ∧ is_ok=FALSE  THEN is_ok :∈ BOOL END;
        failInit =   WHEN state=init ∧ is_ok=FALSE  THEN state:=susp || is_fatal :∈ BOOL END;
        recoverInit= WHEN state=susp ∧ is_ok=FALSE ∧ is_fatal=FALSE THEN state:=init || cmd:=no_cmd
END;
        goReady =    WHEN state=init ∧ is_ok=TRUE  THEN state:=ready END;
        ...
    END
```

The variables model a proper initialisation (*is_ok*), occurrence of a fatal error (*is_fatal*), as well as the command (*cmd*) and the state of the system (*state*). Initially no command is given and the initialisation phase is marked as not completed (*is_ok := FALSE*). The guards of the transitions in the statechart diagram in Fig. 2 are transformed to the guards of the events in the Event B model above, whereas the substitutions in the transitions are given as the substitutions of the events. The feasibility and the consistency of the specification are then proved using the Event-B prover tool.

4 System Refinement and Refinement Patterns

It is convenient not to handle all the implementation issues at the same time, but to introduce details of the system to the specification in a stepwise manner. Stepwise refinement of a specification is supported by the Event-B formalism. In the refinement process an abstract specification *A* is transformed into a more concrete and deterministic system *C* that preserves the functionality of *A*. We use the superposition refinement technique [3, 11, 22], where we add new functionality, i.e., new variables and substitutions on these, to a specification in a way that preserves the old behaviour. The variables are added gradually to the specification with their conditions and properties. The computation concerning the new variables is introduced in the existing events by strengthening their guards and adding new substitutions on these variables. New events, assigning the new variables, may also be introduced.

4.1 Refinement of the System

System *C* is said to be a correct refinement of *A* if the following proof obligations are satisfied [11, 20, 22]:

1. The initialisation in *C* should be a refinement of the initialisation in *A*, and it should establish the invariant in *C*.
2. Each old event in *C* should refine an event in *A*, and preserve the invariant of *C*.
3. Each new event in *C* (that does not refine an event in *A*) should only concern the new variables, and preserve the invariant.
4. The new events in *C* should eventually all be disabled, if they are executed in isolation, so that one of the old events is executed (non-divergence).
5. Whenever an event in *A* is enabled, either the corresponding event in *C* or one of the new events in *C* should be enabled (strong relative deadlock freeness).
6. Whenever an error detection event (event leading to the state *susp)* in *A* is enabled, an error detection event in *C* should be enabled (partitioning an abstract representation of an error type into distinct concrete errors during the refinement process [21]).

The tool support provided by Event-B allows us to prove that the concrete specification *C* is a refinement of the abstract specification *A* according to the Proof Obligations (1) - (6) given above.

4.2 Modelling Refinement Patterns

In order to guide the refinement process and make it more controllable, refinement patterns [12] can be applied. We are using the following notation for the patterns in the rest of the paper. A typical event consists of a guard *G(V)* and an action *S(V)*, where *G(V)* is some supplementary predicate on the variables *V,* often represented as a conjunction of several individual guards, and *S(V)* is some supplementary assignment of the variables *V*. The variables *V* are of a general type (TYPE). For instance, in the Event-B code for the refinement EX3c $G_i(y)$ denotes a guard on variable *y* of the general type TYPE, while $S_i(y)$ denotes some assignment of variable *y*.

In all the pattern diagrams (except in the choice paths in Fig. 5) we omit the guards on the transitions for better readability of the diagrams. The code added in the current refinement step is indicated by a darker background.

4.2.1 Refining the States

Let us first concentrate on the abstract specification given in Fig. 3a. It is pictured by a statechart diagram consisting of two states (*st1* and *st2*), a self transition *tr1* for the state *st1* and a transition *tr2* from state *st1* to the state *st2*. We are focusing on data refinement and event refinement patterns enabled by the data refinement. The former is shown in the statechart diagram in Fig. 3b (splitting states into substates and adding transitions between them), while an example of the latter is given in Fig. 3c (splitting existing transitions).

Splitting the states and adding new transitions are commonly performed in one refinement step. The two steps shown in Figures 3b and 3c are shown separately here only to depict the details of the complete data and event refinement. Generally, when

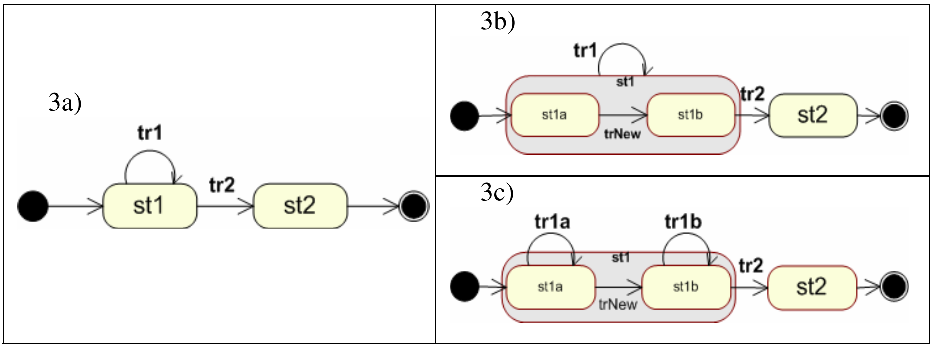


Fig. 3. Refinement patterns – basic data and event refinements

refining the states, we want to add some new features/variables at the same time as we split the transitions.

For the abstract specification depicted in Fig. 3a we have the following Event-B specification code:

```
MACHINE         EX3a
VARIABLES       state
INVARIANT       state ∈ {st1, st2}
INITIALISATION  state:=st1
EVENTS
    tr1 =       WHEN state=st1 THEN state:=st1      END;
    tr2 =       WHEN state=st1 THEN state:=st2      END
END
```

The Event-B code for the pattern concerning the data refinement, i.e. splitting the states, is illustrated in the Fig. 3b as follows:

```
REFINEMENT      EX3b
REFINES         EX3a
VARIABLES       state, state1
INVARIANT       state ∈ {st1, st2} ∧ state1 ∈ {st1a, st1b}
INITIALISATION  state:=st1 || state1:=st1a
EVENTS
    tr1 =       WHEN state=st1               THEN state:= st1 END;
    trNew =     WHEN state=st1 ∧ state1=st1a THEN state1:=st1b END;
    tr2 =       WHEN state=st1 ∧ state1=st1b THEN state:=st2 END
END
```

The pattern for separating an existing transition (event refinement) corresponding to the diagram in Fig. 3c is as follows:

```
REFINEMENT      EX3c
REFINES         EX3a
VARIABLES       state, state1, y, z
INVARIANT       state ∈ {st1, st2} ∧ state1 ∈ {st1a, st1b} ∧ y∈TYPE ∧ z∈TYPE ∧ I(y,z)
INITIALISATION  state:=st1 || state1:=st1a || y:∈TYPE || z:∈TYPE
EVENTS
    tr1a (refines tr1) =    WHEN state=st1 ∧ state1=st1a ∧ $G_{1a}(y)$
                            THEN state:=st1 || state1:=st1a || $S_{1a}(y)$ END;
    tr1b (refines tr1) =    WHEN state=st1 ∧ state1=st1b ∧ $G_{1b}(z)$
                            THEN state:=st1 || state1:=st1b || $S_{1b}(z)$ END;
    trNew =                 WHEN state=st1 ∧ state1=st1a ∧ $G_N(y)$
                            THEN state1:=st1b END;
    tr2 =                   WHEN state=st1 ∧ state1=st1b ∧ $G_2(z)$
                            THEN state:=st2 END
END
```

Each of the refined event uses some of the new variables (*y* and *z*) in its guards (*G(y)* and *G(z)*) and actions (*S(y)* and *S(z)*) to reduce non-determinism. The guards $G_{1a}(y)$ and $G_n(y)$ are created in such a way that they guarantee progress. In the same manner $G_n(y)$ should imply $G_2(z)$ or $G_{1b}(z)$, moreover guards $G_2(z)$ and $G_{1b}(z)$ should also be formed to guarantee the progress of the system. When inserting conditions on new properties to the guards, the failure management is in general also refined in a corresponding manner in order to design a fault tolerant behaviour. Nevertheless, here we concentrate our patterns on the proper and desirable behaviour of the system. An example of another pattern of the basic data and event refinement including failure management is given by Snook and Waldén [20]. In that pattern a loop is created in the superstate.

In order to give an intuition of the correctness of the patterns, we state how the Proof Obligation Rules given in Section 4.1 are satisfied by the patterns. Moreover, the Proof Obligations hint at how problems in the program design can be detected more easily.

According to the Proof Obligations (1) and (2) in Section 4.1 the initialisation and the events are refined to take the new variables into consideration. The guards of the old events may be strengthened and assignments concerning the new variables added. During the system development we may also want to refine an existing event by splitting it into several separate events. As acknowledged in Proof Obligation (3), the new events are only permitted to assign the new variables, but may, however, refer to the old variables. The guard of the new event should be composed in such a way that the new event, together with the refined events, ensures progress of the system (Proof Obligation (5)).

The new events should not take over the execution (Proof Obligation (4)), which can be assured by disallowing the new transitions in the statemachine diagram (corresponding to new events in the Event-B model) from forming a loop. The Event-B prover requires an expression, called a ‘variant’, that gives a Natural number that is decreased by all of the new transitions between the substates. To deduce a suitable variant, graph theory is applied. The states of the statemachine representing the system are numbered according to the minimum path length to a refining transition. Hence, if the new transitions form a sequence that progresses towards a refining transition, the function that defines this numbering for each state will be strictly decreased as the state changes. If loops are unavoidable in the new events, the auxiliary variables must be used in the variant in order to provide a suitable variant that decreases throughout the loop. Each new transition has to lead to a new state with a lower designated number or (in case of loops) alter the auxiliary state variables, thereby decreasing the variant. To avoid deadlock, a route from every new transition to one that refines an old transition should exist. This is a necessary, but not sufficient, condition for relative deadlock freeness. If there exists no sequence of new transitions which can reach one that refines an old transition, meaning there is no route, then new events terminate (without enabling an old event) and a new deadlock is introduced [20].

As properties are added to the system, the potential failure management should also be refined. If a fault appears at a substate it should be viable to return to that substate after recovery. This can be achieved by dividing an abstract failure into more specific failures on the new features in conformance with Proof Obligation (6). Note that new failure situations are not introduced in our case, as it is a general pattern that can be adjusted to the specific needs.

4.2.2 Flattening States

When refining the system by superposition and at the same time splitting the states in a hierarchical manner, we have to deal with the states that are nested in the superstate due to the consequent refinement steps. This development, although performed in a stepwise manner, at some point makes the system model unreadable. Therefore we apply the flattening pattern, which removes the most external superstate, leaving the substates intact.

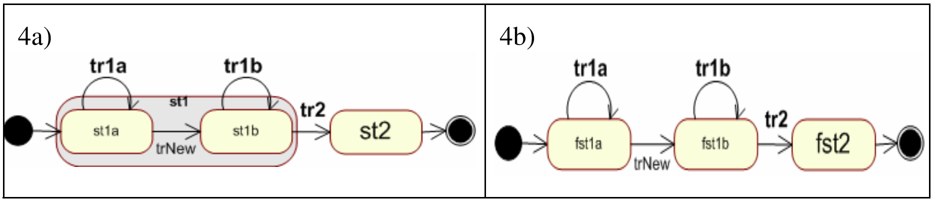


Fig. 4. Refinement pattern – flattening of the hierarchical states

In Fig. 4 we present the flattening pattern applied to the model from Fig. 3c. In Fig. 4a we model the hierarchical structure of states, i.e. state *st1* is a superstate for the states *st1a* and *st1b*. By applying the flattening pattern, we remove the superstate *st1*. This is possible, when giving an appropriate invariant preserving the relation between the states in the model, namely relating the states from the old model to the states in the new model. This is correlated with the change of the naming of the variables in order to preserve the invariant.

Note that flattening can only be performed once the parent state is neither the source nor target of any transitions. That is, other patterns should first be applied to move all the parents' transitions to its substates so that the parent state is completely redundant.

Here we show the Event-B code for the refined model:

```
REFINEMENT      EX4b
REFINES         EX4a
VARIABLES       newState, y, z, v
INVARIANT       newState ∈ {fst1a, fst1b, fst2} ∧ newState ∈ NEWSTATE
                ∧ y∈TYPE ∧ z∈TYPE ∧ v∈TYPE ∧ R4(y,z,v) ∧
                newState=fst1a ⇔ (state=st1 ∧ state1=st1a) ∧
                newState=st1b ⇔ (state=st1 ∧ state1=st1b) ∧ state=st2 □ newState=fst2 ∧
                newState=fst2 □ state=st2 ∧ state=st1 □ (newState=fst1a □ fst1b)
INITIALISATION  newState:=fst1a || v:∈TYPE || y:∈TYPE || z:∈TYPE
EVENTS
    tr1a (refines tr1) =      WHEN newState=fst1a ∧ G1a(y) ∧ G1a(z) ∧ G1a(v)
                              THEN newState:=fst1a || S1a(y) || S1a(z) || S1a(v) END;
    tr1b (refines tr1) =      WHEN newState=fst1b ∧ G1b(y) ∧ G1b(z) ∧ G1b(v)
                              THEN newState:=fst1b || S1b(y) || S1b(z) || S1b(v) END;
    trNew =                   WHEN newState=fst1a ∧ Gnr(y) ∧ Gnr(z) ∧ Gnr(v)
                              THEN newState:=fst1b || Snr(y) || Snr(z) || Snr(v) END;
    tr2 =                     WHEN newState=fst1b ∧ G2r(y) ∧ G2r(z) ∧ G2r(v)
                              THEN newState:=fst2 || S2r(y) || S2r(z) || S2r(v) END
END
```

Since flattening the state hierarchy is rather a rewriting step than a refinement step, the proof obligations in Section 4.1 trivially hold. We rely on the invariant giving the relation between the flattened state and the hierarchical states.

4.2.3 Separating Existing Transitions – Choice Paths

In order to perform event refinement, particularly to separate existing events, we can split the transition into alternative paths using the black diamond-shaped choice symbol (salmiakki) [20], where each choice is responsible for a separate event. The guards on the events are strengthened by the choice points. Each choice represents a separate event whose guard includes the conjunction of all the segments leading up to that path. Thus, the guard enabling a given event is the conjunction of all the conditions of choice paths leading up to the choice point.

Fig. 5 illustrates a simple pattern for adding features to the specification and expanding its functionality. More specifically, the transition *tr1* is refined by two branches - transitions *tr1a* and *tr1b*. This could, for example, model the refinement of a non-deterministic event '*move*' to the more specific events '*move_forward*' and '*move_backward*'.

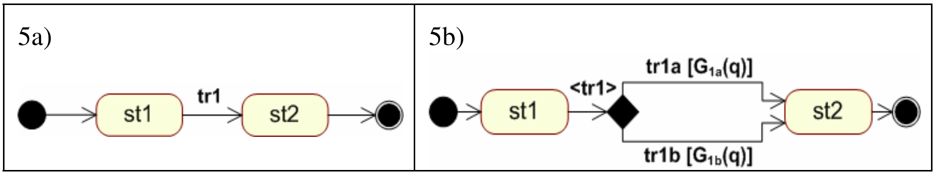


Fig. 5. Refinement pattern – event refinement: simple choice paths

The Event-B code corresponding to this pattern for the abstract machine is as follows:

```
MACHINE          EX5a
VARIABLES        state
INVARIANT        state ∈ {st1, st2 }
INITIALISATION   state:=st1
EVENTS
     tr1 =       WHEN state=st1 THEN state:=st2 END
END
```

The refinement can be expressed as the following Event-B machine:

```
REFINEMENT       EX5b
REFINES          EX5a
VARIABLES        state, q
INVARIANT        state ∈ {st1, st2 } ∧ q ∈ TYPE ∧ I(q)
INITIALISATION   state:=st1 || q :∈ TYPE
EVENTS
     tr1a (refines tr1) =     WHEN state=st1 ∧ G1a(q)     THEN state:=st2 || S1a(q)END;
     tr1b (refines tr1) =     WHEN state=st1 ∧ G1b(q)     THEN state:=st2 || S1b(q) END
END
```

Guards of the transition *tr1* are strengthened via choice point, according to Proof Obligation (2). When splitting the transition *tr1* into two more specific transitions *tr1a* and *tr1b* we should ensure the progress of the system, fulfilling Proof Obligation (5).

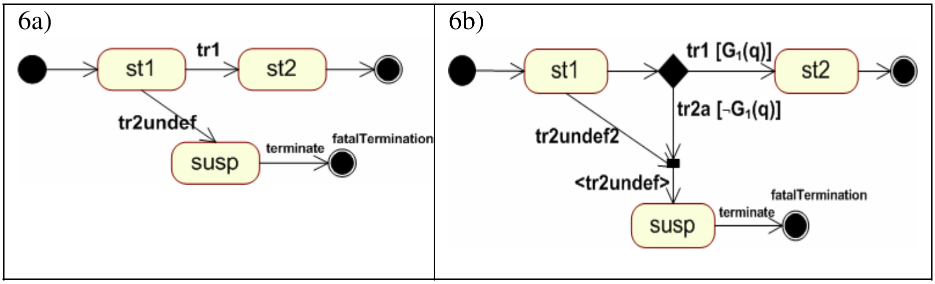


Fig. 6. Refinement pattern – event refinement: choice paths

With the choice paths pattern we can also show more detailed failure management. Fig. 6 depicts the creation of the choice path on the transition *tr1*, thus detailing the error detection. The transition *tr1* is refined by strengthening its guards concerning the new variable q $(G_1(q))$. The transition *tr2undef* is refined into transition *tr2undef2*, which stands for the detection of undetermined errors and *tr2a* which is modelling a particular type of failures $(\neg G_1(q))$.

The Event-B code for the abstract machine that we use as an example for choice paths pattern (event refinement) is as follows:

```
MACHINE          EX6a
VARIABLES        state
INVARIANT        state ∈ {st1, st2, susp}
INITIALISATION   state:=st1
EVENTS
    tr1 =        WHEN state=st1 THEN state:=st2 END;
    tr2undef =   WHEN state=st1 THEN state:=susp END
END
```

The refined model is expressed as an Event-B model given below:

```
REFINEMENT       EX6b
REFINES          EX6a
VARIABLES        state, q
INVARIANT        state ∈ {st1, st2, susp} ∧ q ∈ TYPE ∧ J(q)
INITIALISATION   state:=st1 || state1:=st1a || q :∈ TYPE
EVENTS
    tr1 =                            WHEN state=st1 ∧ G1(q)    THEN state:=st2 || S1(q)     END;
    tr2undef2 (refines tr2undef) =   WHEN state=st1            THEN state:=susp             END;
    tr2a (refines tr2undef) =        WHEN state=st1 ∧ ¬G1(q)   THEN state:=susp || S2a(q)   END
END
```

We use the join (black bar symbol) to illustrate refinement of the failure transition *tr2undef*, which is split into two different failures, *tr2undef2* and *tr2a,* in accordance with Proof Obligation (6). The guard for transition *tr1* is strengthened (Proof Obligation (2)) by the conjunction of the negation of all the particular failures. In this way we can ensure that there will be an enabled event also in the refined model (Proof Obligation (5)).

4.2.4 Orthogonal Regions Pattern

Furthermore, we can also consider a pattern for adding the same behaviour to several states (orthogonal regions [20]) as a type of data and event refinement. This pattern can be used in case of architectural redundancy, i.e. when several states have incoming (entry) and outgoing (exit) transitions of similar functionality. Fig. 7 illustrates adding an orthogonal region (the lower region) to the superstate *susp*, which has new behaviour common to all the previous states (given in the higher region), applicable to all three kinds of failure. In order for the pattern to be correct, several conditions have to be fulfilled. The orthogonal region should not affect the mechanism of error detection. In Fig. 7 we show that the unnamed entry and exit transitions of the lower region connect to the named events of the upper region. It must be ensured that, when the new region is entered, at least one of the new transitions must be enabled (synchronisation condition). Moreover the exit transitions from the upper region are synchronised with equivalent transitions of the lower region, i.e., they are guarded by the lower region reaching a state that has an exit transition with which it can merge.

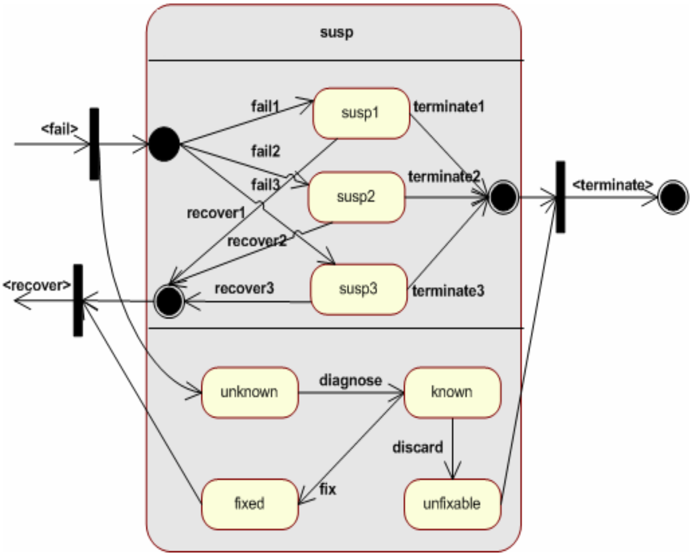


Fig. 7. Refinement pattern - superposition of an orthogonal region

Subsequently we give a machine and its refinement using the orthogonal states. The upper region of the state *susp* forms the abstract machine.

```
MACHINE         EX7
VARIABLES       state, suspState
INVARIANT       state ∈ {susp, state_ok} ∧ suspState ∈ {susp1, susp2, susp3}
INITIALISATION  state:=susp || suspState:={susp1, susp2, susp3}
EVENTS
    fail1 =     WHEN state=state_ok                THEN state:=susp || stateSusp:=susp1  END;
    recover1 =  WHEN state=susp ∧ suspState=susp1  THEN state:=state_ok                  END;
    terminate1 = WHEN state=susp ∧ suspState=susp1 THEN state:=fatalTermination          END;
    ...
END
```

In the refinement the old state *susp* is composed with the orthogonal region.

```
REFINEMENT      EX7a
REFINES         EX7
VARIABLES       state, suspState, ortState, s
```

```
INVARIANT        state ∈ {susp, state_ok, fatalTermination} ∧ suspState ∈ {susp1, susp2, susp3} ∧
                 ortState ∈ {unknown, known, fixed, unfixable} ∧ s∈TYPE ∧ I(s)
INITIALISATION   state:=susp || suspState:={susp1} || ortState:=unknown || s:∈ TYPE
EVENTS
    fail1 =      WHEN state=state_ok ∧ G_f(s)
                 THEN state:=susp || stateSusp:=susp1 || ortState:=unknown END;
    recover1 =   WHEN state=susp ∧ suspState=susp1 ∧ G_r(s) ∧ ortState=fixed
                 THEN state:=state_ok || S_r(s) END;
    terminate1 = WHEN state=susp ∧ suspState=susp1 ∧ ¬G_f(s) ∧ ortState=unfixable
                 THEN state:=fatalTermination || S_f(s) END;
    …
    diagnose =   WHEN state=susp ∧ ortState=unknown     THEN ortState:=known END;
    discard =    WHEN state=susp ∧ ortState=known       THEN orState:=unfixable END;
    fix =        WHEN state=susp ∧ ortState=known       THEN ortState:=fixed || S_f(s)END
END
```

As the events/transitions introduced in the orthogonal region are new events in the refinement, they should only concern new features to satisfy Proof Obligation (3). These new transitions should eventually hand over control to the old transitions, which is guaranteed by Proof Obligation (4). Hence, we need to generate a variant on the distance to an *exit* transition for these new transitions. In order to fulfil Proof Obligation (5) at least one of the new transitions (*diagnose* followed by *discard* or *fix*) must be enabled after entering the orthogonal region. This is caused by the fact that the recovering transitions and the terminating transitions wait to be enabled until the orthogonal region is prepared to synchronise with them. The composed events are then refinements of the old events, like for example *fail1*, *fail2*, *fail3,* in conformance with Proof Obligation (2). The orthogonal region strengthens the guards of the termination and recovery events, but at the same time it guarantees that an exit state of the orthogonal region will be reached (Proof Obligation (6)).

As the size of the system grows during the development, it is difficult to get a clear overview of the refinement process. In this paper we benefit from *progress diagrams* [16] to give an abstraction and graphical-descriptive view documenting the applied patterns in each step. The pattern types are illustrated in more detail with the progress diagrams to show the relevant development changes in a legible manner. This is of high importance especially when the system evolves into a significantly sized one.

5 Progress Diagrams

We exploit the progress diagram [16], which is in the form of a table divided into a description part and a diagram part. With this type of table we can point out the design patterns derived from the most important features and changes done in the refinement step. It provides compact information about each refinement step, thereby indicating and documenting the progress of the development. The tabular part briefly describes the relevant features or design patterns of the system in the development step. Moreover, it depicts how states and transitions are refined, as well as new variables that are added with respect to these features. Progress diagrams do not involve any mathematical notation and are, therefore, useful for communicating the development steps to non-formal methods colleagues.

Event-B classifies events as ‘convergent’ if they are new events that are expected to eventually relinquish control to an old (refined) event (i.e. it must decrease the variant). Events that are not convergent are classified as ordinary. A third classification, 'anticipating', refers to events that will be shown to be convergent in a future refinement, but we do not use such events in the examples of the patterns.

We call the transition that starts a sequence of convergent transitions an 'initiator'. To ensure feasibility of the sequence of transitions, the guard of an initiator must imply the guard of at least one of the old transitions that it could lead to. We call the transition 'refined' if it refines an existing transition according to the refinement rules (given in Section 4.1 of the paper), leaving the system in an equivalent state to the post-state of the transition being refined.

We envisage a tool which will automatically create a new refinement from the progress diagram. In order to be able to design and implement such a tool, we extend the tabular part of the progress diagram to provide the required information. As a result, we add new features in the Refined Transitions part indicating the source and target state of the refined transition, as well as the initialisation of the variables added in the current refinement step. The diagram part gives a supplementary view of the current refinement step and is, in fact, a fragment of the statechart diagram. It can be used as assistance for the developer and to support the documentation.

During the development we profit from the progress diagram, as we concentrate only on the refined part of the system. The combination of descriptive and visual approaches to show the development of the system gives a compact overview of the part that is the current scope of development. This enables us to focus on the details that we are most interested in, and provides a legible picture of the (possibly complex) system development. The visualisation helps us to better understand the refinement steps and proofs that need to be performed.

When proving the refined system, the progress diagram indicates the needed proof obligations. If new states (column "Ref. States") and variables (column "New Var.") are added, they should be initialised according to the invariant (Proof Obligation (1)). In the progress diagram the refined events are given in the column "Refined Transitions" and have a corresponding event in the column "Transitions" (Proof Obligation (2)). Also the convergent events are given in the column "Refined Transitions". However, they do not have a corresponding event in the column "Transitions". They may only assign the variables in column "New Variables" according to the invariant (Proof Obligation (3)). Furthermore, the non-divergence of the convergent transitions (Proof Obligation (4)) is indicated in the diagram part by the fact that these transitions do not form a loop. The columns "Transitions" and "Refined Transitions" also illustrate partitioning of the error detection events (Proof Obligation (6)). The progress of the refined specification always has to be ensured in line with Proof Obligation (5).

In order to illustrate the idea of progress diagrams in combination with refinement patterns, we use the abstract system (shown in Fig. 3a, Section 4.2.1) consisting of two states (*st1* and *st2*) and two transitions (*tr1* and *tr2*). We refine it to the concrete system shown in Fig. 3b, where the state (*st1*) is partitioned into substates (*st1a* and *st1b*) and the anticipating transition *trNew* is added between the new substates. The progress diagram of this sample refinement step is depicted in Fig. 8.

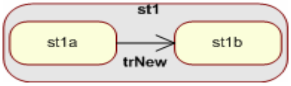
Description	States	Ref. States	Transitions	Ref. Transitions	New Var.
1st refinement step: • creating hierarchical substates (in state *st1*) – data ref. • adding new transition concerning the substates (*trNew*) – event ref.	*st1*	*st1a* *st1b*	-	*trNew (st1a, st1b)*	-
					

Fig. 8. Example of a progress diagram for the pattern 3b from Section 4

In the progress diagram in Fig. 9 we depict the event refinement by separating existing transitions. We continue refining the system shown in Fig. 3b in Section 4.2.1, by detailing its functionality and splitting the existing self-transition *tr1* into self-transitions *tr1a* and *tr1b,* according to the substates separated in the previous step. We also assume that with respect to the added transitions in the refined system we simultaneously add new variables *y* and *z*. The new variables and their initialisation are depicted in the rightmost column of the progress diagram.

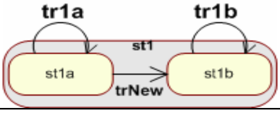
Description	States	Ref. States	Transitions	Ref. Transitions	New Var.
2nd refinement step: • separating the existing transition *tr1* into two transitions (*tr1a* and *tr1b*) concerning the substates (*st1a* and *st1b*) – event ref.	-	-	*tr1*	*tr1a (st1a, st1a),* *tr1b (st1b, st1b)*	*y::* TYPE *z::* TYPE
					

Fig. 9. Example of a progress diagram for the pattern 3c from Section 4

We also consider the flattening pattern to diminish complex hierarchical state structure created while performing consecutive refinement steps. In Fig. 10 we show the progress diagram for the flattening pattern for the diagram in Fig. 3c given in Section 4.2.2.

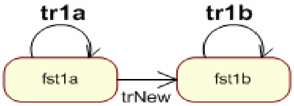
Description	States	Ref. States	Transitions	Ref. Transitions	New Var.
3rd refinement step: • removing the superstate (flattening) – data ref.	*st1 (st1a)*	*fst1a*	-	-	*v::* TYPE
	st1 (st1b)	*fst1b*			
					

Fig. 10. Example of the progress diagram for the flattening pattern

The progress diagram of the choice path pattern is depicted in Fig.11 and Fig. 12. In Fig. 11 we create a choice path on the transition *tr1*, by refining the transition *tr1* into two more specific transitions *tr1a* and *tr1b*. The guards $G_{1a}(q)$ and $G_{1b}(q)$ are created in such a way that we ensure progress of the system.

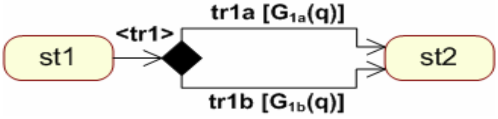
Description	States	Ref. States	Transitions	Ref. Transitions	New Var.
4th refinement step: • adding alternative paths to transitions – event ref. (detailing functionality)	-	-	*tr1*	*tr1a (st1, st2),* *tr1b (st1, st2)*	*q::* TYPE
					

Fig. 11. Example of a progress diagram for the choice paths pattern (specifying functionality)

The splitting could also be used to model more detailed failure. In Fig. 12 we split the transition *tr2undef* between the states *st1* and *susp* into two transitions, *tr2a* and *tr2undef2,* by strengthening the guard condition on the refined transition *tr2a*. The guard of the refined transition *tr1* is the negation of the refined failure transition *tr2a.*

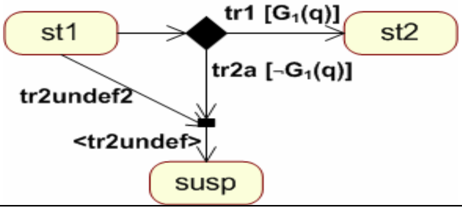
Description	States	Ref. States	Transitions	Ref. Transitions	New Var.
5th refinement step: • adding alternative paths to transitions – event ref. (error detection refinement)	-	-	*tr2undef*	*tr2undef2* *(st1, susp),* *tr2a* *(st1, susp)*	*q* :: TYPE
					

Fig. 12. Example of a progress diagram for the choice paths pattern (specifying error detection)

When adding common behaviour to the existing states (superposition of an orthogonal region), we want to express which of the new transitions are performing the functionality of the old ones. Therefore, we refine existing system (shown in the upper part of the superstate in Fig. 7) to a functionally more structured and unified one (shown in the lower part of the superstate in Fig. 7). Thereby we create a behavioural pattern, where the system before the refinement is synchronised with the system after the refinement.

In the progress diagram in Fig. 13 we depict the orthogonal region as follows. We show only the lower region without the previous higher region in the superstate. In the tabular part we compare the old superstate with the new one using a bracket notation (superstate {subA1, subA2…} {subB1, subB2…}) to indicate the hierarchy of states

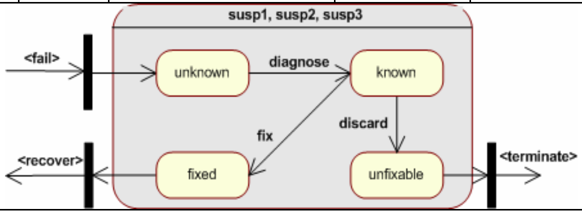
Description	States	Ref. States	Transitions	Ref. Transitions	New Var.
6th refinement step: - adding an orthogonal region to the states with common behaviour – data and event ref.	*susp* *{susp1* *susp2* *susp3}*	*susp* *{susp1, susp2, susp3}* *{unknown(<-fail),* *known,* *fixed(->recover),* *unfixable(->terminate)}*	-	*diagnose* *(unknown, known),* *fix* *(known, fixed),* *discard* *(known, unfixable)*	*r::* TYPE
					

Fig.13. Example of progress diagram for the orthogonal region pattern

in regions. We indicate the states that need synchronisations with existing incoming and outgoing transitions. Furthermore, we specify the new transitions in the orthogonal region and add some variables according to the refinement step.

6 Case Study Memento

Fig. 14 depicts the progress diagram of the *first refinement step* for the Memento system (following the abstract specification presented in Section 3), where states are partitioned into substates and transitions are added with respect to these. Partitioning the state *init* indicates that the initialisation phase is divided into a connection phase (state *conn*) and a preparation phase (state *prep*), that both need cooperation with the server. The state *susp* is treated in a similar way. Namely, the hierarchical substates *sc*, *sp*, *sr* and *sf* are created, implying that there are, in fact, various ways of handling the errors, corresponding to the states *conn*, *prep*, *ready* and *finalised*. Thereby, more elaborate information about conditions of error occurrence is added. Note that introducing hierarchical substates corresponds not only to a more detailed model in the structural sense, but also in the functional sense. The transitions (events) *tryInit*, *failInit* and *recoverInit* are refined to more detailed ones taking into account the partitioning of the initialisation phase. The self-transition *tryInit* is refined by two events, *tryConn* and *tryPrep*, which remain self-transitions for the states *conn* and *prep,* respectively. The error handling is refined by events: *failConn* and *recoverConn* for the substate *conn*, and *failPrep* and *recoverPrep* for the substate *prep*. The initiator, transition *cont* (added between the new substates *conn* and *prep*), converges to *tryPrep* and *failPrep* which are refined transitions of *tryInit* and *failInit* respectively. The initiator is guarded by the guard (*is_ok=FALSE*) from *tryInit*, thus ensuring feasibility. New variables *is_conn*, *is_prep* and *wwaited* are introduced to control the system execution flow. Note that there are separate diagram parts (not shown) for the substates *sr* and *sf*.

Description	States	Ref. States	Transitions	Ref. Transitions	New Var.
1st refinement step: • creating hierarchical substates (in states *init* and *susp*) • adding new transitions concerning the substates	*init*	*conn* *prep*	*tryInit* *(init, init)*	*tryConn (conn, conn),* *tryPrep (prep, prep)*	*is_conn*:= FALSE, *is_prep*:= FALSE, *wwaited*:= FALSE
			-	*cont (conn, prep)*	
	susp	*sc, sp,* *sr, sf*	*failInit* *(init, susp)*	*failConn (conn, sc),* *failPrep (prep, sp)*	
			recoverInit *(susp, init)*	*recoverConn (sc, conn),* *recoverPrep (sp, prep)*	

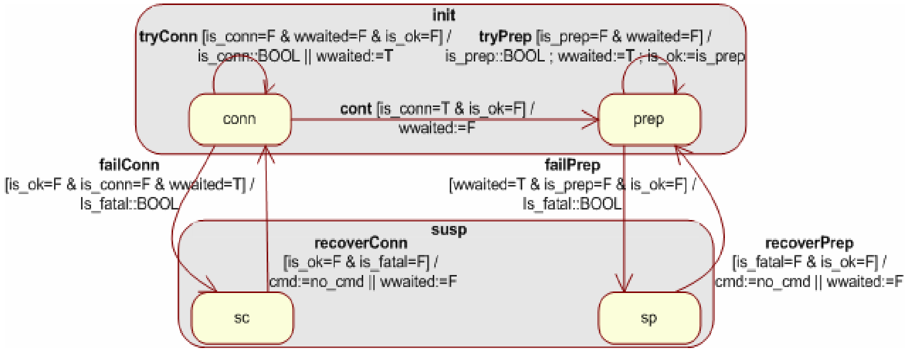

init
tryConn [is_conn=F & wwaited=F & is_ok=F] /
is_conn::BOOL || wwaited:=T
tryPrep [is_prep=F & wwaited=F] /
is_prep::BOOL ; wwaited:=T ; is_ok:=is_prep
conn
cont [is_conn=T & is_ok=F] /
wwaited:=F
prep
failConn
[is_ok=F & is_conn=F & wwaited=T] /
Is_fatal::BOOL
failPrep
[wwaited=T & is_prep=F & is_ok=F] /
Is_fatal::BOOL
susp
recoverConn
[is_ok=F & is_fatal=F] /
cmd:=no_cmd || wwaited:=F
recoverPrep
[is_fatal=F & is_ok=F] /
cmd:=no_cmd || wwaited:=F
sc
sp

Fig. 14. Progress diagram of the first refinement step of Memento

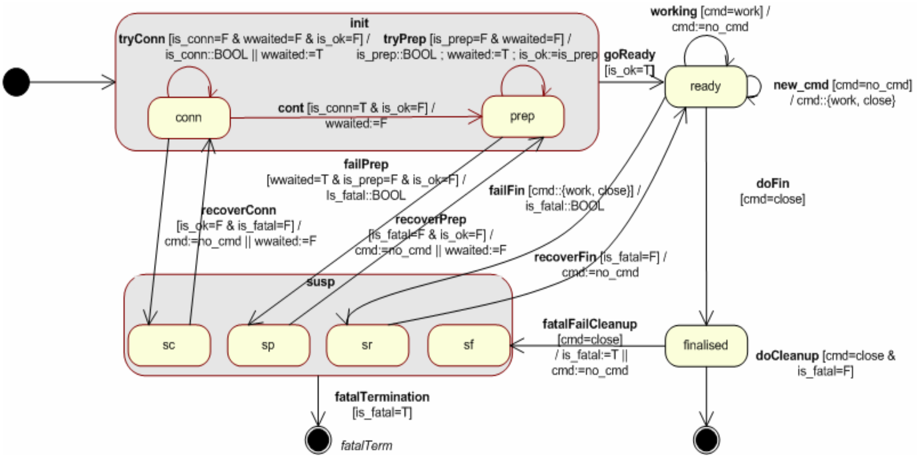


Fig. 15. Statechart diagram of the first refinement step of Memento

As the refined specification is translated to Event B for proving its correctness, the progress diagram provides an overview of the proof obligations needed for the refinement step. Since we add new states and variables, we indicate that the old transitions and initialisation need to be refined, according to Proof Obligation (1) and (2). For example in Fig. 14 events *tryConn* and *tryPrep* refine *tryInit*. Event *cont* is a convergent event that only assigns the new variable *wwaited* (Proof Obligation (3)).

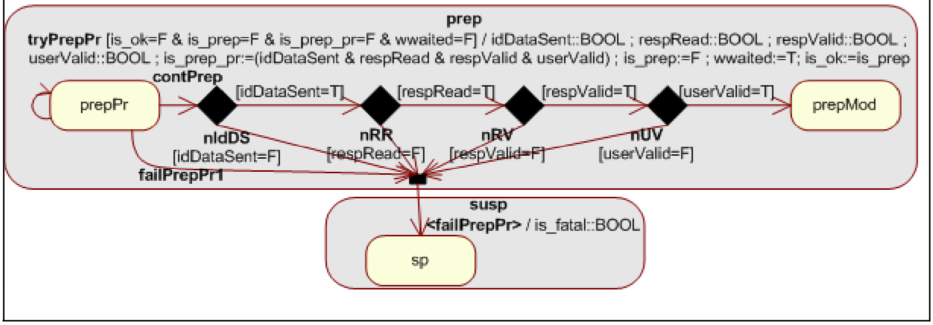
Since this event is the only newly introduced event in this refinement step and it connects two separate states *conn* and *prep,* Proof Obligation (4) is fulfilled. Furthermore, the error detection event *failinit* is partitioned into *failConn* and *failPrep* in line with Proof Obligation (6). The transitions are composed in such a way that they ensure progress in the diagram (Proof Obligation (5)).

The result of the first refinement step is shown in the statechart diagram in Fig. 15. When comparing this diagram to the one in Fig. 14, it is worth mentioning that even if the former shows the complete system, the diagram is more difficult to read with all its details. The progress diagram shows only the relevant changes in a more legible way.

The excerpt of Event-B code depicting the first refinement step of Memento system is given below.

```
REFINEMENT      Memento_Ref
REFINES         Memento
SEES            Data
VARIABLES       is_fatal, is_ok, cmd, state, wwaited, is_conn, is_prep, conn, prep, sc, sp
INVARIANT       is_fatal ∈ BOOL ∧ is_ok ∈ BOOL ∧ cmd ∈ CMD ∧ state ∈ STATE ∧
                (state=init ⇒ cmd=no_cmd) ∧
                init_state ∈ {conn, prep} ∧ susp_state ∈ {sc, sp} ∧
                (state=init ∧ init_state=prep ⇒ is_conn=TRUE) ∧
                (state=susp ∧ susp_state ∈ {sc,sp} ⇒ cmd=no_cmd) ∧
                (state=susp ∧ susp_state=sp ⇒ is_conn=TRUE) ∧
                (is_prep=TRUE ⇒ is_conn=TRUE) ∧ ...
INITIALISATION  is_fatal:=FALSE || cmd:=no_cmd || is_ok:=FALSE || state:=init ||
                is_conn:=FALSE  ||  is_prep:=FALSE  ||  wwaited:=FALSE  ||  susp_state:=sc  ||
                init_state:=conn
EVENTS
    tryConn (refines tryInit) =
                WHEN state=init ∧ init_state=conn ∧ is_ok=FALSE ∧ is_conn=FALSE ∧ wwaited=FALSE
                THEN is_conn :∈ BOOL || wwaited:=TRUE || is_ok :∈ BOOL END;
    failConn (refines failInit) =
                WHEN state=init ∧ init_state=conn ∧ is_ok=FALSE ∧ wwaited=TRUE
                THEN state:=susp || susp_state:=sc || is_fatal :∈ BOOL END;
    recoverConn (refines recoverInit) =
                WHEN state=susp ∧ susp_state=sc ∧ is_ok=FALSE ∧ is_fatal=FALSE
                THEN state:=init || init_state:=conn || cmd:=no_cmd || wwaited:=FALSE END;
    tryPrep (refines tryInit) =
                WHEN state=init ∧ init_state=prep ∧ is_ok=FALSE ∧ is_prep=FALSE ∧ wwaited=FALSE
                THEN is_prep :∈ BOOL; wwaited:=TRUE; is_ok:=is_prep END;
    failPrep (refines failInit) =
                WHEN state=init ∧ init_state=prep ∧ wwaited=FALSE ∧ is_prep=FALSE ∧ is_ok=FALSE
                THEN state:=susp || susp_state:=sp || is_fatal :∈ BOOL END;
    recoverPrep (refines recoverInit) =
                WHEN state=susp ∧ susp_state=sp ∧ is_ok=FALSE ∧ is_fatal=FALSE
                THEN state:=init || init_state:=prep || cmd:=no_cmd || wwaited:=FALSE END;
    goReady =
                WHEN state=init ∧ is_ok=TRUE ∧ is_conn=TRUE ∧ is_prep=TRUE ∧ init_state=prep
                THEN state:=ready END;
    ...
END
```

In *the second refinement step* new hierarchical substates are added in the state *prep* along with new transitions that make use of them. These hierarchical substates indicate that the preparation phase is actually composed of two phases (program as well as module preparation). This step is similar to the one above and is not further described here.

Description	States	Ref. States	Transitions	Ref. Trans.	New Var.
3rd step – adding alternative paths to transitions	-	-	*failPrepPr (prepPr, sp)*	*failPrepPr1 (prepPr, sp),* *nIdDS (prepPr, sp),* *nRR (prepPr, sp),* *nRV (prepPr, sp),* *nUV (prepPr, sp)*	*idDataSent*:=FALSE, *respRead*:=FALSE, *respValid*:=FALSE, *userValid*:=FALSE
					

prep
tryPrepPr [is_ok=F & is_prep=F & is_prep_pr=F & wwaited=F] / idDataSent::BOOL ; respRead::BOOL ; respValid::BOOL ; userValid::BOOL ; is_prep_pr:=(idDataSent & respRead & respValid & userValid) ; is_prep:=F ; wwaited:=T; is_ok:=is_prep
contPrep
prepPr
[idDataSent=T]
[respRead=T]
[respValid=T]
[userValid=T]
prepMod
nIdDS
[idDataSent=F]
nRR
[respRead=F]
nRV
[respValid=F]
nUV
[userValid=F]
failPrepPr1
susp
<failPrepPr> / is_fatal::BOOL
sp

Fig. 16. Third refinement step

The third refinement step (Fig. 16) strengthens the guards of the transitions/events (according to the pattern in Fig. 6) and shows a more detailed failure management. New variables, concerning communication with the server, are introduced to express the details of the program preparation phase. These variables represent sending the identification data (*idDataSent*), reading the response (*respRead*), and checking whether the values for response and user are valid (*respValid* and *userValid*). Furthermore, new failure transitions *nIdDS*, *nRR*, *nRV* and *nUV* corresponding to these variables refine the old general failure transition.

Here, the progress diagram also gives an intuitive representation of the proof obligations, now concerning strengthening the guards of the old events (Proof Obligation (2)). This is indicated by the transitions between the *choice point* symbols in the diagram part of the progress diagram. Moreover, the outgoing transitions of these symbols illustrate intuitively that the relative deadlock freeness (Proof Obligation (5)) is preserved. Again the partitioning of the error detection event *failPrepPr* in the columns "Transitions" and "Refined Transitions" visualises Proof Obligation (6).

Below we show the partial Event-B code for the third refinement step of our case study.

```
REFINEMENT     Memento_Ref2
REFINES        Memento_Ref1
SEES           Data
VARIABLES      is_fatal, is_ok, cmd, state, wwaited, is_conn, is_prep, conn, prep, sc, sp, sr,
               prepPr, prepMod, prep_pr, prep_mod, is_prep_pr, idDataSent, respRead, respValid,
               userValid
INVARIANT      is_fatal ∈ BOOL ∧ is_ok ∈ BOOL ∧ cmd ∈ CMD ∧ state ∈ STATE ∧
               (state=init ⇒ cmd=no_cmd) ∧
               init_state ∈ {conn, prep} ∧ susp_state ∈ {sc, sp} ∧
               (state=init ∧ init_state=prep ⇒ is_conn=TRUE) ∧
               (state=susp ∧ susp_state :∈ {sc, sp, sr} ⇒ cmd=no_cmd) ∧
               (state=susp ∧ susp_state=sp ⇒ is_conn=TRUE) ∧
               (is_prep=TRUE ⇒ is_conn=TRUE) ∧
               idDataSent ∈ BOOL ∧ respRead ∈ BOOL ∧
```

```
                    respValid ∈ BOOL ∧ userValid ∈ BOOL ∧ prep_state:=prep_pr ∧
                    (state=init ∧ init_state=prep ∧ prep_state=prep_mod ⇒
                      idDataSent=TRUE ∧ respRead=TRUE ∧ respValid=TRUE ∧ userValid=TRUE) ∧
                    (state=susp ∧ susp_state=sr ⇒
                      idDataSent=TRUE ∧ respRead=TRUE ∧ respValid=TRUE ∧ userValid=TRUE) ∧
                     (is_prep_pr=TRUE ⇒
                      idDataSent=TRUE ∧ respRead=TRUE ∧ respValid=TRUE ∧ userValid=TRUE) ...
INITIALISATION      is_fatal:=FALSE || cmd:=no_cmd || is_ok:=FALSE || state:=init ||
                    is_conn:=FALSE || is_prep:=FALSE || wwaited:=FALSE || susp_state:=sc ||
                    init_state:=conn || prep_state:=prep_pr || is_prep_pr:=FALSE || idDataSent:=FALSE
                    || respRead:=FALSE || respValid:=FALSE || userValid:=FALSE
EVENTS
    tryPrepPr (refines tryPrep) =
                    WHEN state=init ∧ init_state=prep ∧ is_ok=FALSE ∧
                    is_prep=FALSE ∧ wwaited=FALSE ∧ is_prep_pr=FALSE ∧ prep_state=prep_pr
                    THEN idDataSent :∈ BOOL; respRead :∈ BOOL; respValid :∈ BOOL; userValid :∈ BOOL;
                    IF (idDataSent=TRUE ∧ respRead=TRUE ∧ respValid=TRUE ∧ userValid=TRUE)
                    THEN is_prep_pr:=TRUE
                    ELSE is_prep_pr:=FALSE END;
                    is_prep:=FALSE; wwaited:=TRUE; is_ok:=is_prep END;
    failPrepPr (refines failPrep) =
                    WHEN state=init ∧ init_state=prep ∧ wwaited=TRUE ∧
                    is_prep=FALSE ∧ is_ok=FALSE ∧ is_prep_pr=FALSE ∧ prep_state=prep_pr
                    THEN state:=susp || susp_state:=sp || is_fatal :∈ BOOL END;
    recoverPrepPr (refines recoverPrep) =
                    WHEN state=susp ∧ susp_state=sp ∧
                    is_ok=FALSE ∧ is_fatal=FALSE ∧ is_prep_pr=FALSE
                    THEN state:=init || init_state:=prep || prep_state:=prep_pr ||
                    cmd:=no_cmd || wwaited:=FALSE || is_ok:=FALSE ||
                    idDataSent:=FALSE || respRead:=FALSE || respValid:=FALSE || userValid:=FALSE END;
    nIdDS (refines failPrepPr) =
                    WHEN state=init ∧ init_state=prep ∧ prep_state=prep_pr ∧ is_prep=FALSE
                    ∧ is_prep_pr=FALSE ∧ is_ok=FALSE ∧ wwaited=TRUE ∧ idDataSent=FALSE
                    THEN state=susp || susp_state=sp || is_fatal :∈ BOOL END;
    ...
END
```

The specification presented on the listing above, although more concrete, is not yet implementable. Nonetheless, it provides very good understanding of what actions should be taken in order to ensure stability and fault tolerance.

7 Related Work

Design patterns in UML and B have been studied previously. Chan et al. [6] work on identifying patterns at the specification level, while we are interested in refinement patterns. The refinement approach on design patterns was presented by Ilič et al. [9]. They focused on using design patterns for integrating requirements into the system models via model transformation. This was done with strong support of the Model Driven Architecture methodology, which we do not consider in this paper. Instead we provide an overview of the development from the patterns.

Refinement patterns in the Event-B method were also investigated by Alexei Iliasov [8], but with respect to the rapid development of dependable systems. The author explores a method for mechanised transformation of formal models and merges theory with practice by implementing the tool that supports the formerly created patterns language. Since automation is less error-prone than manual coding, applying patterns with the use of the created tool is profitable for the dependable systems development. We also rely on patterns in order to prevent introducing the errors into the system development, making the construction of the system process more dependable. However, we are not concerned about creating a language for the patterns. Instead we benefit from the progress diagrams through the readability and the intuition they provide.

An approach relating formal and informal development is used in the research of Claudia Pons [17], where the formally defined refinement methodology is submerged into UML-based development. The method is described by the term "formal-to-informal", treated as a complement of the "informal-to-formal" approach standing for translating the graphical notation into formal language. The presented methodology is based on the Object-Z formal language and UML structures. It presents an object decomposition pattern and a non-atomic operation refinement via examples of classes. In our research we focus on statemachines instead of classes and combine the formal and informal approaches, which in our case are complementary to each other. Moreover, we use Event-B in order to have a tool support for our development.

Defining standards in semantics for different level of abstractions in system level design has been studied by Junyu Peng, Samar Abdi, and Daniel Gajski in [15]. The authors' approach to system development relies on the automation of the refinement process via tool support. The main focus in their research is to improve robustness and usefulness of the system design, even if the methodology aims at the architecture of the system in general. Their effort is towards rapid prototyping and evaluation of several design points, while our approach is of a formal nature, focusing on the correctness of the system created in a stepwise manner.

8 Conclusions and Future Work

This paper presents a new approach to documentation of the stepwise refinement of a system. Since the specification for each step becomes more and more complex and a clear overview of the development is lacking, we focus our approach on illustrating the development steps. This kind of documentation is not only helpful for the developers, but also for those that later will try to reuse the exploited features. The documentation is also useful for communicating the development to stakeholders outside of the development team. Thus, a clear and compact form of progress diagrams is appropriate both for industry developers and researchers.

Formal methods and verification techniques are used in the general design of the Memento application to ensure that the development is correct. Our approach uses the B Method as a formal framework and allows us to address modelling at different levels of abstraction. The progress diagrams give an overview of the refinement steps and the needed proofs. Furthermore, the use of progress diagrams during the incremental construction of large software systems helps to manage their complexity and provides legible and accessible documentation.

In future work we will further explore the link between the progress diagrams and patterns. We will investigate how suitable the progress diagrams are for identifying and differentiating patterns used in the refinement steps. Although progress diagrams already appear to be a viable graphical view of the system development, further experimentation on other case studies is envisaged leading to possible enhancements of the progress diagrams.

We have considered the possibility of developing tool support for drawing progress diagrams and automatically generating a new refinement from the progress diagram and the previous level model. The most likely route for tool support is to extend the UML-B tool [19], which is already an extension of the Event-B tool set. The complete tool set is based on the Eclipse development environment as a ‘rich client platform’. UML-B provides a graphical drawing tool for drawing state machine diagrams (as well as class diagrams) and converting them automatically into Event-B where the Event-B static checker and prover automatically perform verification on the model. UML-B uses the ‘Eclipse Modelling Framework’ (EMF) to generate a repository for UML-B models from a meta-model diagram. The UML-B meta-model defines the abstract syntax of the UML-B language. The drawing tool is based on the ‘Graphical Modelling Framework’ (GMF). We envisage a new meta-model for progress diagrams, that extends the UML-B meta-model to define the refinement relations, that we have described in this paper, mapping individual elements of an existing UML-B model to newly created UML-B elements. The diagrammatic part of the progress diagram editor would consist of a reduced UML-B Statemachine diagram. The tabular part of the progress diagram is an elegant view of the refinement properties but it is not the most suitable interface for editing them considering that they are based on the existing UML-B meta-classes. Therefore, a new editor interface (diagrammatic, tree structured or tabular) will be developed for defining the refinement properties as extensions to the referenced existing model elements. The tabular part of the progress diagram will be automatically generated as a read-only view of the refinement properties. A builder will be provided to generate the new refined UML-B model based on the progress diagram refinement model. Since the generated model is a UML-B model, the existing tools will automatically generate an equivalent Event-B model as soon as it is created.

Acknowledgements. We would like to thank Dr Linas Laibinis and Dr Dubravka Ilič for the fruitful discussions on the use of the tools supporting the research.

References

1. Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Arlow, J., Neustadt, I.: Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML. Addison-Wesley, Reading (2004)
3. Back, R.J.R., Kurki-Suonio, R.: Decentralization of process nets with centralized control. In: Proc. of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, pp. 131–142 (1983)
4. Back, R.J.R., Sere, K.: From modular systems to action systems. Software - Concepts and Tools 17, 26–39 (1996)
5. Booch, G., Jacobson, I., Rumbaugh, J.: The Unified Modeling Language - a Reference Manual. Addison-Wesley, Reading (1998)
6. Chan, E., Robinson, K., Welch, B.: Patterns for B: Bridging Formal and Informal Development. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 125–139. Springer, Heidelberg (2006)

7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series, Reading (1995)
8. Iliasov, A.: Refinement Patterns for Rapid Development of Dependable Systems. Proc of Engineering Fault Tolerant Systems Workshop (at ESEC/FSE, Dubrovnik, Croatia), ACM Digital Library, September 4 (2007)
9. Ilič, D., Troubitsyna, E.: A Formal Model Driven Approach to Requirements Engineering. TUCS Technical Report No 667, Åbo Akademi University, Finland (February 2005)
10. Katz, S.M.: A superimposition control construct for distributed systems. ACM Transactions on Programming Languages and Systems 15(2), 337–356 (1993)
11. Metayer, C., Abrial, J.R., Voisin, L.: Event-B Language, RODIN Deliverable 3.2 (D7) (May 2005), http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf
12. Object Management Group. Unified Modelling Language Specification - Complete UML 1.4 specification (September 2001)
13. Object Management Group Systems Engineering Domain Special Interest Group (SE DSIG). S. A. Friedenthal and R. Burkhart. Extending UML^{TM} from Software to Systems (accessed May 2007)
14. Olszewski, M., Pląska, M.: Memento system (2006), http://memento.unforgiven.pl
15. Peng, J., Abdi, S., Gajski, D.: Automatic Model Refinement for Fast Architecture Exploration. In: Proc. of the 15th International Conference on VLSI Design (VLSID 2002), p. 332. IEEE Computer Society, Los Alamitos (2002)
16. Pląska, M., Waldén, M., Snook, C.: Documenting the Progress of the System Development. In: Bulter, M., Jones, C., Romanovsky, A., Troubitsyna, E. (eds.) Methods, Models and Tools for Fault Tolerance. LNCS, vol. 5454, pp. 251–274. Springer, Heidelberg (2009)
17. Pons, C.: Heuristics on the Definition of UML Refinement Patterns. In: Wiedermann, J., Tel, G., Pokorný, J., Bieliková, M., Štuller, J. (eds.) SOFSEM 2006. LNCS, vol. 3831, pp. 461–470. Springer, Heidelberg (2006)
18. Snook, C., Butler, M.: UML-B: Formal modelling and design aided by UML. ACM Transactions on Software Engineering and Methodology 15(1), 92–122 (2006)
19. Snook, C., Butler, M.: UML-B and Event-B: an integration of languages and tools. In: Proc. of The IASTED International Conference on Software Engineering – SE 2008, Innsbruck, Austria (February 2008)
20. Snook, C., Waldén, M.: Refinement of Statemachines Using Event B Semantics. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 171–185. Springer, Heidelberg (2006)
21. Troubitsyna, E.: Stepwise Development of Dependable Systems. Turku Centre for Computer Science, TUCS, Ph.D. thesis No. 29 (June 2000)
22. Waldén, M., Sere, K.: Reasoning About Action Systems Using the B-Method. Formal Methods in Systems Design 13(5-35) (1998)

Fault Tolerance Requirements Analysis Using Deviations in the CORRECT Development Process

Andrey Berlizev[1,2] and Nicolas Guelfi[1]

[1] Laboratory for Advanced Software Systems, University of Luxembourg
6, rue Richard Coudenhove-Kalergi, L-1359 Luxembourg-Kirchberg, Luxembourg
[2] Software Modeling and Verification Group, Centre Universitaire D'Informatique
Université de Genève, Route de Drize, 7 CH-1227 Carouge, Switzerland
{Andrey.Berlizev,Nicolas.Guelfi}@uni.lu

Abstract. Current requirements analysis methods focus on the functional properties of fault free systems. It is known that, regardless of the type of software system, many faults are made during engineering and that these faults may conduct system errors and then system failures. We believe that faulty engineering activities, as well as correct activities, should be given precedence during software development. In this paper we present CORA, which is the analysis phase for the CORRECT methodology. CORA introduces semi-formal models based on UML and OCL that allow for the specification of normal system behaviors, as well as abnormal behaviors, together with their associated recovery strategy. CORA proposes to specify fault-tolerant systems using a domain model as a custom UML class diagram and an activity model as a custom UML activity diagram. The deviation and recovery strategies are expressed explicitly in a specific section of the CORA Activity Diagram. This paper introduces CORA conceptually and it explicitly defines the syntax and semantics of the proposed analysis models. We also use a running example to illustrate our approach.

Keywords: integrated approaches, fault-tolerant systems development, deviations, Semi-formal methodology, MDE, UML.

1 Introduction

Contemporary software development is a difficult task because building software is highly complex (due to: large size, distribution, cross platform, etc.). As a result, systems are prone to developer mistakes and are highly dependant on the hardware they use. In critical (safety or business) applications (such as medical systems, banking systems, etc.), the quality requirement that needs to be possessed by users is further increased and software correctness is of utmost importance. Although there are development methods [5] for systems that satisfy these requirements, they are often costly and involve special skills. Applying such technical methods, even for developing general and not highly critical software systems, is rarely feasible. We believe that by simplifying how dependability techniques are produced within the development process (particularly for fault tolerance and abnormal behavior) will help to enhance the dependability of software.

M. Butler et al. (Eds.): Fault Tolerance, LNCS 5454, pp. 275–296, 2009.
© Springer-Verlag Berlin Heidelberg 2009

A widely accepted definition of *dependability* for computing systems was introduced by Jean-Claude Laprie in 1985 as, “the trustworthiness of the system by which reliance can be justifiably placed on the services the system delivers” [10]. Developer mistakes or hardware defects are known as *faults*. If they are executed and *manifest*, faults may lead to an *error*: an improper internal state of the system. If an error is not *recovered* to normal state and reaches the border of the system, it is a *failure* of the system for the environment (behavior being different from that specified). *Fault-tolerance* (FT) is the ability to comply with specifications even in the presence of faults. Usually FT is introduced during the design phase [7], however, we agree with Beder et al [2] that the earlier FT and dependability are introduced during development, the better the outcome. Dealing with abnormal behavior [14] begins at the requirement elicitation phase, where use case based requirements are produced using concepts from the exception-handling world (exceptions, handlers, etc.). We also utilize some ideas, where degraded use case outcomes are introduced [11][12].

The development process of Complex fault tOlerant distRibuted systems: from aRchitEctural desCription to Java implemenTation (CORRECT), with its focus on FT aspects, has five defined layers: requirements, CORrect Analysis (CORA) (presented in this paper), design, implementation, and verification [3]. According to the standard ISO/IEC 12207 [5], CORRECT process is only within development and does not specify any other system lifecycle processes. The aim of our work is to precisely define the analysis phase of the CORRECT process. Our proposal for CORA models extends scientiFic engIneering of Distributed Java applIcations (FIDJI) analysis models [4, 5] and is composed of:

- Domain Model precisely defines concepts and signals manipulated by use case and operations.
- Activity Model extends and refines the use cases described at the requirements elicitation time. Its purpose is to express the system’s behavior in terms of sequence of operations, signals and deviations and recovery. CORA Activity Diagrams are used to precisely specify each use case with all deviations.
- Operational Model specifies in detail each operation: informal descriptions, parameters, return values and pre/post-conditions.

FIDJI does not consider FT as a primary concept. By integrating deviations and FT principles coherently with the FIDJI models, our paper proposes a solution for improving the FIDJI analysis models in order to cope with FT. FIDJI models were chosen as source models because we plan to extend CORA towards a product line perspective (developing dependable reusable components) [15].

In order to do this, Section 2 describes CORA by defining the terminology used and introduces a running example and its requirements. Section 2.1 introduces terminology used, followed by one running example in Section 2.2. Section 2.3 presents the notions of deviation and recovery. Section 2.4 illustrates these notions for the running example. Then, Section 2.5 presents CORA Domain and Activity Models, specified in the running example. Section 2.6 gives a description of the syntax and semantics of CORA. Section 3 refers to related work. Section 4 provides conclusion and information about future work.

2 CORA

2.1 Terminology

As we introduce our terminology we take into account that some of our terms preexist in different contexts and their semantics is overloaded. Use cases were introduced in 1997 [4] and have been modified through many works since then; we will define a specific type. Concepts and taxonomy of dependable computing [1] provides basic terms, which are quite general, and in this section it is shown how they are applied for software systems for the models used at the analysis level. The concepts of dependability and FT at the analysis phase are closer to those of UML terminology, where "actor" rather than "user" is utilized, etc. Also, the interface of the system is abstracted and only allows message exchange with the environment.

Goals, Customer, Stakeholder, Failure of Stakeholder's Goal

Definitions

An *information system* is a set of interacting entities, where the entity being built is called *system* and other entities are *actors.*

A *system* is constructed to satisfy the *stakeholder's* needs. A *customer* is a stakeholder who decides which of these needs must be realized by the system. The specific needs determine the *system requirements. System specification* is the captured requirements, which are *complete*, *consistent* and *authoritative*. An *analyst* creates *system specification* from customer requirements using a methodology, like CORA, for example.

Failure of information system (failure of stakeholder's goal) is when the state of the information system breaks a goal's constraint and the goal is unable to be reached. Fault-tolerant activity at the level of the information system is achieved by providing additional specification of abnormal system behavior, so that if we have normal behavior, information system's state changes accordingly to the goal. Any deviations from this desired behavior will lead to the goal's breakage. An attempt to avoid this is a fault-tolerant activity at the level of the information system. A *failure of information system* occurs when a system provides behavior different from the behavior predicated by the system specification.

Justification

We do not give the precise border between requirement elicitation and analysis. However, we expect that the first phase (requirement elicitation) is to establish all of the customer's needs, even though they are neither precise nor complete and may be contradictory. The requirements used in CORA, as a source for the specification, are captured in the form of use cases. A method of requirement analysis [11][12] shows step-by-step how to produce requirement in the form of use case with respect to reliability. We have requirements where all concepts, actors and use cases are identified, as well as the initial Use Case Model and the Domain Model, which will be modified during analysis. During CORA Analysis, precise definition of requirements becomes a system specification expressed using CORA models, so that any ambiguities, incompleteness and inconsistencies have to be excluded. Because of this, the CORA system model is the sole basis for judging whether the system behavior is correct or not.

In our example, if a customer (bank) wants to borrow more money than allowed, it is a failure of the customer's goal but not a failure of the system since the system specification requests this refusal. It would be a failure of the system if the system should allow borrowing beyond the maximum amount.

Normal, Abnormal and Failure Behavior

It is not easy to define abnormal behavior for the specification, since everything specified somehow becomes "normal." Thus, every property of the specification is normal behavior unless we explicitly specify that it is abnormal. In our method we distinguish normal and abnormal behavior based on the following criteria: if an actor provides its service we call the message interaction, *normal behavior of the actor*, and if an actor cannot provide the service, then the part of the message interaction which *deviates* from normal behavior is called *abnormal behavior of the actor.* The behavior of the system for providing a service for some actor(s) assumes that every actor involved in this service provision has normal behavior: this is called *normal behavior of the system.* Similarly, every additional effort of the system aimed at providing service due to abnormal behavior of actor(s) is called *abnormal behavior of the system.* Both normal and abnormal behavior should be clearly defined in the system specification. If behavior of the system *deviates* from the specified one, it is called *failure behavior of the system.* The specification may be enhanced with *deviation specification*, where behavior of the system or environment is specified for failures of the system or environment itself. When the system needs to provide specified abnormal behavior, we should clearly distinguish *acceptable deviations.* We can do this by knowing what happens in the information system and its state (relying on customer's knowledge and feedback) so that when developing the system we can define how it should behave with an actor's abnormal behavior. *Unacceptable deviations* occur when an *error* (abnormal situation) is detected but not under control (we do not know why the error happened). To be consistent with *fault tolerance* definitions, deviations are errors in the information system, together with the defined recovery for the information system.

Justification

One of the possible abnormal behaviors might be to send an error message, "service cannot be provided," to the calling actor as a response to another actor's anticipated failure (e.g. sensor's battery is off). In this case, the system state is known but the user's goal is not reached (failure of the user's goal). The system behavior satisfies the specification stating that in the case of another actor's failure, the first actor must receive a message about the occurrence and, consequently, it is not a failure of the system. On the contrary, there can be an error situation in the information system that may lead to the information system's failure when the state of the information system is unknown. The customer can decide if this situation is an acceptable deviation or an unacceptable deviation (failure).

Operation and Use Case

Definition

Operation is a complete specification of the system behavior as a reaction to receiving a message from the environment. The specification includes that system state changes and signals are sent to the environment.

Use case is a group of operations belonging to one goal (one operation can be used by several use cases). Use cases can be nested (using <<include>> and <<extension>> relations between them) and may have generalization relations.

2.2 Running Example

As an academic running example, we will consider a software system that allows several banks to borrow money. The task of the system is to determine if a borrower is authorized to receive funds (up to a limit), to request a correspondent bank make a transfer and to ensure that these things happen successfully. The money transfer is conducted outside the system itself. In the proposed models we define several abnormal situations in the environment that the system needs to deal with by way of a specification. We precisely define what should happen in each situation.

Figure 1 a) shows a use case diagram with two actors, `Bank` and `CorrespondentBank`, together with three use cases; and b) is a simple Domain Model having only one class (signals are not shown). We use only `Login` and `Borrow` use cases, which are enough to illustrate all aspects of the method. We omit an Operational Model, which is merely the textual description in informal language at the requirement level.

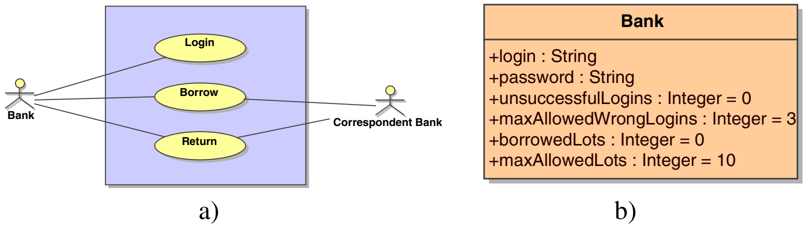

Login
Borrow
Return
Bank
Correspondent Bank
a)
Bank
+login : String
+password : String
+unsuccessfulLogins : Integer = 0
+maxAllowedWrongLogins : Integer = 3
+borrowedLots : Integer = 0
+maxAllowedLots : Integer = 10
b)

Fig. 1. Use case diagram and domain model for the running example

Figure 2 presents the `Login` use case, where the bank providing the id and password is authorized by the system. If the number of unsuccessful logins exceeds the predefined amount, the account is blocked.

Figure 3 shows the use case for the scenario used by the borrower. In this scenario, system sends a request to `CorrespondentBank` to send money to the borrower and receives a confirmation from `CorrespondentBank` that the money was sent; system will, in turn, send the confirmation to the borrower.

The use cases are described in a semi-formal manner since every step has Object Constraint Language (OCL) postcondition expression. The postconditions written inline with textual description are typically incomplete and partial, but usage of them helps to define operation names and parameters. Abnormal behavior will be introduced later in the form of deviations specification. `Borrow` refers to the system in the Borrow use case description when used in the postconditions and `Borrower` is the actor, `Bank`.

Name	Login
ID	UC1 Login
Description	Bank provides its id and login to the system for authorization.
Primary actors	Bank
Trigger event	Login is requested. Post: Login^login(id: String; password: String)
Preconditions	None.
Postconditions	Bank is logged in or WrongIdOrPassword is sent.
Main success scenario: 1. Bank call operation login with id and password. Post: Login^login(id: String, password: String) 2. System checks the login information and, if correct, it sends confirmation SuccessfullyLoggedIn, or otherwise WrongIdOrPassword. If the number of unsuccessful logins exceeds the allowable amount maxAllowedWrongLogins, the account is blocked.	

Fig. 2. Login use case (normal behavior)

Name	Borrow
ID	UC2 Borrow
Description	Bank makes a request from the system to borrow money; the system requests Correspondent Bank to make the transfer.
Primary actors	Borrower: Bank
Secondary actors	CorrespondentBank
Trigger event	Borrowing is requested. Post: Borrow^borrow(amount: Integer)
Preconditions	An actor is logged into the system.
Postconditions	Money is sent and received. Post: Borrow^amountSent() and Borrow^confirm() and Borrower^confirm()
Main success scenario: 1. Borrower requests an amount to be borrowed as Integer lots. Post: Borrow^borrow (amount: Integer) 2. System checks the limit and if it is acceptable it sends a request to CorrespondentBank to make a transfer. CorrespondentBank^SendAmount(borrower.accountNumber: String; amount: Integer) 3. CorrespondentBank confirms sending. Borrow^amountSent() 4. System confirms with the borrower. Post: Borrower^AmountSent() 5. Borrower confirms receiving. Post: Borrow^confirm()	

Fig. 3. Borrow use case (normal behavior)

2.3 Deviation and Recovery

FT aims at making sure that a system continues to behave as specified, even in the presence of faults. FT at the analysis level aims at providing a specification not only describing the normal system behavior, but also behavior that is acceptable to the customer even though it is undesirable. A specification describes what a system should do: its normal behavior. We want to change specifications in such a way that they describe abnormal behavior, as well, and include predefinition about error messages, etc., ultimately simplify classical FT at the design level. In classical FT, the mechanisms used are implicit and hidden from the customer. This means that any deviation from the specified behavior is a failure of the system and is not accepted by the customer. In FT specification we can say that some deviations remain acceptable by the customer with the proper recovery, or that within normal service it may be acceptable to have some degraded service.

To support FT at the analysis level we introduce the notion of *deviation*. Deviation is the expression of the difference between two elements that should be equal. For example, the behavior of the real system should be equal to the specified behavior of the system, if not, there is a deviation in the behavior. Another example: the invariants of the real system must be true. If an invariant expression is not true, it is deviation of the invariant. For any deviation defined, we also want to define acceptable recovery and thereby extend the specification of normal behavior with deviations and recovery accordingly. We use the stereotype <<deviation>> followed by a detection statement, along with some additional stereotypes with their statement's description for the full description of FT requirements:

- <<recovery>> - this required section describes what should be done to tolerate detected deviation.
- <<impact>> - this is an additional optional clause that allows defining of use cases, classes or other elements which will be impacted by this deviation.
- <<continue>> - this optional section defines what should happen after the recovery.

Within the use case we propose to add deviations as additional blocks or lines. Figure 4 exemplifies how this is done in our `Borrow` use case, but because of space limitations it leaves only those parts that were changed (compare with Figure 3).

Comparing our deviation notation to the classical Cockburn template [4], we should note that any backup action is a deviation with recovery. However, in written use case description during fault-tolerant analysis, new deviations may be introduced. Sometimes it is more convenient if they are put near the element deviated (see Figure 4). On occasion, exception clause is used to describe abnormal behavior, which is also deviation and can be expressed in the form of deviation notation for clarity.

2.4 Running Example with Fault Tolerance Requirements

During FT analysis we should find possible deviations to answer the question of "what can go wrong?" with every step and element of the normal specification. A decision of what faults to consider during the development of the system will need to be made – not every found deviation should be considered since it is not possible to offer a solution at the analysis level – they can be determined by the system type and the desired dependability (quality) level. For instance, we do not consider communication is lost with the user unless our specification abstracts communication at the

<table>
<tr><td>Trigger event</td><td>Borrowing is requested.
Post: Borrow^borrow(amount: Integer)
<<deviation>> amount is not type of Integer
<<recovery>> Borrower^WrongParameter</td></tr>
<tr><td>Preconditions</td><td>An actor is logged into the system.</td></tr>
<tr><td>Postconditions</td><td>The money is sent and received.
Post: Borrow^amountSent() and Borrow^confirm() and Borrower^confirm()</td></tr>
<tr><td colspan="2">…
3. The CorrespondentBank confirms sending.
Borrow^amountSent()
<table>
<tr><td>&lt;&lt;deviation>>
transaction failed
<<recovery>> Borrower^ef_MoneyTransactionFailure
<<continue>> UC finishes in failure</td><td>&lt;&lt;deviation>> Borrow^claimNotReceived()
<<recovery>>
CorrespondentBank^TransactionIsLost()
<<continue>> UC ends in failure</td></tr>
</table>
…</td></tr>
</table>

Fig. 4. Use case with abnormal behavior

analysis level, otherwise communication related details appear only during design. At the analysis level we can also abstract from the details of the fault and consider omitting a message to the actor. Typically this can be detected with a timeout detected by Bank, which in this case, instead of sending confirmation sends a message (illustrated in the deviation in Figure 4, on the right). On the left, we define that an error message should be sent and that the use case is finished when CorrespondentBank is unable to send the requested amount (unknown reasons may be specified during the design phase of development).

For trigger event we add reaction on the situation when Borrower does not use proper type for request parameter (such a situation may happen in web services when all parameters are passed through textual XML file). For deviation we have informal error detection.

2.5 Running Example with CORA Domain and Use Case Models

In order to provide an informal introduction to CORA, we present the running example using CORA models. We will show the extended Domain Model and two use cases: Login and Borrow. We have chosen to omit Return use case, as it is similar to Borrow and provides no new variation to our example.

Figure 5 presents the extended Domain Model for the example. There are three logical actors: two are the borrowing Bank (logged in and not logged in) and one is the CorrespondentBank. Every logical actor class is stereotyped with <<la>> and associated to physical actor(s), as well as to the use case classes in which they can participate. For example, UnregisteredUser can only participate in Login use case (once login is successful, UnregisteredUser becomes Bank), where as Bank participates in Login and Borrow use cases.

The operation login instantiates the use case that is defined in the diagram by <<autoCreate>> stereotyped dependency. After this operation execution, the use case finishes and there are no following operations. Consequently, no information about the running use case should be sent to the actor. This is shown by the fact that all signals that may be sent to UnregisteredUser are associated only with the logical actor class and not with the use case class or actor's role class. The signals for Borrow use case differ from the signals for Login use case, in that Borrow signals have additional associations. Signal AmountSent in the Borrow use case carries the information of the use case instance shown by its association between the signal and the use case classes. Signal TransactionIsLost for the Login use case carries the information of the use case instance in the same manner as Borrow, but it also contains information of the actor's role in the use case (which may be changed to the use case instance, because there is only one CorrespondentBank participating in one use case instance).

There are two operations used in our example that are predefined in CORA: connect and cs_WrongParameter. The first one, connect, defines that the PleaseRegister signal is sent to a newly connected actor (similar to how a home page appears by default upon connection to websites). The second one, cs_WrongParameter, is invoked when a deviation in the parameters received from actor is detected by the communication system. In this case, we send the error message, WrongParameter, defined in the deviation of the Login use case description (this behavior is shown explicitly in the Borrow activity diagram).

There is an association between <<la>> Bank class and <<id>> Bank class in Domain Model. The difference between these two classes is that the <<id>> Bank class has one instance per every bank registered in the system, whereas the <<la>> Bank class has one instance per any Bank logged in (a bank can only login if registered). Thus, the association has a multiplicity of 1 at one side and 0..1 at the other, not every Bank is connected to the system at all times. Also, <<la>> instances are managed automatically by the Communication System and <<id>> instances may be changed only explicitly.

For CorrespondentBank the situation is different. It is presumed that CorrespondentBank is always accessible and connected to the system and the association multiplicity is 1 at both sides. The diagram also requires that every Borrow instantiation creates an instance of Correspondent class, so that the secondary actor automatically participates in every Borrow use case.

Only a Bank participating in the use case instance can invoke operations in the role class Borrower. The borrow operation may be invoked without passing a use case role as one of the parameters (static), but all others must be invoked only in the context of the use case role (or use case instance in our example since the diagram has only one Borrower role per use case instance). CorrespondentBank may simultaneously participate in many instances of the Borrow use case. However, the operation amountSent can be invoked only by CorrespondentBank and should be supported with the use case role reference, which it receives together with the SendAmount signal.

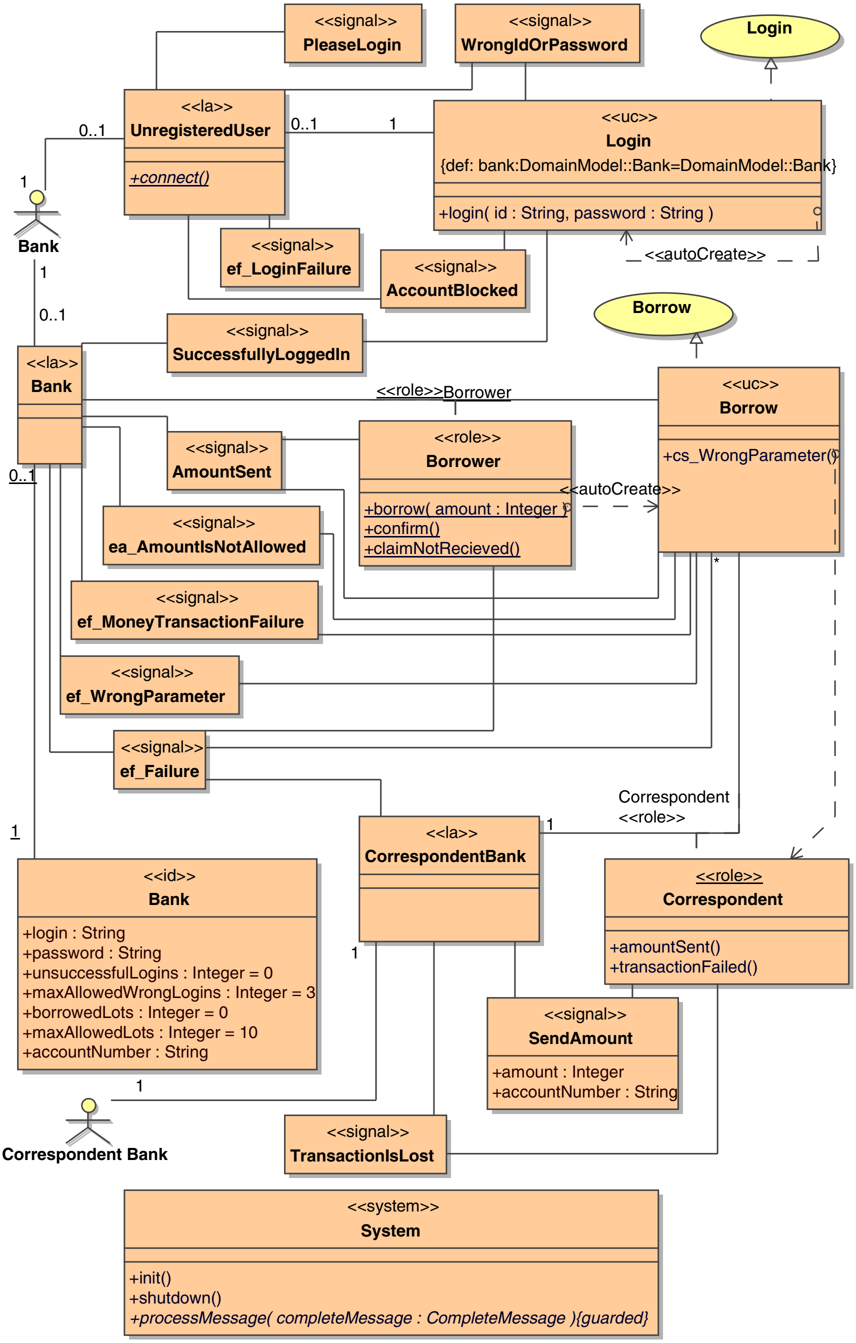


Fig. 5. CORA Domain Model for bank borrowing system

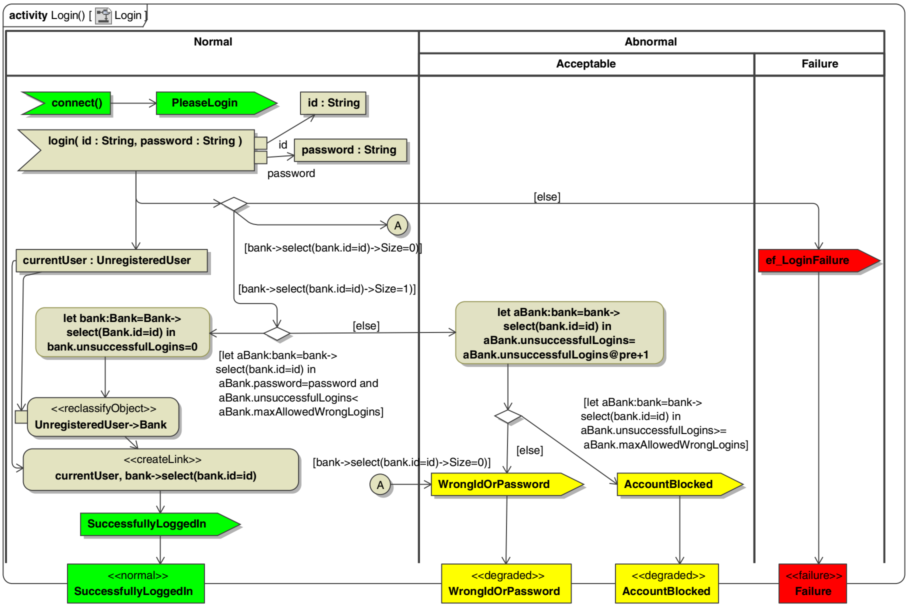

activity Login() [Login]
Normal
Abnormal
Acceptable
Failure
connect()
PleaseLogin
id : String
login(id : String, password : String)
id
password : String
password
[else]
A
[bank->select(bank.id=id)->Size=0)]
currentUser : UnregisteredUser
ef_LoginFailure
[bank->select(bank.id=id)->Size=1)]
let bank:Bank=Bank-> select(Bank.id=id) in bank.unsuccessfulLogins=0
[else]
let aBank:bank=bank-> select(bank.id=id) in aBank.unsuccessfulLogins= aBank.unsuccessfulLogins@pre+1
[let aBank:bank=bank-> select(bank.id=id) in aBank.password=password and aBank.unsuccessfulLogins< aBank.maxAllowedWrongLogins]
<<reclassifyObject>> UnregisteredUser->Bank
[let aBank:bank=bank-> select(bank.id=id) in aBank.unsuccessfulLogins>= aBank.maxAllowedWrongLogins]
[else]
<<createLink>> currentUser, bank->select(bank.id=id)
[bank->select(bank.id=id)->Size=0)]
A
WrongIdOrPassword
AccountBlocked
SuccessfullyLoggedIn
<<normal>> SuccessfullyLoggedIn
<<degraded>> WrongIdOrPassword
<<degraded>> AccountBlocked
<<failure>> Failure

Fig. 6. Activity diagram for Login use case

Thus, the extended Domain Model defines which signals may be sent in each use case context, what additional information should be sent with them (use case reference or role reference) and which actor may invoke every operation. Section 2.6.2 describes this diagram in details. We will continue by defining the use cases using CORA Activity Diagrams.

Figure 6 shows us a precise specification of the use case Login. The diagram is split into three regions using activity partitions to separate normal and abnormal behavior. There are two AcceptEventActions corresponding to the operations connect and login. Once the first operation is invoked, a new user is connected as UnregisteredUser and the message PleaseLogin is sent. Operation login is typically more complex. Because dependency is stereotyped <<autoCreate>> (see previous diagram), operation login creates a new instance of the use case as an instance of class Login. Two operation parameters (id and password) are stored as implicit control variables. These are shown by ObjectNodes connected to the action's OutputPins, and later referred to as id and password in OCL expressions. An instance of UnregisteredUser class corresponding to the actor is also stored in ObjectNode and referred to as currentUser.

If there are no instances of Bank class with id attribute matching id parameter (an incorrect id is entered), the message WrongIdOrPassword is sent to the caller. If there is an instance matching id but not the password, then the unsuccessful login counter is increased and WrongIdOrPassword is sent. And finally, if the number of unsuccessful logins exceeds the maximum number allow, the message

AccountIsBlocked is sent. The specification in our diagram counts all the unsuccessful logins, even when the account is blocked. In each of these situations, the use case is finished in degraded outcomes. If there is a login with correct id and password and the account is not blocked, the counter is reset and the physical actor becomes the logical actor, Bank (reclassification maintains links according to UML semantics so that the logical actor remains connected to the original physical actor and use case), and a link is created to the DomainModel::Bank instance (required by the extended Domain Model). A message SuccessfullyLoggedIn is sent and the Login use case finishes successfully. Finally, if a deviation of more than one instance of Bank class with the same id is found, the ef_LoginFailure signal is sent and the use case finishes with failure outcome.

The Borrow use case is shown with Figure 7. It illustrates the signal parameter passing and gives more detailed abnormal behavior. Instantiated by the borrow message from a bank, the system checks the amount requested. If it exceeds the amount allowed, the ea_AmountIsNotAllowed signal is sent and the use case finishes in the degraded outcome AmountIsNotAllowed. The guard uses keyword token to refer the actor instance in verifying the amount for the bank executing the scenario. Then, amount and accountNumber are passed on to CorrespondentBank by the signal, SendAmount, specifying the amount requested by Borrower and which account to debit. If CorrespondentBank authorizes and sends the money, confirming this in the system with the amountSent message, the stored borrowedLots is increased with the amount value. However, if the transaction fails, the transactionFailed message is sent and the system reports a failure with the ef_MoneyTransactionFailure signal being sent to Borrower; the use case finishes in failure. The borrower receiving the confirmation from the system waits while the money is being transferred. If the transfer is correct, confirm message is sent by Borrower and the use case, upon receiving this message finishes with the Success outcome. If the money does not arrive at the bank, the bank reports this with the message claimNotReceived. The recovery for this deviation is defined by sending the TransactionIsLost signal to CorrespondentBank and the use case finishes in failure. If an internal error happens that cannot be tolerated and detected by mechanisms defined at the design level for any operation performed in the use case, the ef_Failure signal is sent to both actors.

The deviation of parameter in the message borrow when it is not an integer type (this deviation was introduced in Figure 4 in the trigger section) is specified as an overridden predefined operation cs_WrongParameter in the use case class. The signal ef_WrongParameter is sent to the bank if this situation occurs. Preprocessing of a corrupted message will invoke behavior assigned to cs_WrongParameter rather than to borrow. CORA defines this.

In the running example, the CORA specification precisely defines in detail both normal and abnormal behavior. Any abnormal situation that was not defined by the CORA model of the system must be ignored by the system, according to CORA semantics. We do not show all the possible predefined deviations in our running example. The next section defines all the elements used in our running example diagrams, along with some others allowed in CORA, and provides their semantics.

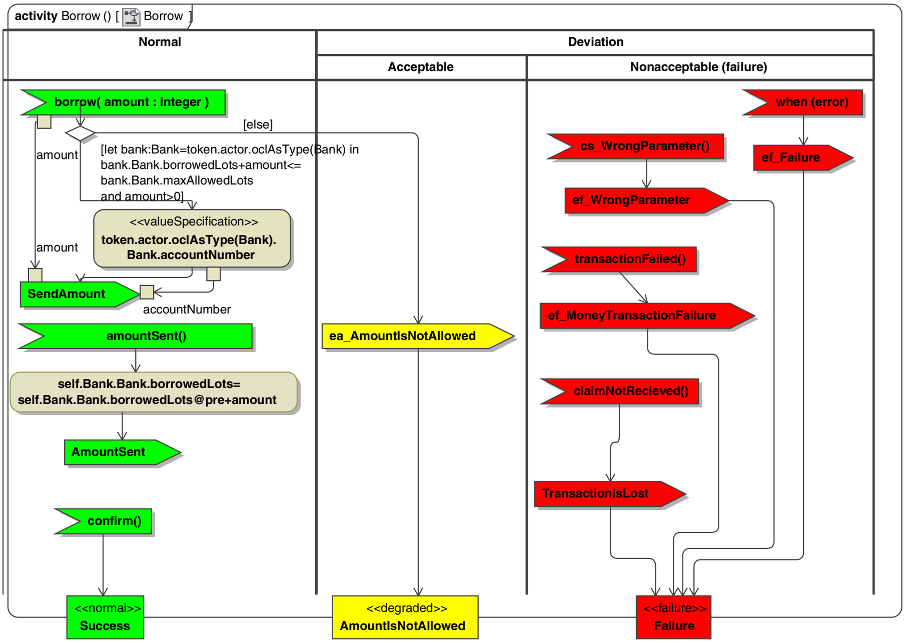


Fig. 7. Activity diagram for Borrow use case

2.6 CORA Syntax and Semantics

2.6.1 Message Processing

We consider three layers in message processing modeling (Figure 8), where the first is the environment represented as a set of actors; the second is a communication system; and the third is the system itself. The layers, Environment and System, are interrelated with the Communication System layer. We have to consider the intermediate communication level since within fault-tolerant system modeling some faults of the communication system may need to be specified along with the recovery. Appropriate error handlers should be defined as part of the development process when determining system's abnormal behavior.

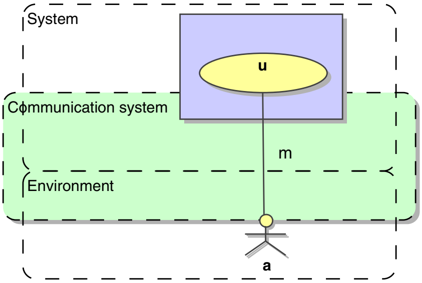


Fig. 8. Layers in message processing

Processing of a received message may change the state of the system and send outgoing messages according to the functional specification. We divide message execution process into three stages: *pre-processing*, where incoming messages are checked against of the model for consistency and as a result, either the message itself or predefined deviation message will be processed during the next stage; *processing*, where use case activity diagram(s) are utilized for processing the messages (the system state is changed and outgoing messages are determined in accordance with the received message or predefined deviation message, along with the Activity Model specification); and *post-processing*, where outgoing messages are sent to the appropriate actor(s). In our model, we consider that outgoing messages can always be sent; we do not say that they will always be delivered by the communication system.

Pre-processing and post-processing are specified in CORA in their standard way when the message processing is determined by the system model. CORA Domain Model describes which messages can start use case, which cannot and which have application in the context of an already initiated use case or role. It also shows which message signatures require supplementary data as use case instance, actor's role, etc. The processing of a message is shown on activity diagram(s) and may include recovery to all abnormal situations in communication system (such as if a message has incorrect parameters, a message cannot be delivered to actor, etc.) and/or incorrect ordering of the messages.

All messages are sequenced by the communication system but may ultimately be processed by the system in a different order if required by the specification. For instance, if the model states that **b** has to follow **c** but communication system receives messages <**a, b, c**>, **b** needs to be deferred so that the system will react as if the messages came in as <**a, c, b**>.

At the requirement elicitation phase, message process is abstracted as if an actor (**a**) sends a message (**m**) to the system and sending this message is described in the context of a use case instance (**u**), which includes description of this step "actor (**a**) sends a message (**m**)." Sometimes for complete identification, it is necessary to say that the actor plays a specific role (**r**) in the use case, which has several participating actors, when sending the message. For instance, a bank might participate as a borrower and a lender simultaneously in the same use case. To completely identify the message, we should use a tuple **M=<m**, **a**, **u**, **r>** even though in most instances some parameters may be reasoned from other parameters. An example of a system where all four parameters are used can be illustrated in a game of chess where one actor simultaneously plays two matches of chess (playing for both players) and sends a message to get information on the last move. If we model one logical actor player with two roles possible in the use case, even though the system receives the information of use case instance and actor, it is not sufficient enough to determine who moved last (player black or player white). When we send the message `getMyLastMove` we want to include the role reference. If the role reference is specified, it implicitly contains the use case and actor instance reference.

In CORA, a full message is defined by a tuple `=<m(p1,…pn), actor, use_case, role>` where `m(p1,…pn)` is the message expression, `m` is the message name and `p1, …pn` are the parameter values; `actor` is the reference of the actor instance, representing the actor who sent the message; `use_case` is the reference of the use case instance that should execute m(p1,...pn) as next step; and, role indicates the role played by actor in use_case. The message tuple is managed automatically at any point in the system life cycle. The CORA semantics described in the next sections distinguish the necessary identifiers that allow us to precisely define the messages. Not all values can be defined for each message, and in these cases we use the predefined null value.

2.6.2 CORA Domain Model

The CORA Domain Model is presented as a class diagram or as a union of a set of class diagrams. It extends Domain Model containing concepts and signals to include System class, Logical Actor classes, Use Case classes, Role classes and associations between them.

The next subsections describe different elements that can be used in the diagram.

System Class

Syntax

There is a predefined <<system>> System class in CORA that identifies general system properties. The System class has predefined operations init, shutdown and processMessage(). The first two, init and shutdown, present only if initialization and finalization behavior is specified. The last, processMessage(), should be abstract. If it is not abstract, a BehavioralFuture (or descendant) which defines full message processing must be assigned to the Method property of the operation. This operation must have attribute concurrency set to guarded or concurrent.

Additional operations may be added to System class if they are not related to use cases. System specification is operation-based rather than actor or use case based. All system operations must be listed in System class and an activity diagram attached to System class or individual activity diagrams must be attached to operations to specify the behavior.

Semantics

The init operation is executed just after the system starts. Operation shutdown corresponds to switching the system off. Both can have activity diagrams attached describing the behavior for initialization and finalization.

The attribute concurrency set to guarded or sequential for operation processMessage() specifies that all incoming messages are sequenced by the communication system and processed in such a way that the next one to be processed is only done so once the previous one is finished and post-processed (the system can actually process messages simultaneously but the resulting system behavior must be equivalent as if they were being processed as specified). The attribute concurrency set to concurrent means that messages are simultaneously processed with the ones following, even with previous messages still processing. In this case, the model becomes highly complicated and the usage of this attribute should be avoided. If the specification states that by receiving message **a1** the system should reply with signal **s1** and when receiving message **a2** the system should reply with signal **s2**, then the guarded mode defines that only **a1, s1, a2, s2** or **a2, s2, a1, s1** sequences can occur. This ensures that other sequencing, like those allowed in `concurrent` mode, **a1 a2 s2 s1** or **a1 a2 s1 s2** or even **a1||a2 s2||s1**, does not take place.

Logical and Physical Actors

The actors from the use case model are defined as physical actors and logical actors in the CORA Domain Model. The physical actors are hidden from the system and are managed only by the communication system. The logical actors are also managed by the communication system but the system specification has a limited ability to use and manipulate logical actor instances (it cannot create or destroy instances, but it can reclassify to another logical actor class and refer the instances).

Syntax

Physical actors are presented in the diagram with their name as a standard UML symbol for actor. Every logical actor is a named class, marked with `<<la>>` stereotype and associated with the physical actor(s).

Each logical actor has associations to all the signals it may receive and all use case classes within which it may participate. The association between logical actor class and use case class is named according to the role played by logical actor in the associated use case. There may be an association class attached to the association. Actor may play different roles in the use case and this is possible by way of different associations.

Logical actor class can have operations defined in it.

Semantics

Multiplicity of the association ends has different meaning for physical actors and logical actors. At the end of physical actor:

- 1 means that exactly one physical actor per one logical actor is allowed and it is always possible to send a message to the physical actor (otherwise, it is a failure of communication);
- 0..1 has the same definition as 1, but physical actor can be inaccessible (typical situation where an internet user, who while interacting with a website, can continue using the website or go to another website);
- More than 1 means that system can represent several physical actors as one logical actor (several real buttons represented by the system as one logical button) and a message sent from any of the physical actors is considered to be coming from the logical actor. All messages sent to a logical actor are broadcasted among all the physical actors.

At the end of logical actor:

- 1 means that only one logical actor is allowed for the physical one (like web forums not wanting different users working simultaneously from one computer);
- More than one means that the customer allows user to be presented as different logical actors.

It is devised with CORA that communication system manages the instances of logical actor classes and encapsulates physical ones. In specifying the system, we can refer to instances of logical actors and can rely on the communication system to control and manipulate references to the actor, use case and role classes within pre-processing and post-processing of the messages. For example, when developing a website we can concentrate on specifying only system behavior and expect the communication system to distinguish actors by sending and matching cookies or by some other means detailed during design. The specifications for operations defined within a logical actor class are described in the Operations Section below.

Use Case Class

Syntax

Every use case from use case diagram has an activity diagram with the same name as the source use case and attached with `OwnedBehavior`. This is presented in the CORA Domain Model as a <<`uc`>> stereotyped class (it also has UML standard <<`activity`>> stereotype, which is suppressed).

Every use case class is associated by the realization association with the appropriate use case. This implicitly applies all the relations between use cases which were made in use case diagrams, such as include, extend, handler, etc.

Associations between use case class and actors (see Logical Actor Section) and between use case and signals (see Signals Section) are shown in the diagram.

Semantics

The activity diagram defines the behavior as it was informally defined in use case description. Full semantics will be described below in the CORA Activity Model section.

Signals

Syntax

All the signals that can be sent to actors must be defined in CORA Domain Model. The name of the signal is the name of class with UML standard <<`signal`>> stereotype, and parameters are defined as attributes accordingly.

The signal is associated to all the actor(s) to which it may be sent. It can also be associated to use case classes, which can send this signal and/or to use case role classes.

Semantics

The meaning of the associations is to define which references are sent as additional parameters to the actor. During processing of the message, according to the CORA Activity Model, `SendSignalAction` execution creates instances of signal class and creates links to use case (if there is an association between the signal and the use case) and to the role instance in the same way. Then, the communication system sends the signal to the physical actor together with the references to all the linked objects, providing ability for the actor to send the next message with the context in which it should be executed.

Operations

Syntax

Operations can be defined in System class, Logical Actor classes, Use Case classes and Role classes. Operations in Logical Actor, Use Case and Role can be static or non-static, where in System class they must be non-static. The signature of the operations is UML 2.0 compliant. Some operations may have <<autoCreate>> dependency(ies) to Use Case class(es) or Role class.

There are several predefined operations in CORA: init, shutdown, processMessage, cs_UnknownMessage, cs_WrongParameter, cs_UnknownActor, cs_ActorMustBeProvided, cs_WrongActor, cs_WrongUseCase, cs_WrongRole, cs_UseCaseMustBeProvided, cs_RoleMustBeProvided.

Semantics

As described in message processing, a full message contains four parameters: the message itself and references to use_case, actor and role.

If an operation is listed in System class it requires the correct message name and parameters, otherwise cs_UnknownMessage or cs_WrongParameter is processed.

If an operation is listed in a Logical Actor's class, it requires the communication system to determine the actor's reference and place at AcceptEventAction(s) corresponding the message name a full message tuple with other parameters (apart from the message parameter) set to null, in accordance to activity diagrams. If the operation is static, Logical Class reference, rather than instance reference should be passed in the tuple because the communication system will only be able to determine that the message received belongs to class, as it does not contain information about the instance. This situation happens, for example, if the reference cannot be sent to actors and consequently would not be required (and received) with the message. For example, if there are several sensors connected to the system, by making the operation non-static we would require that the system has precise information for which sensor sends a value. However, if the system requirement states that it cannot distinguish sensors, such a situation is abstracted using static operation defined in logical class. If the reference is required but not provided (and cannot be reasoned by the communication system) cs_UnknownActor, cs_ActorMustBeProvided or cs_WrongActor are processed instead.

If the operation is listed in Use Case class it has meaning only in the context of this use case. If it is static, it may be invoked without supplying the use case reference (an operation may trigger the use case by creating a new instance of corresponding class), otherwise the reference must be present, or cs_UseCaseMustBeProvided or cs_WrongUseCase are processed instead.

If the operation is listed in Use Case Role class it has meaning only in the context of this role. If it is static, the reference to role class instance is not necessary. If it is non-static, the reference must be present with the message and accurate, otherwise cs_WrongRole or cs_RoleMustBeProvided are processed.

The <<autoCreate>> dependency shows which operations are triggers for the use case (defined in Use Case or Role class they are static, and may be non-static in Logical Actor class) and prompt the use case instance to be created as an initial step of operation processing. Note, that in some complex cases, instantiating of Use Case classes and Role classes may be more complex and may be shown explicitly in CORA Activity Diagram using CreateObjectAction.

If the same operation signature appears in different activity roles or Logical Actor classes, semantically it is not the same operation, since the signature includes the references implicitly. Therefore, a normal form of the class diagram may introduce a generalization in order to factorize this operation occurring among activity classes or different roles.

Control Variables

Syntax

Control variables are defined as attributes of System class, Logical Actor classes, Use Case classes or Role classes. These classes are specific control variables and are used in our model and marked with <<system>>, <<la>>, <<uc>> and <<role>> stereotypes. If a control variable is defined as an attribute of another class in Domain Model, the attribute is stereotyped <<cv>>.

Semantics

The control variables are used to specify complex behavior of the system. Since they are not part of the Domain Model and the designer does not have to design them, they are marked with special stereotypes. However, the CORA model requires the system to behave in such a way that all the invariants, pre/post-conditions, as well as guards which utilize control variables, are satisfied at the runtime. In this paper's running example, we did not need and consequently did not use additional control variables apart from the standard CORA classes (Use Case, Logical Actor and Role).

2.6.3 CORA Activity Model

Syntax

CORA Activity Model consists of a set of CORA Activity Diagrams, each one attached to a use case and has three activity partitions: normal (describing normal behavior), and abnormal (describing abnormal behavior) divided into two subpartions: acceptable and failure (not acceptable).

Each message, which is described in the Activity Diagram, must have corresponding AcceptEventAction with the trigger attribute set to CallEvent and with the operation attribute set to one of the operations defined in the corresponding CORA Domain Model diagram.

AcceptEventAction may have ControlFlow edges linking it to DecisionNode, OpaqueAction, ReclassifyObjectAction, CreateLinkAction, SendSignalAction, CreateObjectAction (and all other actions manipulating objects and links). They in turn can have ControlFlow edges linking them to any of the actions listed in the previous statement. AcceptEventAction may have OutputPin with the parameter of operation. The diagram also uses ObjectFlow edges, which can connect pins, ObjectNodes and ActivityParameterNodes.

Semantics

When the system processes the message it puts the message tuple as a token to every CORA Activity Diagram containing corresponding AcceptEventAction for the message (operation). If the corresponding operation is static and has dependency with <<autoCreate>> stereotype, then new instances of Use Case class or Role class are created and tuple will contain the references to the newly created objects. Then the token goes from one node to another according to general UML 2.0 semantics for activity diagrams. The guards can contain OCL 2.0 expressions and may use reserved keyword token to refer the token's message parameters, use case instance, actor instance and role instance (some of them may be null). OpaqueAction, with an OCL expression inside, shows the postcondition to this action execution and may use standard @pre postfix in the statement. This action can also manipulate token parameters. For example, when we have to create a new use case instance, we want to replace the token's use case reference as well.

Even though OpaqueAction sufficiently expresses any type of information manipulation in CORA model, some other actions may be used with UML 2.0 predefined semantics. For example, ReclassifyObjectAction is used in our running example to express the meaning of registering an actor so that it becomes another actor. This makes the model more pictorial and hides the unnecessary details, in comparison with OCL expression, which could be used instead.

If token is consumed by SendSignalAction, then a signal instance corresponding to the action is created for every actor instance linked to the use case instance referred by the token. If the use case instance reference is null, then a signal instance is created for every actor instance that has an association to the signal class. A link is then created to the corresponding actor and to use case instance or role instance if there are associations between the signal class and use case or role classes. During post-processing of the message, the communication system will actually send every signal instance to the physical actor together with references to linked instances (or the equivalent information which will allow the communication system to later form the token with the appropriate use case and other instances references).

If a node has an ObjectFlow edge with ActivityParameterNode, then a token offered by the node passes this edge and places a use case instance at ActivityParameterNode. The use case should be finished with the outcome, which is the value of this use case return parameter linked with ActivityParameterNode. If the use case instance cannot be consumed by ActivityParameterNode, because there are no connected ObjectFlow edges, the use case instance is destroyed. However, in the future we may be able to demonstrate that when the activity diagram is nested to other activity diagrams as an activity instance, the token can be consumed by an edge connected to OutputPin corresponding with the ActivityParameterNode of the nesting activity diagram.

When token has passed a node that has no outgoing edges it is destroyed (but not the use case instance). When there are no more tokens, the operation has been executed completely and the post-processing of the message is started.

2.6.4 CORA Operational Model

The operational model is implicit in CORA. Each operation can be defined with post-conditions derived from all CORA Activity Diagrams and the CORA Domain Model. The activity diagram containing `AcceptEventAction`, corresponding to the operation, defines a part of postcondition that can be formed by transforming the tree of all the nodes that the action connects to with outgoing edges; in turn, they (recursively) connect to other nodes. These parts, derived from all of the activity diagrams using this operation are put together with conjunction operand and thus form the operation postcondition.

For example, operation `amountSent(use_case:Borrow)`:

```
Post:
use_case.Bank.Bank.borrowedLots=
use_case.Bank.Bank.borrowedLots@pre+amount and Bor-
rower^AmountSent()
```

Note, that this operation uses the implicitly defined control variable `amount` set by operation `borrow(amount:Integer)`, as well as the implicitly defined parameter `use_case:Borrow` defined in CORA Domain Model by putting the operation `amountSent()` in `Role` class.

3 Related Works

There is work presenting a formal technique for requirement elaboration for capturing abnormal behavior in the form of obstacles, using temporal logic formalization of goals and domain properties [9]. Compared to our method, it is formal, complete and robust but lacks simplicity and visualization. With little training, CORA models can be used by any UML expert.

There is also a method for requirement analysis that uses the OCL and UML activity diagrams to capture the requirements with the intent of a future test generation [13]. This method, however, does not focus on abnormal behavior.

The Exception-Aware Requirement Elicitation method [14] works with abnormal behavior at the level of requirements. It gives informal requirements with captured normal and abnormal behavior and shows how to map Exceptional Use Cases to DA-Charts for analysis and can be used as an input that can then be analyzed and precisely specified using CORA models. This method is in line with the CORA method.

4 Perspectives and Conclusion

Future work on this topic is planned in several directions. First of all, all models of CORA should be enhanced with transactional behavior. Secondly, deviations and recovery only deal with a part of dependable requirements and should include other aspects of dependability like, availability, security, etc. Thirdly, testing is costly in the development process and should be simplified through automatic test generation from the analysis model. We would like to study the links between fault-tolerant requirement specification and test case.

Acknowledgments. We would like to thank Alexander Romanovsky and Jörg Kienzle for their helpful discussions. We also appreciate reviewing and suggestions from Jennifer Witcher. This research is partially supported by the Ministry of Culture, Higher Education and Research of Luxembourg by grant BFR04/053.

References

1. Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing 1(1), 11–33 (2004)
2. Beder, D.M., Randell, B., Romanovsky, A., Rubira, C.M.F.: On Applying Coordinated Atomic Actions and Dependable Software Architectures for Developing Complex Systems. In: ISORC - 2001 Proceedings. Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, pp. 103–112. IEEE Computer Society Press, Los Alamitos (2001)
3. Capozucca, A., Guelfi, N., Pelliccione, P.: The fault-tolerant insulin pump therapy. In: Butler, M., Jones, C.B., Romanovsky, A., Troubitsyna, E. (eds.) Rigorous Development of Complex Fault-Tolerant Systems. LNCS, vol. 4157, pp. 59–79. Springer, Heidelberg (2006)
4. Cockburn, A.: Structuring use cases with goals. Journal of Object-Oriented Programming, in two parts, the September-October issue and the November-December issue (1997)
5. ISO/IEC 12207: Information Technology - Software life cycle processes (1995)
6. Guelfi, N., Perrouin, G.: A flexible requirements analysis approach for software product lines. In: Sawyer, P., Paech, B., Heymans, P. (eds.) REFSQ 2007. LNCS, vol. 4542, pp. 78–92. Springer, Heidelberg (2007)
7. Guelfi, N., Perrouin, G.: Using Model Transformation and Architectural Frameworks to Support the Software Development Process: the FIDJI Approach. In: 2004 Midwest Software Engineering Conference, pp. 13–22 (2004)
8. Kaaniche, M., Laprie, J.-C., Blanquart, J.-P.: A framework for dependability engineering of critical computing systems. Safety Science, vol. 40, pp. 731–752. Elsevier Science Ltd., Amsterdam (2002)
9. Lamsweerde, A., Letier, E.: Handling Obstacles in Goal-Oriented Requirement Engineering. IEEE Transactions on Software Engineering 26(10) (2000)
10. Laprie, J.C.: Dependability: A Unifying Concept For Reliable Computing and Fault Tolerance. In: Anderson, T. (ed.) Dependability of Resilient Computers. BSP Professional Books, Oxford (1989)
11. Mustafiz, S., Kienzle, J.: Addressing Dependability in Use Case Driven Requirements Elicitation. Tech. Rep. SOCS-TR-2006.3, McGill University, Montreal, Canada (2006)
12. Mustafiz, S., Sun, X., Kienzle, J., Vangheluwe, H.: Model-driven assessment of use cases for dependable systems. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 558–573. Springer, Heidelberg (2006)
13. Nebut, C., Baudry, B., Kamoun, S., Saeed, W.A.: Multi-Language Support for Model-Driven Requirement Analysis and Test Generation. In: ECMDA workshop on Integration of Model Driven Development and Model Driven Testing, Bilbao, Spain (2006)
14. Shui, A., Mustafiz, S., Kienzle, J.: Exception-aware requirements elicitation with use cases. In: Dony, C., Knudsen, J.L., Romanovsky, A., Tripathi, A.R. (eds.) Advanced Topics in Exception Handling Techniques. LNCS, vol. 4119, pp. 221–242. Springer, Heidelberg (2006)
15. Parnas, D.L.: On the Design and Development of Program Families. IEEE Transactions on Software Engineering 2(1), 1–9 (1976)

Step-Wise Development of Resilient Ambient Campus Scenarios

Alexei Iliasov, Budi Arief, and Alexander Romanovsky

School of Computing Science, Newcastle University,
Newcastle upon Tyne NE1 7RU, England
{Alexei.Iliasov,L.B.Arief,Alexander.Romanovsky}@newcastle.ac.uk

Abstract. This paper puts forward a new approach to developing resilient ambient applications. In its core is a novel rigorous development method supported by a formal theory that enables us to produce a well-structured step-wise design and to ensure disciplined integration of error recovery measures into the resulting implementation. The development method, called AgentB, uses the idea of modelling database to support a coherent development of and reasoning about several model views, including the variable, event, role, agent and protocol views. This helps system developers in separating various modelling concerns and makes it easier for future tool developers to design a toolset supporting this development. Fault tolerance is systematically introduced during the development of various model views. The approach is demonstrated through the development of several application scenarios within an ambient campus case study conducted at Newcastle University (UK) as part of the FP6 RODIN project.

1 Introduction

We use the term *ambient campus* to refer to the *ambient intelligence* (AmI)[1] systems deployed in an educational setting (a university campus). Ambient campus applications are tailored to support educational, administrative and research activities typically found in a campus, including delivering lectures, organising meetings, and facilitating collaborations among researchers and students.

This paper reports our work on the development of the ambient campus case study within the RODIN project [1]. This EU-funded project, led by the School of Computing Science of Newcastle University, aimed to create a methodology and supporting open tool platform for the cost-effective rigorous development of dependable complex software systems and services. In the RODIN project, the ambient campus case study acted as one of the research drivers, where we investigated how to use formal methods combined with advanced fault-tolerance techniques in developing dependable AmI applications.

[1] A concept developed by the Information Society Technologies Advisory Group (ISTAG) to the EC Information Society and the Media DG, where humans are surrounded by unobtrusive computing and networking technology to assist them in their activities – http://cordis.europa.eu/ist/istag.htm.

M. Butler et al. (Eds.): Fault Tolerance, LNCS 5454, pp. 297–323, 2009.
© Springer-Verlag Berlin Heidelberg 2009

Software developed for AmI applications needs to be able to operate in an unstable environment susceptible to various errors and unexpected changes (such as network disconnection and re-connection) as well as to deliver context-aware services. These applications tend to rely on the *mobile agent paradigm*, which supports system-structuring using decentralised and distributed entities (*agents*) working together in order to achieve their individual aims. Development of multi-agent applications poses many challenges due to their openness, the inherent autonomy of their components (i.e. the agents), the asynchrony and anonymity of their communication, and the specific types of faults they need to be resilient to. To address these issues, we designed a framework called CAMA *(Context-Aware Mobile Agents)*, which encourages disciplined development of open fault-tolerant mobile agent applications by supporting a set of abstractions ensuring exception handling, system structuring and openness. These abstractions are backed by an effective and easy-to-use middleware allowing high system scalability and guaranteeing agent compatibility. More details on CAMA and its abstractions can be found in [2,3,4].

The rest of this paper discusses the challenges in developing fault tolerant AmI systems (Section 2), describes the theory behind our design approach (Section 3), outlines various approaches to tackling fault-tolerant issues (Section 4), and illustrates how our design approach was applied in the case study scenarios (Section 5).

2 Challenges in Developing Fault-Tolerant Ambient Intelligence Systems

Developers of fault-tolerant AmI systems face many challenging factors, some of the most important ones are:

- *Decentralisation and homogeneity*

  AmI systems are composed of a number of independent computing nodes. However, while traditional distributed systems are *orchestrated* – explicitly, by a dedicated entity, or implicitly, through an implemented algorithm – in order to solve a common task, agents in AmI system make *independent* decisions about *collaboration* in order to achieve their individual goals. In other words, AmI systems do not have inherent hierarchical organisation. Typically, individual agents are not linked by any relations and they may not have the same privileges, rights or capabilities.
- *Weak Communication Mechanisms*

  AmI systems commonly employ communication mechanisms which provide very weak, if any, delivery and ordering guarantees. This is important from the implementation point of view as AmI systems are often deployed on wearable computing platforms with limited processing power, and they tend to use unreliable wireless networks for communication means. This makes it difficult to distinguish between a crash of an agent, a delay in a message delivery and other similar problems caused by network delay. Thus, a recovery mechanism should not attempt to make a distinction between network failures and agent crashes unless there is a support for this from the communication mechanism.

- *Autonomy*

  During its lifetime, an agent usually communicates with a large number of other agents, which are often developed in a decentralised manner by independent developers. This is very different from the situation in classical distributed system where all the system components are part of a closed system and thus fully trusted. Each agent participating in a multi-agent application tries to achieve its own goal. This may lead to a situation where some agents may have conflicting goals. From recovery viewpoint, this means that no single agent should be given an unfair advantage. Any scenarios where an agent controls or prescribes a recovery process to another agent must be avoided.
- *Anonymity*

  Most AmI systems employ anonymous communication where agents do not have to disclose their names or identity to other agents. This has a number of benefits: agents do not have to learn the names of other agents prior to communication; there is no need to create fresh names nor to ensure naming consistency in the presence of migration; and it is easy to implement group communication. Anonymity is also an important security feature - no one can sense an agent's presence until it produces a message or an event. It is also harder to tell which messages are produced by which agent. For a recovery mechanism, anonymity means that we are not able to explicitly address agents which must be involved in the recovery. It may even be impossible to discover the number of agents that must be involved. Even though it is straightforward to implement an exchange for agents names, its impact on agent security and the cost of maintaining consistency usually outweigh the benefits of having named-agents.
- *Message Context*

  In sequential systems, recovery actions are attached to certain regions, objects or classes which define a context for a recovery procedure. There is no obvious counterpart for these structuring units in asynchronously communicating agents. An agent produces messages in a certain order, each being a result of some calculations. When the data sent along with a message cause an exception in an agent, the agent may want to notify the original message producer, for example, by sending an exception. When an exception arrives at the message producer (which is believed to be the source of the problem), it is possible that the agent has proceeded with other calculations and the context in which the message was produced is already destroyed. In addition, an agent can disappear due to migration or termination.
- *Message Semantics*

  In a distributed system developed in a centralised manner, semantics of values passed between system components is fixed at the time of the system design and implementation. In an open agent system, implementation is decentralised and thus the message semantics must be defined at the stage of a multi-agent application design. If an agent is allowed to send exceptions, the list of exceptions and their semantics must also be defined at the level of an abstract application model. For a recovery mechanism, this means that each agent has to deal only with the exception types it can understand, which usually means having a list of predefined exceptions.

We have to take these issues into account when developing and implementing fault-tolerant AmI systems. The following section outlines the design approach intended for constructing the ambient campus case study scenarios.

3 Design Approach

Our overall aim is to develop a fairly detailed model which covers a number of issues critical for ambient systems, including communication, networking failures, proactive recovery, liveness, termination, and migration.

No existing formal modelling technique can adequately address all the development stages of a complex, large-scale agent system. The proposed modelling method combines high-level behavioural descriptions, detailed functional specifications and agent-level scenarios for an integral approach addressing the issues of parallelism, distribution, mobility and context.

Different modelling techniques are used to operate on a common general model. A general system model is projected through a number of views, each emphasizing some specific aspect of a model. The choice of views was dictated by the availability of the verification toolkits that can be used to automate analysis of model properties. Another important role of views is to help a designer to better understand a model.

Unlike hybrid methods – where two or more notations are used – in our approach, a whole model is described in single basic notation based on the standard set theory language. This reduces the possibility of consistency problems and makes the method more elegant and flexible. The model notation is deliberately very schematic, no tool is going to support it and no real system can be described in it. Instead we propose to use a graphical modelling tool that can visually layout and manipulate different model parts.

Since it is hard to produce an efficient and scalable verification tool, we try to reuse the well-established formalisms supported by verifications tools. Our approach is based on a combination of a process algebra and a state-based modelling method. A CSP-inspired process algebra is used for verification of high-level system behaviour while the Event-B specification method [5] is employed to construct detailed functional specifications. The Mobility Plugin model checker [6] is used to verify hybrid specifications composed of a process algebraic model and an Event-B machine.

The motivations for this work is the construction of a tool for computer-aided development of agent systems. Such tool would combine simplicity of visual modelling tools such as UML, the expressive power of specifications languages such as B ([7]) and Z ([8]) and systematic step-wise development of the refinement.

The result of such development is a set of specifications of agent *roles*. Due to the top-down development, such specifications are interoperable. Role specifications can be taken apart and implemented or further developed independently without risking losing interoperability with the other agent roles of the system. In many cases, executable code can be generated directly from a role specification.

3.1 The AgentB Modelling Database

The AgentB modelling database concept unites the various model types used in a development. Most of the models are rather simple on their own and thus easy to read and update. Combining these models together produces a powerful modelling tool, and their combination is never written out as a single specification. Instead, a software tool is responsible for maintaining links among model parts to ensure consistency of the whole development.

A refinement of a formal development is constructed by refining different database parts, one at a time. This is possibly the most attractive feature of the approach. Instead of tackling a whole model, a modeller can choose a specific aspect of a model to work on. At any given moment, the focus of a modeller is on a single database part or a synthetic view constructed from a combination of several parts. The modelling database has the following structure:

$$Sys = (S, V, E, R, P, F, C, D, A, I)$$

where

S - collection of carrier sets and variables. They are used in the functional, communication and agent database parts.

V - variable model. This includes variables used by functional, communication and agent models.

E - event model, as a set of possible system events.

R - role model, collection of system roles.

P - a protocol model, in the form of a CSP-like process algebraic expression. A protocol model is a high-level description of observable system behaviour. This model does not refer to or update system state. For this reason it is very convenient to use the protocol part alone to design an initial system abstraction.

F - functional model. It describes the state updates done by events present in the system. Functional model of a single event includes a set of local variables, a guard, and a before-after predicate relating a new system state to an old one.

C - communication model. It is used to describe how information is passed between different agents roles, as roles do not normally share variables. The model helps to distinguish between internal control flow of an agent and external message passing.

D - distribution model. This model relates elements of the event and variable models to roles from the role model. An empty distribution model stands for an implicit single role which contains all the variables and events.

A - agent model. This model describes *locations* and *agents* of a system. The model helps to address the problems of mobility and context-awareness.

I - a system invariant. Properties expressed by a model invariant must be preserved at all stages of a system execution. Typically, an invariant consists of typing predicates for system variables, functional model properties, agent model properties and, possibly, a gluing invariant for linking model refinements.

Model context contains static information used by a development. It declares user carrier sets, constants and a context properties: (S, C, P).

Variable Model. The variable model part describes the state space of the modelled system. This model must be accompanied by the invariant part providing typing predicates for all the variables.

In a modelling database, the variable model part is used by four other parts – functional, distribution, communication and agent models. All variables are visible to these parts. The initialisation event of the functional model is responsible for computing the initial state of a system. Agent and functional models are the only parts which can update variable states.

Event Model. Event is an observable action of a system, it has no duration and thus only one event can be observed at any given moment. An event model indicates which events may be observed in a correct model implementation. For example, for event model $\{a, b\}$, all the systems with the following observable behaviours are correct implementations:

$$\begin{array}{l} \langle a, a, a, ...\rangle \\ \langle b, b, b, ...\rangle \\ \langle a, a, b, a, b, b, ...\rangle \\ \langle\rangle \end{array}$$

where by $\langle\rangle$ we denote a system which stops immediately when started (a system doing nothing is a valid implementation of any event model). Any system with events other than a and b is not a valid model implementation. For instance, $a, a, a, c, ...$ does not implement the model since c is not included in the event model.

Role Model. The role model part declares a set of system roles (component types) and for each role, specifies the minimum number of role instances required to construct a system and the maximum number of role instances that is supported by a system. A role model is defined with a tuple made of a set of roles and two functions defining the restriction on role instance number:

$$(R, \mathrm{rmin}, \mathrm{rmax})$$

where R is the set of roles, rmin and rmax are functions specifying the minimum and maximum number of instances for each role:

$$\mathrm{rmin} : R \rightarrow \mathbb{N}_1$$
$$\mathrm{rmax} : R \rightarrow \mathbb{N}_1 \cup \{\infty\}$$

Protocol Model. Behavioural modelling is a natural choice for high-level modelling of parallel and distributed systems. Behavioural specifications focus on temporal ordering of events, omitting details of state evolution. The agent system paradigm is one example where behavioural model is preferable for high-level system abstraction although state-based description may be required at later stages. The protocol model language is based on a subset of CSP notation (Figure 1).

$e \rightarrow P$	synchronisation prefix
$P; Q$	sequential composition
$P \| Q$	parallel composition
$P \sqcap Q$	choice
$\mu x \cdot P(x)$	recursion
skip	no-effect process

Fig. 1. The language of protocol model expression

We will often use the following shortcut notation for describing loops:

$$^*(P) = P; P; P; ... = \mu X \cdot (P; (X \sqcap \mathbf{skip}))$$

Reactions In AgentB, we are interested in the modelling of distributed systems. To make transition into implementation stage easier, we try to achieve distribution at the modelling level. Protocol model is one of the parts that must be split somehow into pieces to faithfully model a distributed system. For this, we represent a protocol model as a collection of several independent protocol models:

$$P_1, P_2, ..., P_n$$

To be able to refer to parts of a protocol model, protocol sub-models are identified with unique labels:

$$l_1 : P_1, l_2 : P_2, ..., l_n : P_n$$

Here P_i are the protocol model parts and l_i are the attached labels. For example, a model of a server providing two different services – reading a file and saving a file – can be described as:

$$readfile : P \|$$
$$savefile : Q$$

Each reaction name has a special meaning. Reaction with label $\star$ is a protocol model of a whole system before it is completely decomposed into models of individual roles. Other reactions labels are only notational decorations and are not given any interpretation.

Functional Model. With the functional model part, a modeller specifies how an event updates the state of a system. This is done by formulating predicates relating old and new system states. Such a predicate does not have to describe a unique new state, instead it describes a whole family of possible next states. Functional model of an event is equipped with a guard. A guard defines the states when the event can be enabled. If an event execution is attempted in a state prohibited by its guard, execution is suspended until the system arrives at a state satisfying the guard.

Functional model of an event computes a new system state in a single atomic step. It does not use any intermediate steps or intermediate local results and it is not interleaved with the execution of other events. Functional model always contains initialisation event. This event is special: it cannot be referred-to anywhere in a model (for example, in a protocol model expression) and this event is always prior to any other event. The functional model of an event is described by event guard and event action:

$$F : Ev \mathrel{+\!\!\!\!\!\rightarrow} (Grd \times Act)$$

where

Ev - event identifier, must be an element of the Event model;
Grd - event guard. In addition to typing predicates for parameters and event enabling conditions, an event guard can also have free variable that must be typed by the guard. These variables are the local variables of the event. They are not seen outside the event and cannot be updated by the event action;
Act - generalised substitution defined on variables from the variable model.

We use generalised substitutions to describe how an action transforms a model state. The table below lists the substitution styles that are used to describe an action:

notation	relation predicate	
$v := F(c, s, v, l)$	$v' = F(c, s, v, l)$	assignment
skip	$v' = v$	no-effect assignment
$v :\in F(c, s, v, l)$	$v' \in F(c, s, v, l)$	set choice
$V :\mid F(c, s, V_0, V_1, l)$	$F(c, s, V_0, V_1, l)$	generalised substitution

where v is a variable, F is an expression, V is a vector of variables and V_0 and V_1 are the old and new values of V. Expression F may refer to constants c, sets s, system variables v and local variables l. The first of the substitution type, $:=$, is a simple assignment. The assigned variable becomes equal to the value of expression F. Substitution $v :\in F$ selects a new value for v such that it belongs to set F. The most general substitution operator, $:\mid$, uses a predicate to link the new and old model states.

Several substitution types can be combined into a single action with the parallel composition operator:

$s_1 \| s_2 \| ... \| s_k$

We perceive parallel composition of actions as a simultaneous execution of all the actions. We can always replace the set of parallel substitutions with a single generalised substitution.

Communication Model. The communication model allows a modeller to analyse and update communications that may occur in a system. By communication we understand a pair of ”a sending event” and ”a receiving reaction” (described by the protocol model part) and a predicate-binding parameters that define the communication-enabling conditions. As with the functional model part, the communication model does not have explicit parameters. Parameter passing is modelled by the conjunction of event and communication guards.

The purpose of this model is to keep the information about communication separate from other parts. The reason to do this is because we cannot assign communication to protocol model as it would make it very hard to achieve decomposition into agent models which is important to our method. Communication cannot be described in the functional model part as a functional model is formulated on per-event basis. Introducing communication would destroy this simple architecture.

Communication is introduced when a system has more than one role. To make sure that parts of the system that are to be implemented as independent components are linked in a manner that does not prevent their distribution, we use communication model to describe possible messages exchanged by such components.

A communication model associates a set or sets of communications with a source event (message sender). Each communication is a tuple of a guard and a destination event:

$$C : Ev \mathrel{+\!\!\!\!\rightarrow} \mathcal{P}(Grd \times Rct)$$

where Ev is the message source – the event sending the message. Predicate Grd determines whether a message should be sent and the values for the parameters should be passed to the designation event. The message target is a reaction name.

Distribution Model. Distribution model defines how the functionality and the state of a model are partitioned among the roles of a system. This permits a system to be realised as a set of independent components. Each variable and event of a model is associated with a particular role and additional restrictions are imposed on protocol and functional models. Formally, a distribution model is described as a tuple of functions partitioning events and variables:

$$D = (D_e, D_v)$$

where function $D_e : Ev \twoheadrightarrow \mathcal{P}(R)$ maps an event into a role to which the event belong. Function $D_v : V \twoheadrightarrow \mathcal{P}(R)$ does the same for a model variable.

Agent Model. An agent model is a tuple of locations set L and agent specifications A:

$$M = (L, A)$$

An agent specification describes the behaviour of a single agent. At any given moment, an agent is located at some location from the set L. The set L always contains the predefined location *limbo* which is understood as the whole of the 'outside' world. From this location, new agents appear in a system and via this location agents may leave a system.

The role of the location concept is to structure an agent system into disjoint set of communicating agent groups. This addresses the scalability problem of agents-system. A large and complex system can be described as a composition of smaller and simpler sub-systems, well isolated from each other.

An agent may communicate with other agents in the same location. Agents from different locations may also communicate. An agent may decide to change its position by migrating to a new location. This changes the set of agents it sees and can communicate to. The structuring of an agent system and agent grouping is dynamic. An agent can use its current state, produced by interacting with other agents, to compute the next migration destination. This permits the description of dynamic agent systems with complex reconfiguration polices.

Specification of an agent behaviour is a process algebraic expression. The decomposition model makes sure agents are defined in a non-conflicting manner.

The starting point for the construction of an agent system is the assignment of a set of roles to each agent. An agent with roles R is understood to be a component implementing the complete functionality of all the roles from R and is required to provide all the services attributed to these roles.

An agent model is described by two functions: *Arl* provides the list of roles implemented by an agent and *Asp* returns an agent specification.

$$\begin{aligned} Arl &: Agt \mathrel{+\!\!\!\rightarrow} \mathcal{P}_1(R) \\ Asp &: Agt \mathrel{+\!\!\!\rightarrow} Sp \end{aligned}$$

We require that $dom(Asp) = dom(Arl) \neq \oslash$.

An agent specification is described using a small subset of CSP, which features the following constructs:

$Pred?a$	guarded action
$P;Q$	sequential composition
$P\Vert Q$	parallel composition
$P \sqcap Q$	choice
$Pred?^{*}(P)$	loop

where action a is:

$\overline{[m,p]}$	invocation of an internal reaction
$\mathbf{go}(l)$	migration
$evt(p_1, p_2, ..., p_n)$	execution of an event from a functional model
$[m,p]$	communication process sending event m with parameters p

The migration action changes the current position of an agent. Invocation of an internal reaction results in a creation of a new process within an agent. Such a process is described by a combination of protocol and functional models. An agent can send a message to another agent provided the destination agent can be found at the current location. As a reaction to a message, the receiver creates a new process with the internal reaction invocation action. Finally, an agent model may call an event defined in the functional model of an agent.

To summarize, the proposed modelling framework has been developed to fit well with the major characteristics of the agent systems identified in Section 2. Role, distribution and communication models guarantee agent decentralization and weak communication. The event-based communication between agents ensures their anonymity. Agents are autonomous and do not have to communicate if this does not fit their goals. The framework supports independent development of individual agents in such a way that they are interoperable, function in the distributed settings and can move by changing locations when and where they want to achieve their individual goals.

4 Fault-Tolerance

To ensure fault tolerance of complex ambient applications, we address the fault-tolerance issues through the entire development process starting from eliciting relevant operational and functional requirements. In our approach, system operational and functional requirements – among other information – capture all possible situations which are abnormal from the point of view of the system stakeholders (including, system users, support, developers, distributors and owners). First of all, this allows us to state the high level fault assumptions, which, generally speaking, define what can go wrong with the system – and as such, needs tolerating – and what we assume will never go wrong (the latter is as important as the former as it defines the foundation on which fault tolerance can be built). These requirements guide the modelling of the error detection and system recovery.

Due to the complex nature of large-scale AmI applications – caused by their dynamic nature and openness – the traditional fault tolerance structuring techniques, such as procedure-level exception handling, atomic transactions, conversations and rollback cannot be applied directly as the systems typically need combined approaches used for dealing with different threats in different contexts. Within our modelling approach, fault tolerance becomes a crosscutting concern integrated into a number of model views and at different phases of incremental system development. System structuring, ensuring that potential errors are contained in small scopes (contexts) represented as the first class entities during system modelling, is in the core of this approach (in the same way as it is in the core of providing any application fault tolerance [9]).

Fault tolerance is systematically introduced during the development of various model views. Thus, the event model includes both normal and abnormal events, where the latter represents various situations ranging from detecting errors to successful completion of system recovery. Each role model typically constitutes a simple scope, which becomes the first level of system recovery, to be conducted by the individual role, without involving other agents or other roles of the same agent. In some situations, this type of recovery can be successful considering the agent's autonomy and the decentralized nature of the AmI applications.

Unfortunately, our experience shows that in real systems, we often need to conduct a higher level recovery which involves other agents. There are many reasons for this, including cooperative and interactive nature of these applications, in which agents come together to achieve their goals, so that they often need to cooperate to recover and to ensure that during and after recovery, the whole system is in a consistent state.

Within AgentB, different model views deal with faults and errors in view-specific ways and use specific fault tolerance measures. The protocol model allows us to raise and propagate exceptional messages and to conduct application-specific recovery modelled as a separate part of each protocol (providing a special form of exception handler – see [10]). The abnormal part of the protocol view shows message sequences typically exchanged during system recovery.

In the agent models, we introduce fault-tolerance properties at the level of agents, which are the units of deployment and mobility in the AmI systems, as well as at the level of groups of cooperating agents. This allows us to represent fault tolerance at both: the level of individual agents and the level of groups of agents deployed in the same location. In particular, we can represent the use of redundancy (e.g. to achieve fault handing by spawning an agent copy to survive an agent crash) and diversity (to achieve error recovery by employing the same service provided by independently implemented agents). We can also model fault tolerance of a group of agents (for example, when they need to leave a location in emergency or when we need to conduct load balancing operation to avoid system degradation).

To automate system modelling, we are currently designing a number of fault tolerance patterns to help system developers introduce some common fault-tolerance techniques when modelling an agent system [11,12]. These techniques range from abstract system-level patterns to very specific agent-level patterns dealing with specific faults and focus on integrating fault tolerance in the specific modelling views.

Early on, we have extended the blackboard communication pattern [13] with nested scopes and exception propagation [14]. These two extensions are essentially the modelling and the implementation techniques aiming at representing recovery actions. In the implementation of fault tolerance, we extensively rely on the reactive agent architecture. This has two immediate benefits: its implementation style matches the event modelling style, captured by the event and functional model views; and recovery of multi-threaded agents becomes similar to that of the asynchronous reactive architecture.

In spite of some success in modelling different fault tolerance solutions for the AmI systems, we realise that our approach needs further work, in particular, in coherent modelling of fault tolerance represented in different model views. In the work we report here, in most cases we treat fault tolerance in different views as being orthogonal and non interfering, assuming that the erroneous state is always confined to one model view at a time: in this case, error recovery can be localised in this view. To address the more general cases where the same error affects several views or recovery from concurrent errors that need coordinated activities in several views, we will need to define common parts of the views and some rules of their sharing/transformation.

5 Case Study Scenarios

In our previous work, we implemented two scenarios within the ambient campus case study using the Cama framework as the core component of the applications [15,16,4]. The first scenario (*ambient lecture*) deals with the activities carried out by the teacher and the students during a lecture – such as questions and answers, and group work among the students – using various mobile devices (PDAs and smartphones). The second scenario (*presentation assistant*) covers the activities involved in giving and attending a presentation. The presenters uses a PDA to control the slides during their presentation and they may receive 'quiet' questions on the topic displayed on the slide from the audience. Each member of the audience will have the current slide displayed on his/her PDA, which also provides a feature to type in questions relevant to that slide.

In this section we discuss our work on a more challenging scenario which involves greater agent mobility as well as the use of the location specific services. Agents may move physically among multiple locations (rooms), and depending on the location, different services will be provided for them. In this work, we shift our focus from implementation to design and we use this scenario to validate our formal development approach.

In this scenario – we call it the *student induction assistant* scenario – we have new students visiting the university campus for the first time. They need to register to various university departments and services, which are spread on many locations on campus, but they do not want to spend too much time looking for offices and standing in queues. They much prefer spending their time getting to know other students and socialising. So they can delegate the registration process to their personalised software agent, which then visits virtual offices of various university departments and institutions, obtains the necessary information for the registration, and makes decisions based on the student's preferences. The agent also records pieces of information collected during this process so that the students can retrieve all the details about their registration.

Unfortunately, not all the registration stages can be handled automatically. Certain steps require personal involvement of the student, for example, signing paperwork in the financial department and manually handling the registration in some of the departments which do not provide fully-featured agents able to handle the registration automatically. To help the student to go through the rest of registration process, his/her software agent creates an optimal plan for visiting different university departments and even arranges appointments when needed.

Walking around on the university campus, these new students pass through *ambients* – special locations providing context-sensitive services (see Figure 2). An ambient has sensors detecting the presence of a student and a means of communicating to the student. An ambient gets additional information about students nearby by talking to their software agent. Ambients help students to navigate within the campus, provide information on campus events and activities, and assist them with the registration process. The ambient infrastructure can also be used to guide students to safety in case of emergency, such as fire.

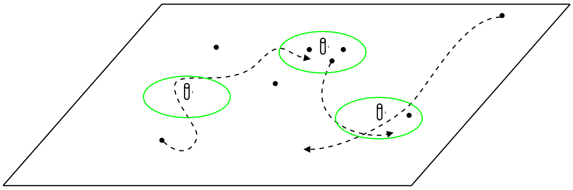


Fig. 2. *Student induction assistant* scenario: the dots represent free roaming student agents; the cylinders are static infrastructure agents (equipped with detection sensors); and the ovals represent *ambients* – areas where roaming agents can get connection and location-specific services.

5.1 Application of Our Approach to the Scenario

To proceed further, we need to agree on some major design principles, identify major challenges and outline the strategy for finding the solution. In order to understand the scenario better, we apply the *agent metaphor*. The agent metaphor is a way to reason about systems (not necessarily information systems) by decomposing it into agents and agent subsystems. In this paper, we use the term *agent* to refer to a component with an independent thread of control and state, and the term *agent system* to refer to a system of cooperative agents.

From agent systems' viewpoint, the scenario is composed of the following three major parts: physical university campus, virtual university campus and ambients. In the physical university campus, there are students and university employees. Virtual campus is populated with student agents and university agents. Ambients typically have a single controlling agent and a number of visiting agents. These systems are not isolated, they interact in a complex manner and information can flow from one part to another.

However, since we are building a distributed system, it is important to get an implementation as a set of independent but cooperative components (agents). To achieve this, we apply the following design patterns:

agent decomposition. During the design, we will gradually introduce more agents by replacing abstract agents with two or more concrete agents.

super agent. It is often hard to make a transition from an abstract agent to a set of autonomous agents. What before was a simple centralised algorithm in a set of agents must now be implemented in a distributed manner. To aid this transition, we use *super agent* abstraction, which controls some aspects of the behaviour of the associated agents. Super agent must be gradually removed during refinement as it is unimplementable.

scoping. Our system has three clearly distinguishable parts: physical campus, virtual campus and ambients. We want to isolate these subsystems as much as possible. To do this, we use the scoping mechanism, which temporarily isolates cooperating agents. This is a way to achieve the required system decomposition. The isolation properties of the scoping mechanism also make it possible to attempt autonomous recovery of a subsystem.

orthogonal composition. As mentioned above, the different parts of our scenario are actually interlinked in a complex manner. To model this connections, we use the *orthogonal composition* pattern. In orthogonal composition, two systems are connected by one or more shared agents. Hence, information from one system into another can flow only through the agent states. We will try to constrain this flow as much as possible in order to obtain a more robust system.

locations definition. To help students and student agents navigate within the physical campus and the virtual campus, we define location as places associated with a particular agent type.

decomposition into roles. The end result of system design is a set of agent roles. To obtain role specifications, we decompose scopes into a set of roles.

5.2 Formulating the Requirements

From the initial description of the scenario, we formulated a set of requirements that would assist us in implementing the student induction assistant system, in particular concerning the registration process. These requirements can also be found in the RODIN Deliverable D27 [17]. We divided the system requirements into the following categories:

ENV	Facts about the operating environment of the system.
DES	Early design decisions captured as requirements.
FUN	Requirements to the system functionality.
OPR	Requirements to the system behaviour.
SEC	Requirements related to the security properties of the system.

Top-Level Requirements. First we attempt a high-level description of the system. The description captures different aspects of the system: environment, some design decisions (dictated by the motivation for this case study), and few general functionality and security requirements.

FUN1	*The system helps new students to go through the registration process.*

DES1	*The system is composed of university campus, virtual campus and ambients.*

OPR1 | *A student must have a choice between automated and manual registration.*

OPR2 | *Malfunctioning or failure of the automated registration support should not prevent a student from manual registration.*

SEC1 | *The system should not disclose sensitive information about students.*

SEC2 | *The system must prevent malicious or unauthorised software to disguise itself as acting on behalf of a student or an employee.*

University. University campus forms the environment for the software-based registration process (Figure 3). The university campus is obviously not something that can be designed and implemented. However it is important to consider it in the development of the scenario as it provides an operating environment for the other two parts (virtual campus and ambients) which can be implemented in software and hardware.

ENV1 | *In university campus, students interact with university employees.*

ENV2 | *Students can freely move around while employees do not change their position.*

ENV3 | *Each university employee is permanently associated with a unique location.*

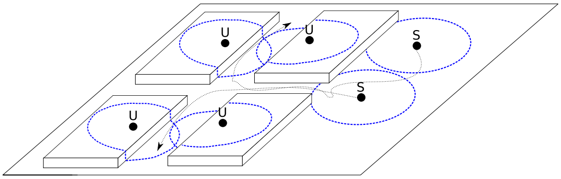


Fig. 3. University campus is modelled as a number of university employees (U) and students (S). Virtual campus has the same structure but is populated with student and university agents.

Virtual Campus. Virtual campus uses software-based solution to process student registration automatically. Its organisation is similar to that of a real campus.

DES2 | *Virtual campus is composed of university agents and student agents.*

DES3 | *In virtual campus, student agents can autonomously change their location.*

DES4 | *Each university agent is permanently associated with a unique location.*

Virtual campus is a meeting place for student agents and university agents. During registration, student agent talks to different university agents.

FUN2 | *Student agents and university agents can exchange information related to the registration process.*

Some registration steps require intervention from a student.

OPR3 | *A registration process may fail due to inability of a particular university agent to handle the registration.*

Before the registration process is initiated, a student agent has to go through several other stages. This results in a tree of dependencies. The root of the tree represents a successful registration and its leaves represent the registration stages without any prerequisites (see Figure 4). Student agent does not know about the tree structure and so it has to explore it dynamically. Reconstructing the tree for each agent makes the system more flexible and robust.

OPR4 | *Each registration stage has number of dependencies.*

DES5 | *Initially, student agent does not know the dependency tree.*

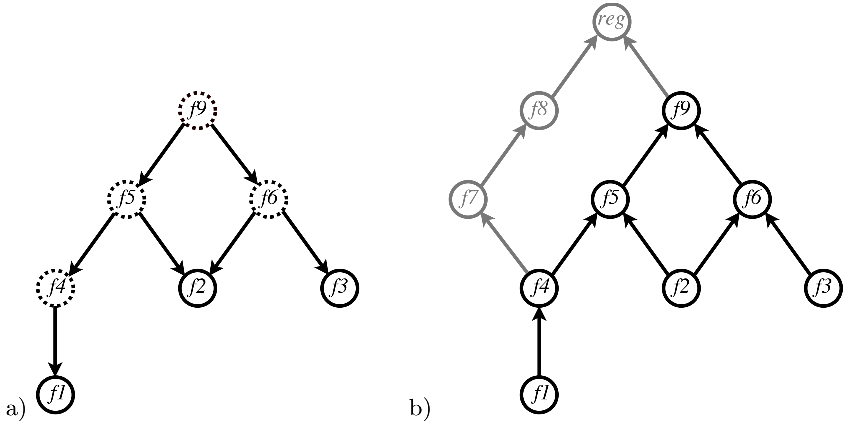


Fig. 4. a) Registration process starts from a random location (*f9* on the figure). The basic requirements *f1, f2, f3* are discovered by tracing back the requirements graph. b) Student agent attempts to do the registration by satisfying each known requirement. It does not yet know the full set of registration requirements (unknown steps are greyed). They are discovered during this process.

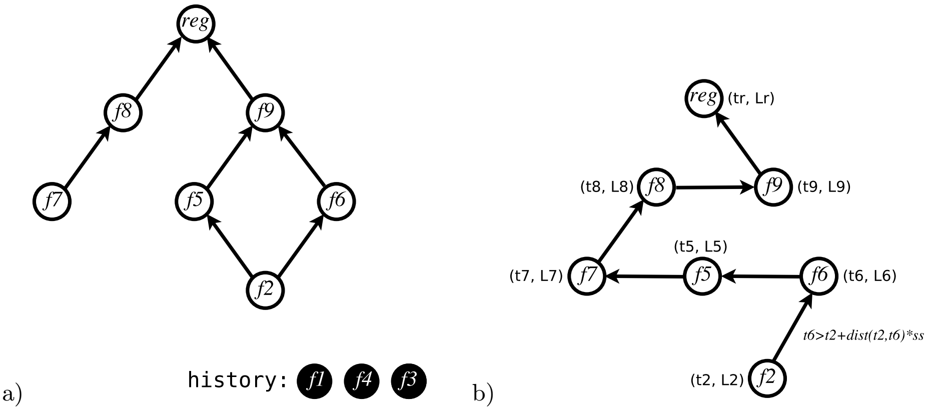


Fig. 5. a) During registration a student agent accumulates registration information. b) Itinerary for manual continuation of a registration is a path covering all the remaining registration graph nodes and satisfying a number of constrains. A node of the path is described by a pair containing when and where should go to resolve a given registration dependency.

DES6 | *Student agent autonomously constructs the dependency tree.*

Interacting with university agents, student agent records all the information related to the registration process. This information can be used to restart the registration process or to be passed to the student in order to do manual registration. In the latter case, student agent creates a schedule that helps a student to visit different university offices in the right order and at the right time (see Figure 5).

DES7 | *Student agent keeps a history of the registration process that can be used to restart the registration from the point of last completed registration step.*

DES8 | *Student agent can create an itinerary for a student to complete the registration manually.*

DES9 | *Itinerary must satisfy the registration dependencies.*

Ambient. As implied by the scenario, ambients provide services within a predefined physical location. By services we understand an interaction of an ambient with student's software. An interaction is triggered when a student enters a location associated with a given ambient.

OPR5 | *Ambients interact with student agents to assist with the registration process.*

FUN3 | *Ambient provides services by interacting with student software.*

FUN4 *Interaction with an ambient is triggered when a student enters a location associated with the ambient.*

FUN5 *Interaction with an ambient is terminated when a student leaves the location of the ambient.*

Positioning Service. For simplicity, we assume that ambient locations are discreet – a student is either within a location or outside of it – and do not change over time.

FUN6 *Ambient locations are discreet and static.*

Discovery of an ambient by a student (or vice versa) does not come for free. It is achieved using tiny mobile sensor platforms called *smart dust* [18]. Smart dust devices – also known as *motes* – has low-power, short-range radio capability, enabling them to communicate with other motes within range (Figure 6).

ENV4 *Ambients detect students nearby using the mote radio communication.*

Each student carries one such mote which broadcasts student's identity at certain intervals.

ENV5 *Each student carries a mote.*

FUN7 *Student mote broadcasts student id.*

Student motes' signals are sensed by ambients. Ambient agent is equipped with a mote radio receiver.

ENV6 *Each ambient is equipped with a mote radio receiver.*

When an ambient senses that a student mote is within range, it transmits this information to all other ambients.

FUN8 *Position of a student detected by an ambient is made available to all other ambients.*

We will rely on this functionality to implement recovery in emergency situations.

Student. Automated registration must be under the full control of a student. A student should be able to start, stop and inspect the current state of a registration.

FUN9 *Student starts and stops registration process.*

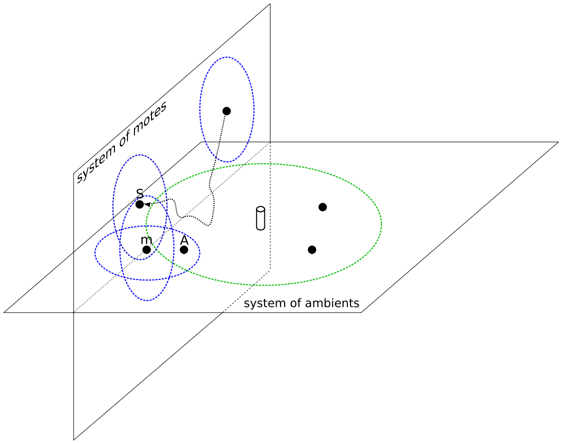


Fig. 6. Composition of motes and ambients system

FUN10 | *Student may enquire the current state of a registration while registration is in progress.*

FUN11 | *When registration is finished or interrupted, a student can access the recorded registration state.*

Student Agent. Student agent is a software unit assisting a student in registration.

FUN12 | *Student agent assists a student in manual registration by creating a schedule for visiting university employees.*

FUN13 | *Student agent records the state of registration process.*

Mobility. The scenario includes several types of mobility. There is physical mobility of computing platforms owned by students (e.g. mobile phones and PDAs). Students' agents can migrate to and from a virtual campus world. In this case, agent code and agent states are transferred to a new platform using code mobility. Finally, agents migrate within a virtual campus using virtual mobility.

Different styles of mobility have different requirements. Code mobility is a complex and fail-prone process: it is dangerous to have an agent separated from its state or having an agent with only partially available state or code. There is also a danger of an agent disappearing during the migration: the source of migration, believing that migration was successful, shuts down and removes the local agent copy, while the destination platform fails to initialise the agent due to transfer problems.

OPR6 | *Agent either migrates fully to a new platforms or is informed about inability to migrate and continues at a current platform.*

Physical mobility presents the problems of spontaneous context change. A student agent may be involved in a collaboration with an ambient when a student decides to walk away. Clearly, student behaviour cannot be restricted and such abrupt changes of context and disconnections must be accounted for during the design of agents and ambients.

OPR7 *Interaction between an ambient and a student agent can be interrupted at any moment.*

Virtual mobility is the simplest flavour of mobility as it does not involve any networking and nothing is actually moving in space. The only possible failure that can affect virtual migration is a failure or a shut-down of the hosting platform. However, such dramatic failure is unlikely to happen during an agent lifetime and thus we do not consider it at all in this document.

Fault-Tolerance. The system we are designing is a complex distributed system with a multitude of possible failure sources. In addition to traditional failures associated with networking, we have to account for failures related to environmental changes which are beyond the control of our system. Below is the list of faults we are going to address and which we believe covers the possible failures in our system:

- disconnections and lost messages:

  OPR8 *Agents must tolerate disconnections and message loss.*

- failure of ambients:

  OPR9 *Student agents must be able to autonomously recover from a terminal ambient failure.*

  also, since ambient services are not critical, it is better to avoid failing or misbehaving ambient:

  FUN14 *Student agent drops interaction with an ambient if it suspects that the ambient is malfunctioning.*

- failure of university agents. University agents are critical for the completion of the registration, so it is worth trying to recover cooperatively:

  OPR10 *Student and university agents cooperate to recover after failure.*

  It does not make sense to remain in virtual campus if one of the university agents is failing to interact:

  FUN15 *Student agent leaves virtual campus when it detects a failing university agent.*

- failure of student agents. Failure of a student agent may be detected by a university agent, ambience agent or student.

FUN16 | *University agent detecting student agent crash should attempt to notify the agent owner.*

And there is a possibility that a student suddenly terminates without leaving any notice. In this case, we rely on the student to detect this situation and possibly try again by sending another agent.

FUN17 | *Student should be able to restart registration process.*

5.3 Refinements

In this section, we demonstrate few initial development steps for the case study. These steps are done in a process-algebraic style, but at a later refinement, the development method changes into state-based modelling using Event-B. More details on our modelling approach can be found in [19].

To ensure interoperability among different agent types in our scenario and also to verify properties (such as eventual termination of the registration process), we use the combination of CSP process algebra [20], AgentB modelling (Event-B with some syntactic sugar – outlined in Section 3), and the Mobility Plugin [6]. The AgentB part of the design is responsible for modelling functional properties of the system; for the verification purposes, it is translatable into proper Event-B models. With the Mobility plugin, we are able to construct scenarios describing typical system configurations and verify properties related to system dynamics and termination. For example, we can model-check the migration algorithm described in Event-B to verify that the algorithm will never omit a location.

The whole development process is lengthy, so we only show some excerpts here.

Our system is concerned with the registration of a new student. At a very abstract level, the registration process is accomplished in one step:

$$\frac{S_0 \xrightarrow{\mathsf{REF\_PREFIX}} S_1 \text{ sat. } \mathsf{FUN1}}{register.}$$

From the description of the system, we know that the registration process is made of an automatic or manual parts, either of which properly implements the registration process

$$\frac{S_1 \xrightarrow{\mathsf{REF\_ICH}} S_2 \text{ sat. } \mathsf{OPR1}}{\begin{array}{l|l} \begin{array}{l} auto \mapsto register \\ manual \mapsto register \end{array} & auto. \sqcap manual. \end{array}}$$

(steps S_3 - S_6 omitted)

At this stage, we are ready to speak about roles of agents implementing the system. We introduce two roles: student (s), representing a human operator using a PDA; and agent (a) which for now stands for all kinds of software in our system.

$$\begin{array}{l|l}
\multicolumn{2}{l}{S_6 \xrightarrow{\mathsf{REF\_ROLE}} S_7 \text{ sat. } \mathsf{DES1}} \\ \hline
 & (s'\,\mathit{send} \rightarrow a'\,\mathit{move}.; a'\,\mathit{communicate}.; a'\,\mathit{automatic}.) \sqcap (\\
s, a \in \rho S_7 & \quad (\mathit{auto\_fail} \rightarrow \mathit{manual\_anew}.) \sqcap \\
 & \quad (\mathit{auto\_part} \rightarrow \mathit{manual\_cont}.))
\end{array}$$

In the next model, we focus on a sub-model of the system which represents virtual campus (vc) activities: the *communicate* process. The process is refined into a loop where a student agent visits different university agents and speaks to them. The loop alternates between termination (**break**) and the registration process:

$$\begin{array}{l|l}
\multicolumn{2}{l}{\mathit{communicate}. \text{ from } S_7 \xrightarrow{\mathsf{REF\_LOOP}} S_1^{vc} \text{ sat. } \mathsf{FUN3}} \\ \hline
\mathit{done} \mapsto \mathit{communicate} & a'^{+}\big(\mathit{auto\_register}. \sqcap \mathbf{break}\big); a'\,\mathit{done}.
\end{array}$$

(steps S_2^{vc} - S_6^{vc} omitted)

By adding more details on the interactions between the student and the university agents, we arrive to the following model. The model implements a simple request-reply protocol where the university agent's role is given through a choice from a number of replies. Event *reply_ok* is used when registration is successful, event *reply_docs* indicates that there are missing documents and that student agent must visit some other virtual offices before registration can be completed. In the case when the registration is not possible without the student being present in person, the *reply_pers* reply is used.

$$\begin{array}{l}
\qquad\qquad S_6^{vc} \xrightarrow{\mathsf{REF\_DCPL}} S_7^{vc} \\ \hline
sa'^{+}\begin{pmatrix}
sa'\,\mathit{migrate} \rightarrow sa'\,\mathit{ask}.; (\\
\quad (ua'\,\mathit{reply\_ok}.; sa'\,\mathit{save\_repl}.)ua' \sqcap \\
\quad (ua'\,\mathit{reply\_docs}.; sa'\,\mathit{doclist}.)ua' \sqcap \\
\quad (ua'\,\mathit{reply\_pers}.; sa'\,\mathit{do\_pers} \rightarrow sa'\mathbf{break}))ua' \sqcap \\
\quad (ua'\,\mathit{fail}.; sa'\,\mathit{leave\_vc} \rightarrow sa'\mathbf{break}))
\end{pmatrix}; sa'\,\mathit{done}.
\end{array}$$

(steps S_8^{vc} and S_9^{vc} omitted)

This model prepares the transition to a state-based model with completely decoupled agent roles:

$$\begin{array}{l}
\qquad\qquad\qquad S_9^{vc} \rightarrow S_{10}^{vc} \\ \hline
([\psi_1] \sqcap \mathbf{skip}) \parallel \\
{}^{+}\big(\psi_1 \rightarrow (sa'(\mathit{migrate} \rightarrow \mathit{ask}.); [\varphi_1])\big) \parallel \\
{}^{+}\big(\varphi_1 \rightarrow ua'((\mathit{reply\_ok}; [\varphi_2]) \sqcap (\mathit{reply\_docs}; [\varphi_3]) \sqcap (\mathit{reply\_pers}; [\varphi_4])) \sqcap (\mathit{fail}; [\varphi_5]))\big) \parallel \\
{}^{+}\big(\varphi_2 \rightarrow sa'\,\mathit{save\_repl}.; [\psi_1]\big) \parallel \\
{}^{+}\big(\varphi_3 \rightarrow sa'\,\mathit{doclist}.; [\psi_1]\big) \parallel \\
{}^{+}\big(\varphi_4 \rightarrow sa'\,\mathit{do\_pers}.\big) \parallel \\
{}^{+}\big(\varphi_5 \rightarrow sa'\,\mathit{leave\_vc}.\big)
\end{array}$$

The next two refinement steps add further details to the behavioural model. Refinement step 10 introduces functional model with the modelling decisions taken by the student and the university agents. Further refinements introduce details on how a university agent decides what documents to ask and when the registration process is complete. The student agent keeps track of all visited locations and is able to remember branching points in order not to visit the same university agents twice.

Further details on the refinement process can be found in [17,21,19].

5.4 Implemented System and Screenshots

To implement the ambients, we incorporate smart dust devices or motes [18] into the scenario. In particular, we use off-the-shelf MPR2400 MICAz motes (see Figure 7) from Crossbow Technology [22]. These motes communicate with each other using Zigbee radio, and by customising the transmit power of the radio (in this case, reducing the radio range to around 3-5 meters), we can use them as a localisation sensor. This enables us to deliver location-specific information and services to the users.

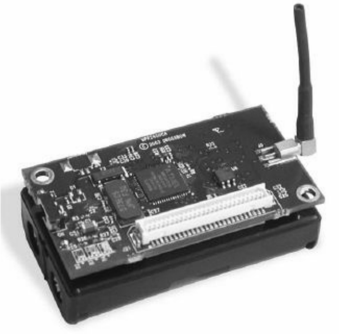
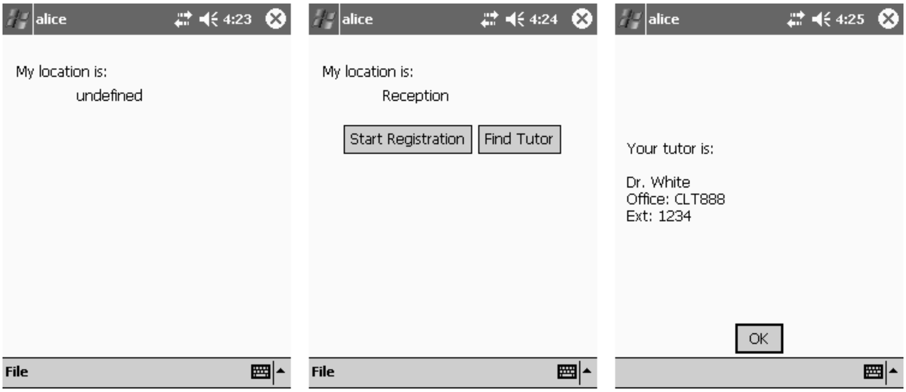


Fig. 7. MICAz mote used for localisation sensor

Each user carries a mote (programmed with a unique identification number, so that the mote acts as a badge - sort of speak), as well as a PDA as an interaction device. Each room is equipped with a smart dust base station (receiver), which is connected to a controller application. The latter uses the CAMA middleware [3] to communicate with the PDAs through Wi-Fi. When a user enters a particular room, his/her PDA shows the relevant information and/or services available for that room.

A set of rooms can be prepared to be smart dust aware. This can include the reception office, in which the users (i.e. students) can start the registration process or can find out who their tutor is. Figure 8 shows the screen captures of the PDA used by the student ("Alice"). The picture on the left shows the situation where Alice is not in any location that supports the scenario. When she enters the reception room, her PDA adjusts its location and displays the services available in that room (as can be seen in the picture in the middle). In this example, Alice opts to find out who her tutor is (the picture on the right).



My location is:
undefined

My location is:
Reception

Your tutor is:

Dr. White
Office: CLT888
Ext: 1234

Fig. 8. Screen captures of the registration assistant scenario

6 Conclusion

This paper provides an outline of the work that we had carried out in developing fault-tolerant ambient applications. We introduce a theoretical approach called AgentB, that is based on the modelling database concept, and is composed of several simple modelling methods focusing on various aspects of the system. These modelling techniques allow us to validate the formal development, and to model and build fault tolerant ambient applications. The approach has been demonstrated through a rigorous development of an ambient campus *student induction assistant* scenario, starting from the definition of a set of requirements, the modelling and refinement processes, and finally, the implementation of the system. We developed an agent-based system implementing this scenario, using the Cama framework and the middleware [2] that we have previously developed.

Acknowledgements

This work is supported by the FP6 IST RODIN STREP Project [1], the FP7 ICT DEPLOY Integrated Project [23], and the EPSRC/UK TrAmS Platform Grant [24].

References

1. Rodin: Rigorous Open Development Environment for Complex Systems. IST FP6 STREP project (last accessed August 6, 2008), `http://rodin.cs.ncl.ac.uk/`
2. Arief, B., Iliasov, A., Romanovsky, A.: On developing open mobile fault tolerant agent systems. In: Choren, R., Garcia, A., Giese, H., Leung, H.-f., Lucena, C., Romanovsky, A. (eds.) SELMAS 2007. LNCS, vol. 4408, pp. 21–40. Springer, Heidelberg (2007)
3. Iliasov, A.: Implementation of Cama Middleware (last accessed August 6, 2008), `http://sourceforge.net/projects/cama`

4. Iliasov, A., Romanovsky, A., Arief, B., Laibinis, L., Troubitsyna, E.: On Rigorous Design and Implementation of Fault Tolerant Ambient Systems. Technical report, CS-TR-993, School of Computing Science, Newcastle University (December 2006)
5. Metayer, C., Abrial, J.R., Voisin, L.: Rodin Deliverable 3.2: Event-B Language. Technical report, Project IST-511599, School of Computing Science, University of Newcastle (2005)
6. Iliasov, A., Khomenko, V., Koutny, M., Niaouris, A., Romanovsky, A.: Mobile B Systems. In: Proceedings of Workshop on Methods, Models and Tools for Fault Tolerance at IFM 2007, CS-TR 1032, Newcastle University (2007)
7. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (2005)
8. Abrial, J.R., Schuman, S.A., Meyer, B.: A specification language. In: McNaughten, R., McKeag, R. (eds.) On the Construction of Programs. Cambridge University Press, Cambridge (1980)
9. Randell, B.: System Structure for Software Fault Tolerance. IEEE Trans. Software Eng. 1(2), 221–232 (1975)
10. Plasil, F., Holub, V.: Exceptions in Component Interaction Protocols - Necessity. In: Architecting Systems with Trustworthy Components, pp. 227–244 (2004)
11. Iliasov, A.: Refinement patterns for rapid development of dependable systems. In: Proceedings of Engineering Fault Tolerant Systems Workshop et ESEC/FSE. ACM Digital Library, Croatia (2007)
12. Iliasov, A., Romanovsky, A.: Refinement Patterns for Fault Tolerant Systems. In: The Seventh European Dependable Computing Conference (EDCC-7) (Technical paper). IEEE CS, Los Alamitos (2008)
13. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System Of Patterns. John Wiley & Sons Ltd., West Sussex (1996)
14. Iliasov, A., Romanovsky, A.: Structured coordination spaces for fault tolerant mobile agents. In: Dony, C., Knudsen, J.L., Romanovsky, A., Tripathi, A.R. (eds.) Advanced Topics in Exception Handling Techniques. LNCS, vol. 4119, pp. 181–199. Springer, Heidelberg (2006)
15. Arief, B., Coleman, J., Hall, A., Hilton, A., Iliasov, A., Johnson, I., Jones, C., Laibinis, L., Leppanen, S., Oliver, I., Romanovsky, A., Snook, C., Troubitsyna, E., Ziegler, J.: Rodin Deliverable D4: Traceable Requirements Document for Case Studies. Technical report, Project IST-511599, School of Computing Science, University of Newcastle (2005)
16. Troubitsyna, E. (ed.): Rodin Deliverable D8: Initial Report on Case Study Development. Project IST-511599, School of Computing Science, University of Newcastle (2005)
17. Troubitsyna, E. (ed.): Rodin Deliverable D27: Case Study Demonstrators. Project IST-511599, School of Computing Science, University of Newcastle (2007)
18. Smartdust: Wikipedia definition (last accessed August 6, 2008), `http://en.wikipedia.org/wiki/Smartdust`
19. Iliasov, A., Koutny, M.: A Method and Tool for Design of Multi-Agent Systems. In: Pahl, C. (ed.) Proceedings of Software Engineering (SE 2008). ACTA Press (2008)
20. Hoare, C.A.R.: Communicating Sequential Processes. Communications of the ACM 21(8), 666–677 (1978)
21. Troubitsyna, E. (ed.): Rodin Deliverable D26: Final Report on Case Study Development. Project IST-511599, School of Computing Science, University of Newcastle (2007)

22. CrossbowTechnology: MPR/MIB User's Manual (last accessed August 6, 2008), http://www.xbow.com/Support/Support_pdf_files/MPR-MIB_Series_Users_Manu%al.pdf
23. Deploy: Industrial Deployment of System Engineering Methods Providing High Dependability and Productivity. IST FP7 IP project (last accessed August 6, 2008), http://www.deploy-project.eu/
24. TrAmS: Trustworthy Ambient Systems Platform Grant (last accessed August 6, 2008), http://www.cs.ncl.ac.uk/research/current%20projects?pid=223/

Using Inherent Service Redundancy and Diversity to Ensure Web Services Dependability

Anatoliy Gorbenko[1], Vyacheslav Kharchenko[1],
and Alexander Romanovsky[2]

[1] Department of Computer Systems and Networks, National Aerospace University,
Kharkiv, Ukraine
A.Gorbenko@csac.khai.edu, V.Kharchenko@khai.edu
[2] School of Computing Science, Newcastle University, Newcastle upon Tyne, UK
Alexander.Romanovsky@newcastle.ac.uk

Abstract. Achieving high dependability of Service-Oriented Architecture (SOA) is crucial for a number of emerging and existing critical domains, such as telecommunication, Grid, e-science, e-business, etc. One of the possible ways to improve this dependability is by employing service redundancy and diversity represented by a number of component web services with the identical or similar functionality at each level of the composite system hierarchy during service composition. Such redundancy can clearly improve web service reliability (trustworthiness) and availability. However to apply this approach we need to solve a number of problems. The paper proposes several solutions for ensuring dependable services composition when using the inherent service redundancy and diversity. We discuss several composition models reflecting different dependability objectives (enhancement of service availability, responsiveness or trustworthiness), invocation strategies of redundant services (sequential or simultaneous) and procedures of responses adjudication.

1 Introduction

The Web Services (WS) architecture [1] based on the SOAP, WSDL and UDDI specifications is rapidly becoming a de facto standard technology for organization of global distributed computing and achieving interoperability between different software applications running on various platforms. It is now extensively used in developing numerous business-critical applications for banking, auctions, Internet shopping, hotel/car/flight/train reservation and booking, e-business, e-science, Grid, etc. That is why analysis and dependability ensuring of this architecture are emerging areas of research and development [1–3]. The WS architecture is in effect a further step in the evolution of the well-known component-based system development with off-the-shelf (OTS) components. The main advances enabling this architecture have been made by the standardisation of the integration process, by a set of interrelated standards such as SOAP, WSDL, UDDI, etc.

Web Services are autonomous systems, the ready-made OTS components belonging to different organizations, without any general or centralised control, that may

M. Butler et al. (Eds.): Fault Tolerance, LNCS 5454, pp. 324–341, 2009.
© Springer-Verlag Berlin Heidelberg 2009

change their behaviour on the fly. This architecture brings a number of benefits to the users but at the same time poses many challenges to researchers and developers. By their very nature Web Services are black boxes, as neither source code, nor specification, nor information about deployment environment are available; the only known information about them is their interfaces. Moreover, their quality is not completely known and they may not provide sufficient quality of service; it is often safe to treat them as "dirty" boxes, assuming that they always have bugs, do not fit enough, have poor specification and documentation. WSs are heterogeneous, as they might be developed following different standards, fault assumptions, and different conventions and may use different technologies. Finally, their construction and composition are complicated by the fact that the Internet is a poor communication medium (has low quality, not predictable).

The main motivation for our work is the fact that ensuring and assessing dependability of complex service-oriented systems is complicated when these systems are dynamically built or when their components (i.e. Web Services) are dynamically replaced by the new ones with the same (or similar) functionality but unknown dependability characteristics. The lack of evidence about the characteristics of the communication medium, components used in the composition and their possible dependencies makes it extremely difficult to achieve and predict SOA dependability which can vary over a wide range in a random manner. Therefore, users cannot be confident in availability, trustworthiness, reasonable response time and others dependability characteristics. Dealing with such uncertainty, mainly coming from the SOA nature, is the main challenge.

This uncertainty should be treated as the threat (similar and in addition to the commonly known faults, errors and failures). The paper discusses fault-tolerance solutions for building dependable service-oriented systems out of undependable Web Service components, which have changeable functional sets and uncertain dependability characteristics, making use of natural redundancy and diversity inherent to such systems.

In the paper we analyse different dependability-oriented composition models of Web Services and also propose solutions guaranteeing that the overall dependability (availability, correctness and responsiveness) of the composite system is improving.

2 Web Services Redundancy and Diversity

SOA supports construction of the globally distributed massive-scale systems with growing number of services. This makes it unique in allowing access to a number of services with identical or similar functionalities, provided by different vendors and deployed on different platforms all over the Internet. In other words, SOA possesses the inherent redundancy and diversity of the existing Web Services [17]. We should use this fact to build dependable Service-Oriented Systems out of undependable Web Services. Table 1 shows several examples of the existing alternative (redundant) stock quotes and currency exchange Web Services (see [18] for a more detailed discussion of these examples). In [12] the authors present a practical experience report on dependability monitoring of three diverse Bioinformatics Web Services performing

Table 1. An example of alternative (redundant) Web Services

Alternative (redundant) Stock Quotes Web Services
stock_wsx.GetQuote: http://www.webservicex.com/stockquote.asmx?WSDL
stock_gama.GetLatestStockDailyValue: http://www.gama-system.com/webservices/stockquotes.asmx?wsdl
stock_xmethods.getQuote: http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl
stock_sm.GetStockQuotes: http://www.swanandmokashi.com/HomePage/WebServices/StockQuotes.asmx?WSDL
Alternative (redundant) Currency Exchange Web Services
currency_exchange.getRate: http://www.xmethods.net/sd/CurrencyExchangeService.wsdl
currency_convert.ConversionRate: http://www.webservicex.com/CurrencyConvertor.asmx?wsdl

similar BLAST[1] function. A mediator approach (set of intermediate monitoring services) was used to monitor WS dependability metadata and provide it for users. This work was a motivation for us to show i) that there are multiple similar WSs, and ii) that they can be used simultaneously to achieve better dependability.

72-87% of the faults in open-source software are independent of the operating environment (i.e. faults in application software) and are hence permanent [20]. Half of the remaining faults are environment depended and permanent. And only 5-14% of the faults are environment depended caused by transient conditions. Hence, software diversity can be an efficient method of fault-tolerance provisioning and decreasing common mode failures caused by software faults [21, 22]. SOA supports inherent diversity at the different level:

1. ***Application diversity:*** i) development diversity of application software (different developers, languages, implementation technologies and tools, etc); ii) data diversity (diversity of data used and data sources); iii) Service diversity (diversity of 'physical' resources and services, for example, flights available, hotel rooms, etc).
2. ***Deployment diversity.*** Different service providers can use diverse deployment environments (different hardware platform, operating systems, web and application servers, DBMS, etc.).
3. ***Spatial (geographical) diversity.*** Redundant Web Services can be dispersed all over the Internet and different service vendors can use different Internet Service Providers.

To build dependable Service-Oriented Systems, developers (systems integrators) and end users should be able to choose and use the most dependable components (i.e. Web Services) from the existing ones of similar functionality but diverse nature [23]. Other approach we are discussing in this paper is using all available services simultaneously with the purpose to improve overall system dependability.

[1] http://www.ncbi.nlm.nih.gov/blast/html/BLASThomehelp.html

3 Web Services Dependability

Dependability of a computing system is its ability to timely deliver service that can justifiably be trusted [24]. According to this definition we need to deal with the following dependability attributes, which are relevant to Web Services, and which can be easily measured during WS invocations: (i) availability; (ii) reliability; and (iii) response time (performance). There are several other attributes, describing Quality of Service (QoS), service level agreements (SLAs) and dependability, including authentication, confidentiality, non-repudiation, service cost, etc. [25], but we do not deal with them in this paper.

Service availability. The degree to which a service is operational and accessible when it is required for use determines service's availability. Availability of a system is a measure of the delivery of correct service with respect to the alternation of correct and incorrect service [24]. It can be defined by a ratio of the system's uptime to all execution time (including downtime). Unfortunately, such technique can be hardly applied for determining the availability of Web Services in a loosely coupled SOA. More adequate, the availability of a Web Service most likely can be defined by the ratio of the total number of service invocations to the number of events when the service was unavailable (i.e. an exception "HTTP Status-Code (404): Not Found" was caught by client). The easier recovery action here is simple retry.

Service reliability. System reliability can be measured in terms of probability of failure-free operation, mean time between failures (MTBF) or failure rate. Reliability assessment of Web Services is complicated, taking into account the fact that service invocation rate can vary in a wide range for different services and services customers. Another problem here is that Web Service returns errors of two main types [9]:

1. Evident erroneous response which results in exception message. The probability of such errors can be measured by the proportion of the total number of service invocation number of exception messages received (apart from exception "HTTP Status-Code (404): Not Found" that indicate about service unavailability). If such error occurs, user could retry the same service latter or (most likely) invoke an alternative one.
2. Non-evident erroneous response. It can be present in a form of incorrect data or calculation errors which do not entail immediate exception. The last type of error is the most dangerous and can lead to unexpected program behaviour and unpredicted consequences, and, as a result, service discredit. Detection of such errors is possible by comparing service response with response from another diverse service.

Therefore, the key problem services developers and users are faced with is enhancing the service trustworthiness (correctness) rather then decreasing probability of exception (i.e. evident error occurrence).

Service performance (response time). The service response time can be divided into (i) network delay time, (ii) connection waiting time and (iii) execution time. The execution time is the duration of performing service functionality, the connection waiting time is the time during request waits in application server's queue, and, finally, network delay time is the delay of request transmissions between service consumer and provider.

The network delay time can be hardly predicted due to the uncertain network fluctuations whereas connection waiting time and execution time depend on service load and throughput.

4 Web Services Composition

Web service composition is currently an active area of research, with many languages being proposed by academic and industrial research groups. IBM Web Service Flow Language (WSFL) [4] and Microsoft's XLANG [5] were two of the earliest languages to define standards for Web services composition. Both languages extended W3C Web Service Description Language (WSDL) [6], which is the standard language for describing the syntactic aspects of a Web service. Business Process Execution Language for Web Services (BPEL4WS) [7] is a recently proposed specification that represents the merging of WSFL and XLANG. BPEL4WS combines the graph oriented process representation of WSFL and the structural construct based processes of XLANG into a unified standard for Web services composition.

In addition to these commercial XML-based standards, there have been work on a unique Web service composition language called Web Ontology Language for Services (www.daml.org/services) OWL-S (previously known as DAML-S) [8], which provides a richer description of Web service compositions by specifying its semantics.

In our work we focus on the general patterns (types) of the WS composition and identify two typical blueprints of composing WSs: i) "vertical" composition for functionality extension, and ii) "horizontal" composition for dependability improvement (dependability-oriented composition).

The first type of service composition ("vertical") is used for building the WorkFlow (WF) of the systems and is already supported by BPEL, BPML, XPDL, JPDL and other WF languages. The second one ("horizontal") deals with a set of redundant (and possibly diverse) Web Services with identical or similar functionality. Rather than investigate fixed redundancy schemes [28] (like two-out-of-three or three-out-of-five) in this paper we discuss flexible patterns improving various dependability attributes (availability, trustworthiness or responsibility) taken separately.

Bellow we show some illustrative examples of the two types of WSs composition and discuss the way in which the *horizontal* composition improves dependability of SOA. We use the web-based *Travel Agency* as an example of Web Services composition, this example has been extensively used by other researchers [3, 26, 27].

4.1 *Vertical* Composition for Functionality Extension

The "vertical" composition (Fig. 1-a) extends the Web Services functionality. A new Composite Web Service is composed out of several particular services which provide different functions. For example, the *Travel Agency* (TA) Service can be composed of a number of services such as *Flight Service*, *Car Rental Service*, *Hotel Service*, etc.

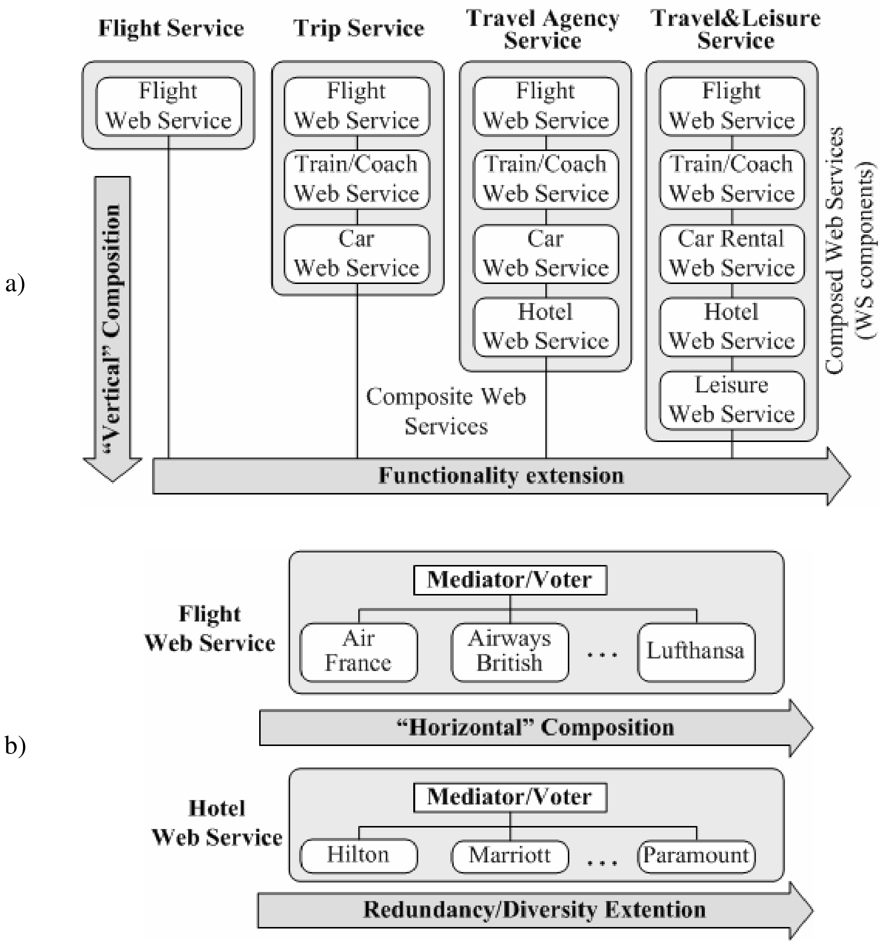


Fig. 1. Web Services Composition: a) "vertical"; b) "horizontal"

The main invocation parameters for such a composite *TA Service* are trip endpoint (country and city), dates (and time) of arrival and departure, user details and preferences. The *TA Service* then invokes corresponding services which books flight/train/coach tickets, hotel room, rents a car, etc.

The Composite Web Service can invoke a set of target (composed) services simultaneously to reduce the mean execution time or sequentially (if execution of one service depends on the result of another one).

If some of the services fail or cannot satisfy the user's request (for example, when there are no flights available for specified dates) all other services have to be rolled back and their results should be cancelled.

To improve dependability of such composite system various means of fault-tolerance and error recovery should be applied, including redundancy, exception handling, forward error recovery, etc.

4.2 *Horizontal* Composition for Dependability Improvement

The “horizontal” composition (Fig. 1-b) uses several alternative (diverse) Web Services with the identical or similar functionality, or several operational releases of the same service. Such kind of redundancy based on inherent service diversity improves service availability and reliability (correctness and trustworthiness) of Web Service composition.

Architecture with the “horizontal” (dependability-oriented) composition includes a “Mediator” component, which adjudicates the responses from all diverse Web Services and returns an adjudicated response to the consumer. In the simplest case the “Mediator” is a *voter* (i.e. performs majority voting using the responses from redundant Web Services). It can be also programmed to perform more complex operation like aggregation, or provide the best choice according to selection criterions specified by user (for example, highest possible exchange rate or minimal asked quotation).

Papers [10-13] introduce special components (called “Service Resolver”, “Proxy”, “Service Container” or “Wrapper”) with the similar functionality.

5 Middleware-Based Architecture Supporting Dependability-Oriented Composition

5.1 Patterns of Dependability-Oriented Composition

In [9] we proposed an architecture which uses a dedicated middleware for a managed dependable upgrade and the “horizontal” composition of Web Services. The middleware runs several redundant (diverse) Web Services (Fig. 2). It intercepts the user’s requests coming through the WS interface, relays them to all the redundant services and collects the responses. It is also responsible for dependability measurement and publishing the confidence in dependability [9] associated with each service.

The architecture proposed supports several composition models meeting different dependability objectives (such as enhancement of service availability, responsiveness or trustworthiness), various strategies for invoking redundant services (sequential or simultaneous) and procedures for response adjudication. These models form *patterns* of dependability-oriented composition. The basic ones are:

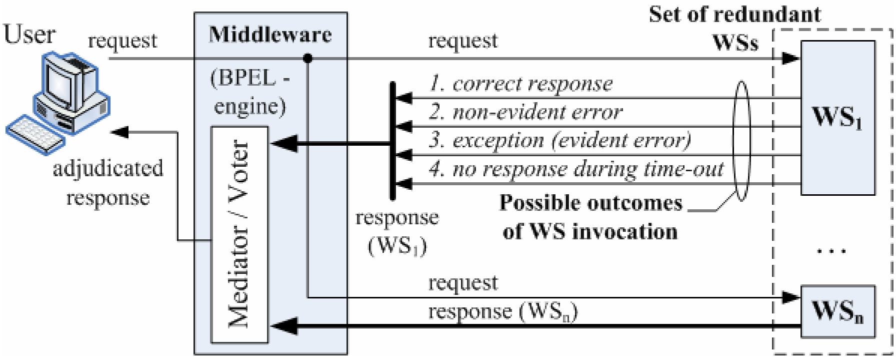
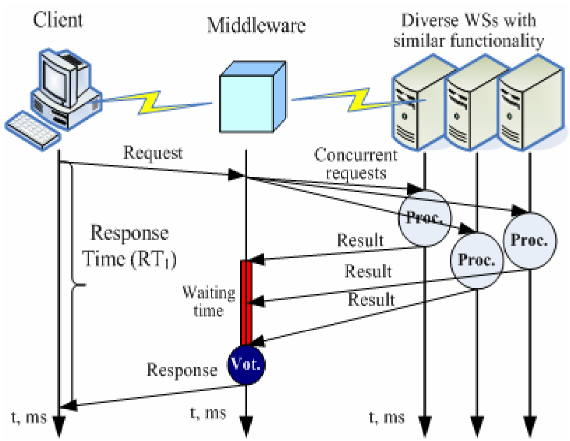


Fig. 2. Architecture of dependability-oriented composition

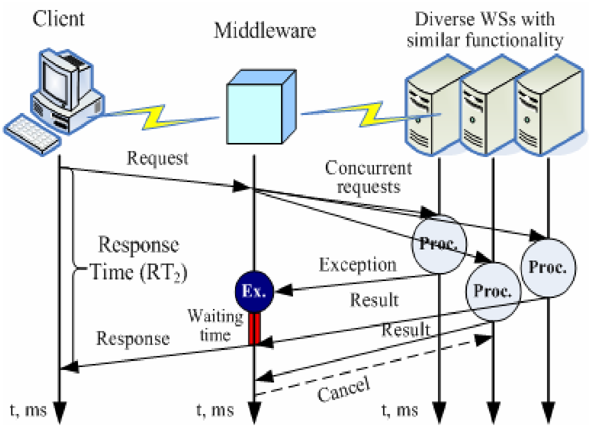


Simulation results:

	Case1	Case2	Case3
P(cr)	0.966	0.936	0.928
P(ne)	0.010	0.048	0.013
P(ex)	0.016	0.008	0.032
P(ua)	0.008	0.008	0.027
MRT	397.60	397.60	431.40

Vot. - voting; **Ex.** - exception; **Com** - comparison; **Proc.** - processing; 🟥 - waiting time

Fig. 3. Simulation results of the *Reliable concurrent execution* pattern



Simulation results:

	Case1	Case2	Case3
P(cr)	0.961	0.915	0.922
P(ne)	0.003	0.017	0.005
P(ex)	0.029	0.060	0.046
P(ua)	0.008	0.008	0.027
MRT	141.10	133.85	163.82

Fig. 4. Simulation results of the *Fast concurrent execution* pattern

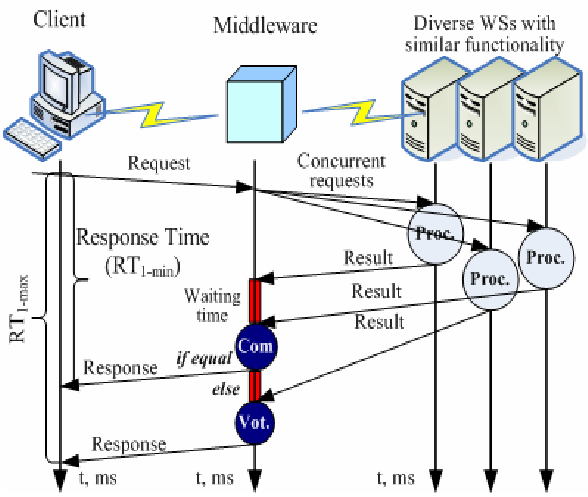
1. *Reliable concurrent execution* for trustworthiness improvement (Fig. 3). All available redundant (diverse) WSs are invoked concurrently and their responses are used by the middleware to produce an adjudicated response to the consumer of the WS (i.e. voting procedure). Initial values of measures *P(cr), P(ne), P(ua), P(ex)* and mean response time (MRT) used in three different simulation cases are discussed in section 5.2.
2. *Fast concurrent execution* for responsiveness improvement (Fig. 4). All available redundant (diverse) WSs are invoked concurrently and the fastest non-evidently incorrect response is returned to the service consumer.
3. *Adaptive concurrent execution* (Fig. 5). All (or some of) redundant (diverse) WSs are executed concurrently. The middleware is configured to wait for up to a certain

number of responses to be collected from the redundant services, but no longer than a pre-defined timeout.

4. *Sequential execution* for minimal service loading (Fig. 6). The subsequent redundant WS is only invoked if the response received from the previous one is evidently incorrect (i.e. exception).

5.2 Simulation

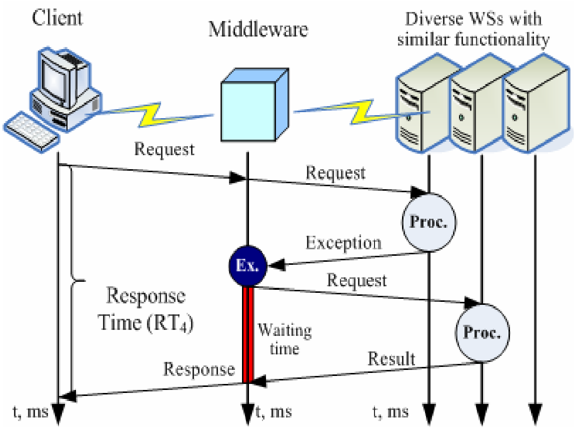
Effectiveness of the different composition models depends on the probability of service unavailability, occurrence of evident (exceptions raised) and non-evident (erroneous results returned) failures. The probability of service unavailability due to



Simulation results:

	Case1	Case2	Case3
P(cr)	0.966	0.936	0.928
P(ne)	0.010	0.048	0.013
P(ex)	0.016	0.008	0.032
P(ua)	0.008	0.008	0.027
MRT	289.72	289.08	330.52

Fig. 5. Simulation results of the *Adaptive concurrent execution* pattern



Simulation results:

	Case1	Case2	Case3
P(cr)	0.961	0.915	0.922
P(ne)	0.003	0.017	0.005
P(ex)	0.029	0.060	0.046
P(ua)	0.008	0.008	0.027
MRT	283.99	266.25	387.86

Fig. 6. Simulation results of the *Sequential execution* pattern

different reasons (service overload, network failures, and congestions) is several orders greater than probability of failure occurrence. Moreover, different exceptions arise during service invocation more frequently than non-evident failures occur.

To analyse the effectiveness of the proposed patterns we developed a simulation model running in the MATLAB 6.0 environment. It used the following initial values chosen using the real-life statistics [19]:

	Case 1	*Case 2*	*Case 3*
P(correct response), P(cr)	*0.70*	*0.70*	*0.60*
P(non-evident error), P(ne)	*0.01*	*0.05*	*0.01*
P(exception), P(ex)	*0.09*	*0.05*	*0.09*
P(service unavailability), P(ua)	*0.20*	*0.20*	*0.30*

Each redundant Web Service was modelled as a black box that is assumed to fail independently of all the others but with the same probability [26]. The first two cases (cases 1 and 2) correspond to services that have the same availability but different probabilities of evident and non-evident error occurrence (we assume more trusted services in case 1). The third one simulates trusted service with worse availability as compared to Case 1 (possibly, due to narrow network bandwidth or frequent congestions). During simulation we also set *Mean Response Time* (MRT) which equals 200 ms and *Maximum Waiting Time* (time-out) which equals 500 ms.

The simulation results of each proposed patterns of dependability-oriented composition are shown at the Figures 3 – 6 respectively. Figure 7 gives a summary of all simulation cases.

A practical application of the horizontal composition requires developing new workflow patterns and languages constructs, supporting different composition models and procedures of multiple results resolving and voting.

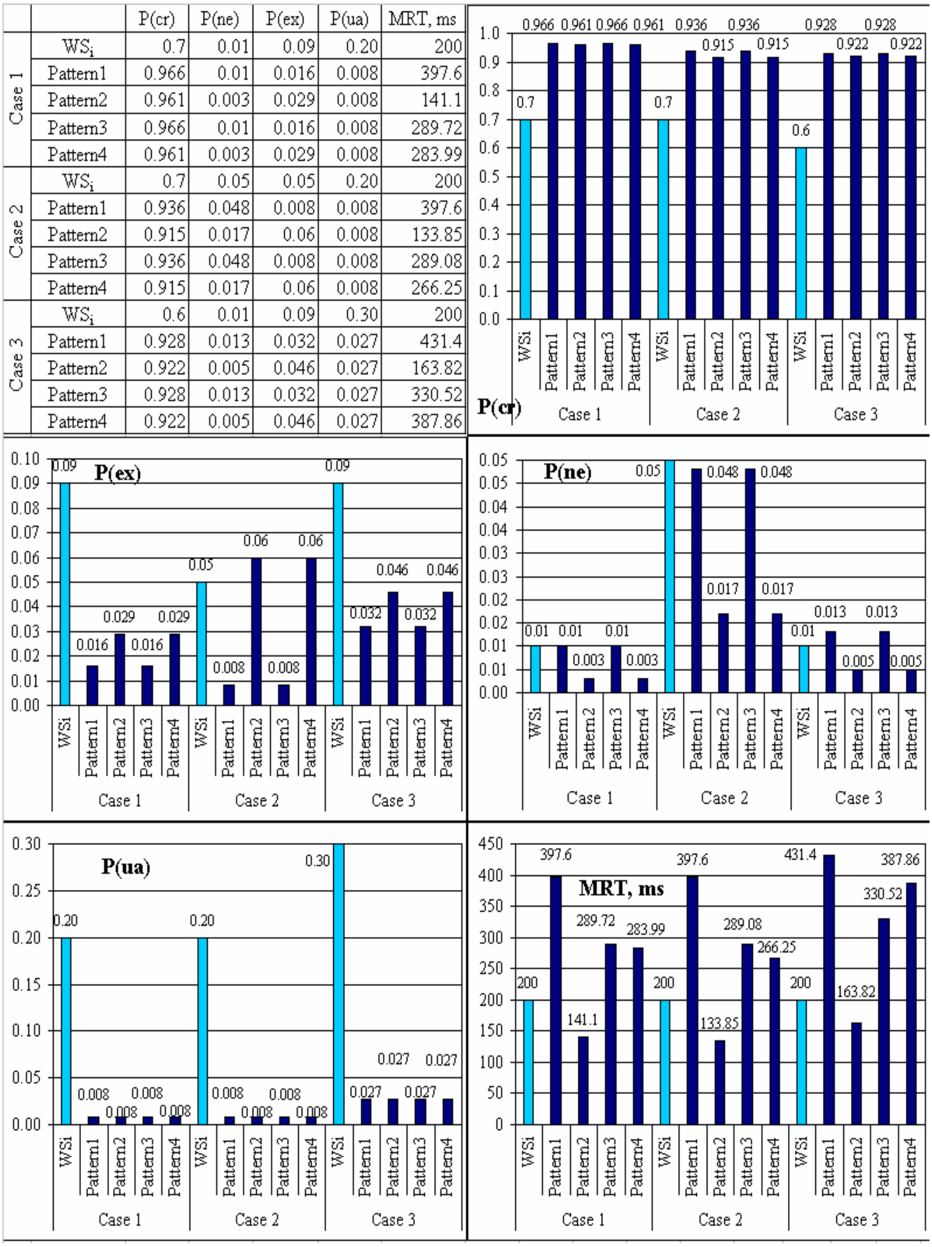
6 Implementation

6.1 Work-Flow Patterns Supporting Web Services Composition

The workflow patterns capture typical control flow dependencies encountered during workflow modelling. There are more then 20 typical patterns used for description of different workflow constructions of "vertical" composition [14]. The basic ones are: *'Sequence'*, *'Exclusive Choice'*, *'Simple Merge'*, *'Parallel Split'*, *'Synchronization'*, *'Discriminator'*, *'Regular Cycle'*, etc.

Each WF language describes a set of elements (activities) used for implementing different WF patterns. For example, BPEL4WS defines both primitive (*'invoke'*, *'receive'*, *'reply'*, *'wait'*, *'assign'*, *'throw'*, *'terminate'*, *'empty'*) and structured (*'sequence'*, *'switch'*, *'while'*, *'flow'*, *'pick'*, *'scope'*) activities.

The first ones are used for intercommunication and invoking operations on some web service. Structured activities present of complex workflow structures and can be nested and combined in arbitrary ways.



		P(cr)	P(ne)	P(ex)	P(ua)	MRT, ms
Case 1	WS_i	0.7	0.01	0.09	0.20	200
	Pattern1	0.966	0.01	0.016	0.008	397.6
	Pattern2	0.961	0.003	0.029	0.008	141.1
	Pattern3	0.966	0.01	0.016	0.008	289.72
	Pattern4	0.961	0.003	0.029	0.008	283.99
Case 2	WS_i	0.7	0.05	0.05	0.20	200
	Pattern1	0.936	0.048	0.008	0.008	397.6
	Pattern2	0.915	0.017	0.06	0.008	133.85
	Pattern3	0.936	0.048	0.008	0.008	289.08
	Pattern4	0.915	0.017	0.06	0.008	266.25
Case 3	WS_i	0.6	0.01	0.09	0.30	200
	Pattern1	0.928	0.013	0.032	0.027	431.4
	Pattern2	0.922	0.005	0.046	0.027	163.82
	Pattern3	0.928	0.013	0.032	0.027	330.52
	Pattern4	0.922	0.005	0.046	0.027	387.86

Fig. 7. Summary of simulation results (WSi – particular Web Service; Pattern 1 – the *Reliable concurrent execution* pattern; Pattern 2 – the *Fast concurrent execution* pattern; Pattern 3 – the *Adaptive concurrent execution* pattern; Pattern 4 – Simulation results of the *Sequential execution* pattern)

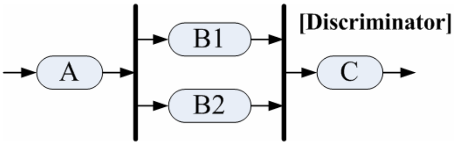


Fig. 8. Workflow pattern *Discriminator*

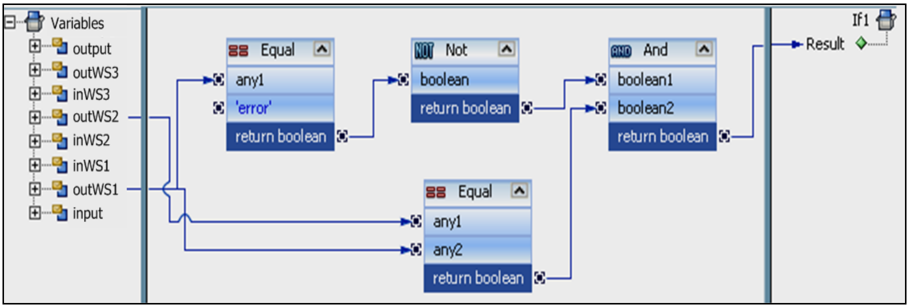
In fact, the only one of the basic WF patterns *Discriminator* fits for implementing the *Fast Concurrent Execution* pattern providing maximum responsiveness. Discriminator (see Fig. 8) is a point in the workflow process that waits for one of the incoming branches to complete before activating the subsequent activity. The first one that comes up with the result should proceed the workflow. The other results will be ignored.

However, only BPML `<all>` and BPEL `<pick>` activities support such WF pattern [15, 16] (Fig. 9). To support dependability-oriented composition the additional WF patterns need to be developed and implemented for different WF languages.

The new activities allowing a business process to support *redundancy* and perform *voting* procedure should also be developed. This is a motivation of our further work.

```
<process name=″PatternDiscriminator″>
  <sequence> <context>
      <signal name=″completed_B″/>
      <process name=″B1″>
         …
        <raise signal=″completed_B″/>
      </process>
      <process name=″B2″>
         …
        <raise signal=″ completed_B″/>
      </process>
    </context>
    <action name=″A″ …>
      …
    </action>
    <all>
      <spawn process=″B1″/>
      <spawn process=″B2″/>
    </all>
    <synch signal=″completed_B″/>
    <action name=″C″ …>
      …
    </action>
  </sequence> </process>
```

Fig. 9. BPML implementation of the *Discriminator* pattern



```
<if name="If1">
    <condition> ( not( ( $outWS1 = 'error' ) )
     and ( $outWS1 = $outWS2 ) ) </condition>
    <sequence name="Sequence4">
       <assign name="Assign2">
          <copy>
             <from variable="outWS1"/>
             <to variable="output"/>
          </copy>
       </assign>
    </sequence>
    <else>
       <sequence name="Sequence4">
          <invoke name="Invoke3" partnerLink="WS3" .../>
  ...
```

Fig. 10. WS-BPEL *If* statement implementing responses matching (graphic and text notion)

6.2 Testbed Workflows

The patterns of dependability-oriented composition discussed above have been implemented as a set of testbed workflows (see, for example, Fig. 11) by using a graphics-based BpelModule which is a part of IDE NetBeans 6.0[2].

We used WS-BPEL 2.0[3] specification, which introduces two constructs specifically for extensions (‘*extensionActivity*’ and ‘*extensionAssignActivity*’). It also has improved fault handling and process termination features.

New fault handlers having ‘*catch*’, ‘*catchAll*’, ‘*compensate*’, ‘*throw*’ and ‘*rethrow*’ constructs as well as ‘*terminationHandler*’ and ‘*exitOnStandardFault*’ activities were added in WS-BPEL.

A business logic supporting voting and comparison procedures within particular pattern is implementing by using *if* statements.

Fig. 10 gives an example of how to compare the responses from first two services invoked at the first step of the *Adaptive concurrent execution* pattern (Fig. 5 and 11). If the results are equal and are not ‘exception’ (standard error) then the agreed result can be returned to the user, otherwise the third service should be invoked to perform voting procedure.

[2] www.netbeans.org/community/releases/60/

[3] www.oasis-open.org/committees/wsbpel/

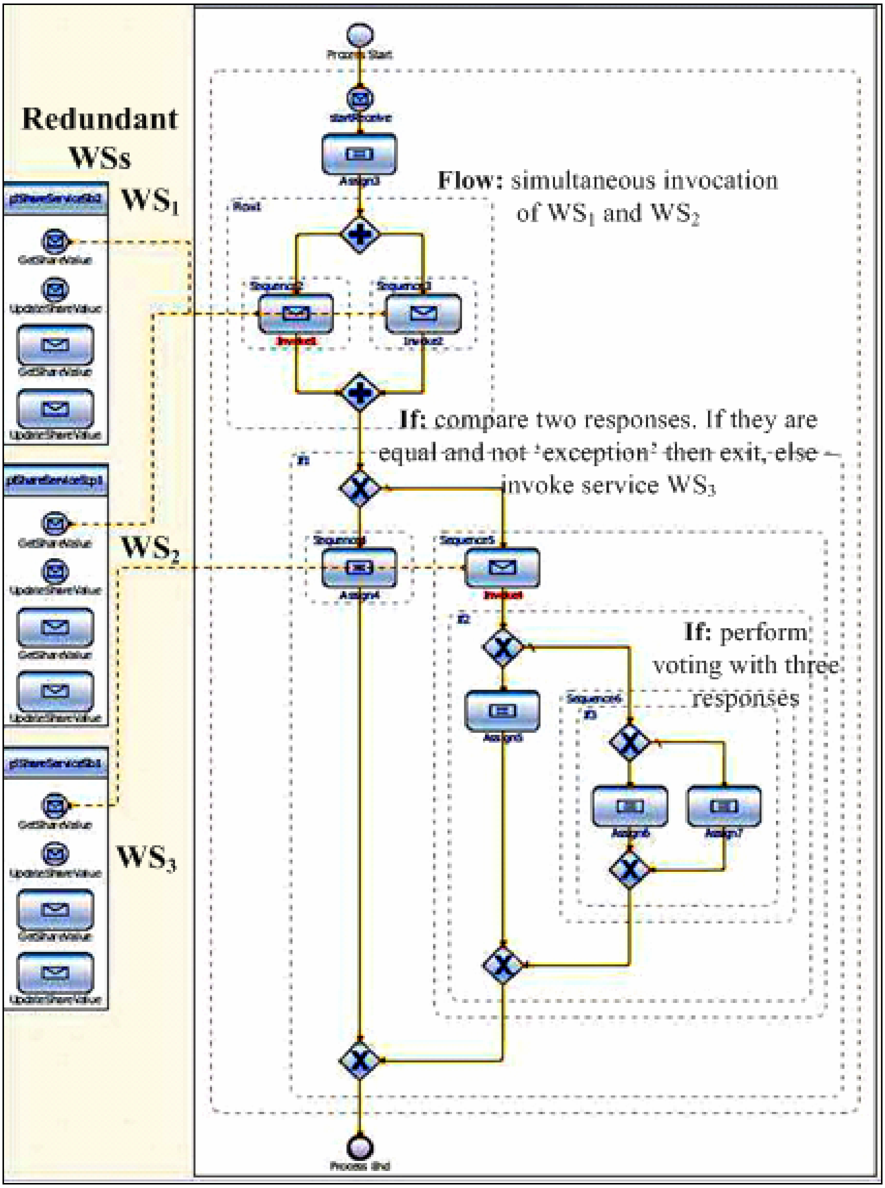


Fig. 11. Screen shot of WS-BPEL-workflow implementing the *Adaptive concurrent execution* pattern

7 Conclusions and Future Researches

We have addressed different models of a Web Services composition which extend functionality (“vertical” composition) or improve dependability (“horizontal” composition).

“Vertical” composition uses redundancy based on natural diversity of existing Web Services with the identical or similar functionality deployed by third parties. We discussed middleware-based architecture that provides dependability-oriented composition of Web Services. For the best result middleware has to implement on-line monitoring and dependability control.

“Horizontal” composition, which uses redundancy in combination with diversity, is one of the promising means of enhancing service availability and providing fault-tolerance. Different patterns are applicable here. As it is shown from simulation and experimentation, all models of dependability-oriented composition significantly improve service availability (as it was expected) and probability of correct response (see Table 2).

The *Adaptive concurrent execution* and *Reliable concurrent execution* patterns give maximal reliability (probability of correct response) and minimal probability of exception at the expense of performance deterioration. However the *Adaptive concurrent execution* pattern provides better reliability-to-response-time ratio.

The *Fast concurrent execution* and *Sequential execution* patterns improve service correctness (decrease probability of non-evident error). Besides, the first one provides minimal response time (less than mean response time of each particular WS). An unexpected result was that the *Sequential execution* pattern improves reliability and correctness without performing unnecessary services invocation and, at the same time, provides rather good response time.

Finally, a more complex composition model combining the “vertical” and “horizontal” compositions is also possible. It supports two boundary architectures:

1. *Multilevel mediation* (Fig. 12-a) with co-ordination at each level of functional composition.
2. *One-level mediation* (Fig. 12-b) with co-ordination at only the top level.

A number of intermediate architectures are also possible. However, questions like “How many horizontal composition levels will provide the maximal improvement?”, “When mediator (voter) should be placed?” are yet unsolved and are objectives of future researches.

Another question is how to assess and take into account actual services diversity. Because it is obvious that different Web Services can refer to the same ‘physical’ services or resources like it is shown in the Fig, 12-b where two independent *TA Services* (v.1 and v.2) use the same *Car Rental Service* ‘Hertz’.

Table 2. Effectiveness of different patterns of dependability-oriented composition

№	Patterns of dependability-oriented composition	*max. probability of correct response*	*min. probability of non-evident error*	*min. probability of exception*	*min. probability of service unavailability*	*min. response time*
1	Reliable concurrent execution	++		+	+	--
2	**Fast concurrent execution**	**+**	**+**		**+**	**+**
3	Adaptive concurrent execution	+	+		+	-
4	**Sequential execution**	**++**		**+**	**+**	**-**

(‘+’ – *advantage;* ‘-’ – *disadvantage*)

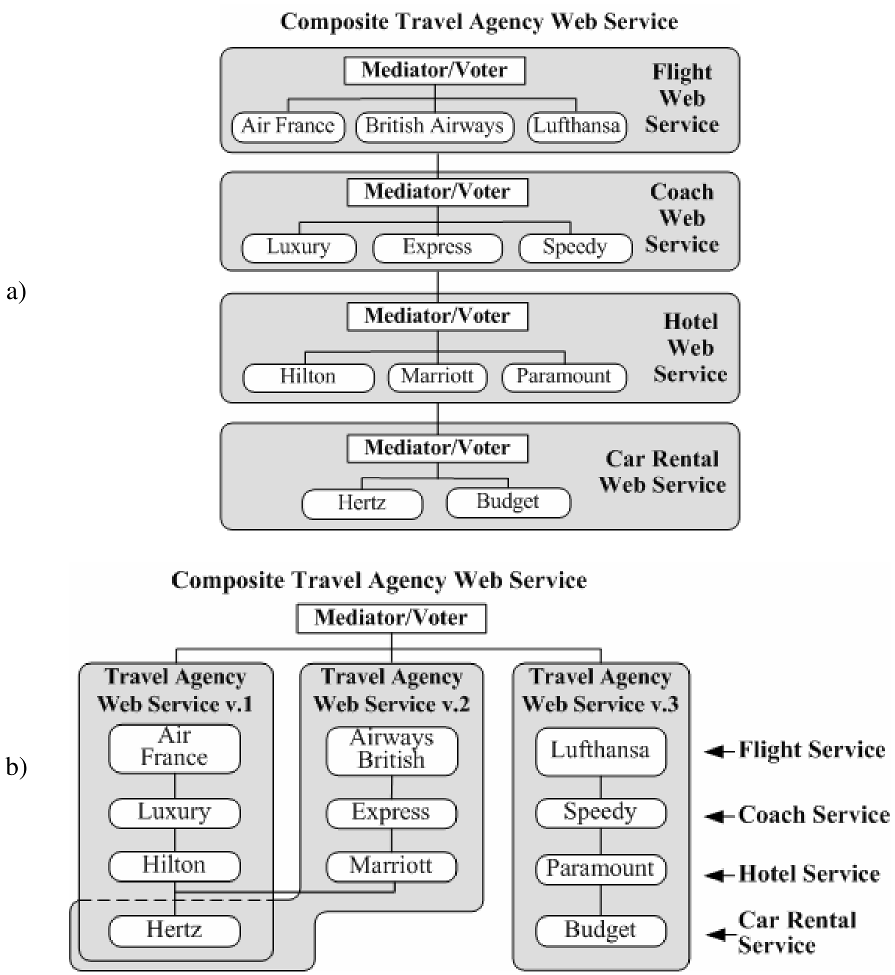


Fig. 12. Combined "vertical and horizontal" Web Service composition using *multilevel* (a) and *one-level* (b) mediation

Applying in practice techniques of the "horizontal" composition and other means of improving SOA dependability requires developing new workflow patterns and implementing them in different WF languages.

In our future work we are going to use WS-BPEL 2.0, which is a new version of popular language widely-used in industry for the specification of business processes and business interaction protocols. Of a particular interest to us is its support for extensibility by allowing namespace-qualified attributes to appear in any standard element and by allowing new user-specified activities to be defined by using '*extensionActivity*' and '*extensionAssignActivity*' constructions.

Acknowledgments. Alexander Romanovsky is partially supported by the EC ICT DEPLOY project.

References

1. W3C Working Group. Web Services Architecture (2004), `http://www.w3.org/TR/ws-arch/`
2. Ferguson, D.F., Storey, T., Lovering, B., Shewchuk, J.: Secure, Reliable, Transacted Web Services: Architecture and Composition. Microsoft and IBM Technical Report (2003), `http://www-106.ibm.com/developerworks/webservices/library/ws-securtrans`
3. Tartanoglu, F., Issarny, V., Romanovsky, A., Levy, N.: Dependability in the Web Service Architecture. In: Architecting Dependable Systems, pp. 89–108. Springer, Heidelberg (2003)
4. Leymann, F.: Web Services Flow Language. Technical report, IBM (2001)
5. Thatte, S.: XLANG: Web Services for Business Process Design. Technical report, Microsoft (2001)
6. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: WSDL: Web services description language (2001), `http://www.w3.org/TR/wsdl`
7. Andrews, T., Cubera, F., Dholakia, H.: Business Process Execution Language for Web Services Version 1.1. OASIS (2003), `http://ifr.sap.com/bpel4ws`
8. Ankolekar, et al.: Ontology Web Language for Services (OWL-S) (2002), `http://www.daml.org/services`
9. Gorbenko, A., Kharchenko, V., Popov, P., Romanovsky, A.: Dependable composite web services with components upgraded online. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems III. LNCS, vol. 3549, pp. 92–121. Springer, Heidelberg (2005)
10. Hall, S., Dobson, G., Sommerville, I.: A Container-Based Approach to Fault Tolerance in Service-Oriented Architectures (2005), `http://digs.sourceforge.net/papers/2005-icse-paper.pdf`
11. Maheshwari, P., Erradi, A.: Architectural Styles for Reliable and Manageable Web Services (2005), `http://mercury.it.swin.edu.au/ctg/AWSA05/Papers/erradi.pdf`
12. Chen, Y., Romanovsky, A., Li, P.: Web Services Dependability and Performance Monitoring. In: Proc. 21st Annual UK Performance Engineering Workshop, UKPEW 2005 (2005), `http://www.staff.ncl.ac.uk/nigel.thomas/UKPEW2005/ukpew-proceedings.pdf`
13. Townend, P., Groth, P., Xu, J.: A Provenance-Aware Weighted Fault Tolerance Scheme for Service-Based Applications. In: Proc. of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (2005)
14. Wohed, P., van der Aalst, W.M.P., Dumas, M., ter Hofstede, A.H.M.: Pattern-Based Analysis of BPEL4WS. QUT Technical report, FIT-TR-2002-04, Queensland University of Technology, Brisbane, Australia (2002), `http://is.tm.tue.nl/staff/wvdaalst/publications/p175.pdf`
15. van der Aalst, W.M.P., Dumas, M., ter Hofstede, A.H.M., Wohed, P.: Pattern-Based Analysis of BPML (and WSCI). QUT Technical report, FIT-TR-2002-05, Queensland University of Technology, Brisbane, Australia (2002), `http://is.tm.tue.nl/research/patterns/download/qut_bpml_rep.pdf`

16. Wohed, P., van der Aalst, W.M.P., Dumas, M., ter Hofstede, A.H.M.: Analysis of web services composition languages: The case of BPEL4WS. In: Song, I.-Y., Liddle, S.W., Ling, T.-W., Scheuermann, P. (eds.) ER 2003. LNCS, vol. 2813, pp. 200–215. Springer, Heidelberg (2003)
17. Gorbenko, A., Kharchenko, V., Romanovsky, A.: Vertical and Horizontal Composition in Service-Oriented Architecture. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 139–147. Springer, Heidelberg (2007)
18. Ernst, M.D., Lencevicius, R., Perkins, J.H.: Detection of Web Service substitutability and composability. In: Proc. International Workshop on Web Services Modeling and Testing (WS-MaTe 2006), pp. 123–135 (2006)
19. Chen, Y., Romanovsky, A.: Improving the Dependability of Web Services Integration. IT Professional: Technology Solutions for the Enterprise. IEEE Computer Society, issue, pp. 20–26 (January/February 2008)
20. Chandra, S., Chen, P.M.: Whither Generic Recovery From Application Faults? A Fault Study using Open-Source Software. In: Proc. Int. Conf. on Dependable Systems and Networks, pp. 97–106 (2000)
21. Deswarte, Y., Kanoun, K., Laprie, J.-C.: Diversity against Accidental and Deliberate Faults Computer Security. In: Dependability and Assurance: From Needs to Solutions. IEEE Computer Society Press, Washington (1998)
22. Lyu, M.R. (ed.): Handbook of Software Reliability Engineering, 805 p. McGraw-Hill Company, New York (1996)
23. Wang, Y., Vassileva, J.: Toward Trust and Reputation Based Web Service Selection: A Survey. In: Proc. International Transactions on Systems Science and Applications (ITSSA) Journal, special Issue on New tendencies on Web Services and Multi-agent Systems (WS-MAS), vol 3(2) (2007)
24. Avizienis, J.-C., Laprie, B., Randell, C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing 1(1), 11–33 (2004)
25. Yang, S., Lan, B., Chung, J.-Y.: Analysis of QoS Aware Web Services. In: Proc. International Computer Symposium on Web Technologies and Information Security Workshop (ICS) (2006)
26. Kaâniche, M., Kanoun, K., Martinello, M.: A User-Perceived Availability Evaluation of a Web Based Travel Agency. In: Proc. International Conference on Dependable Systems and Networks (DSN 2003), pp. 709–718 (2003)
27. Thomas, A., Venter, L.: Propagating Trust In The Web Services Framework. In: Proc. Information Security South Africa Conference (ISSA 2004), http://icsa.cs.up.ac.za/issa/2004/Proceedings/Full/012.pdf
28. Pat, P.W., Chan, M., Lyu, R., Malek, M.: Making Services Fault Tolerant. In: Proc. 3rd International Service Availability Symposium (ISAS 2006) (2006), http://www.cse.cuhk.edu.hk/~lyu/paper_pdf/ISAS06.pdf

Author Index