Lecture Notes in Computer Science 5525

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

Betty H.C. Cheng Rogério de Lemos
Holger Giese Paola Inverardi Jeff Magee (Eds.)

Software Engineering for Self-Adaptive Systems

Springer

Volume Editors

Betty H.C. Cheng
Michigan State University, Department of Computer Science and Engineering
3115 Engineering Building, East Lansing, MI 48824-1226, USA
E-mail: chengb@cse.msu.edu

Rogério de Lemos
University of Kent, Computing Laboratory
Canterbury, Kent CT2 7NF, UK
E-mail: r.delemos@kent.ac.uk

Holger Giese
Hasso Plattner Institute for Software Systems Engineering
Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany
E-mail: holger.giese@hpi.uni-potsdam.de

Paola Inverardi
Università dell'Aquila, Dipartimento di Informatica
67100 L'Aquila, Italy
E-mail: inverard@di.univaq.it

Jeff Magee
Imperial College, Department of Computing
180 Queen's Gate, London SW7 2BZ, UK
E-mail: j.magee@imperial.ac.uk

Library of Congress Control Number: 2009928525

CR Subject Classification (1998): D.2, D.3, F.1.1, I.2.2

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-642-02160-3 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-02160-2 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12686551 06/3180 5 4 3 2 1 0

Preface

The complexity of current software-based systems has led the software engineering community to look for inspiration in diverse related fields (e.g., robotics and control theory) as well as other areas (e.g., biology) to find innovative approaches for building, running, and managing software systems and services. Therefore, *self-adaptation* – systems that are able to adjust their behavior at run-time in response to their perception of the environment and the system itself – has become a hot topic within the software engineering community.

This book and the roadmap paper that is included here are two key outcomes from the Dagstuhl Seminar 08031 on "Software Engineering for Self-Adaptive Systems" that took place in January 2008. In addition to the roadmap paper, this book includes invited papers from recognized experts in the area that describe the current state of the art in the field, and papers that provide an insight into the key features of self-adaptive systems and how these should be designed. All the papers were peer-reviewed, with the exception of the roadmap paper, which was based on the discussion held at the Dagstuhl Seminar and put together by several of its participants. The book consists of four parts: "Research Roadmap," "Architecture-Based Self-Adaptation," "Context-Aware and Model-Driven Self-Adaptation," and "Self-Healing."

The first part entitled "Research Roadmap" includes the roadmap paper on research challenges for the area of software engineering for self-adaptive systems as well as the two papers that are extended versions of two self-adaptation views that were presented in the roadmap paper.

With the contribution of several authors—participants of the Dagstuhl Seminar on Software Engineering for Self-Adaptive Systems—the roadmap paper, which is entitled "Software Engineering for Self-Adaptive Systems: A Research Roadmap" summarizes the state of the art and identifies critical challenges for the systematic software engineering of self-adaptive systems.

The second paper of this part, authored by J. Andersson, R. de Lemos, S. Malek and D. Weyns, entitled "Modelling Dimensions for Self-Adaptive Systems" proposes a classification of modelling dimensions for self-adaptive software systems, which are key aspects that characterize self-adaptation. The identified modelling dimensions are illustrated by applying them to several application scenarios.

The third paper "Engineering Self-Adaptive Systems Through Feedback Loops," which is authored by Y. Brun, G. Di Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Muller, M. Pezzè and M. Shaw, promotes, in the design of self-adaptive systems, the use of feedback loops as first-class entities. The paper argues that feedback loops are essential for understanding all types of self-adaptive systems, and identifies critical challenges that must be

addressed to enable systematic and well-organized engineering of self-adaptive and self-managing software systems.

Part two of the book entitled "Architecture-Based Self-Adaptation" consists of three papers describing solutions in which architectures take a central role in the development of self-adaptive software systems.

The first paper by S.-W. Cheng, V. V. Poladian, D. Garlan and B. Schmerl, entitled "Improving Architecture-Based Self-Adaptation Through Resource Prediction," discusses how self-adaptation using architectural models can be improved by adopting an anticipatory approach in which predictions are used to inform adaptation strategies rather than operate in a purely reactive way. It is demonstrated that predictions can be incorporated into an architecture-based adaptation framework showing the resulting benefits.

J.C. Georgas and R.N. Taylor in the paper "Policy-Based Architectural Adaptation Management: Robotics Domain Case Studies" continue previous work in which the authors developed notations and tools that support the design and development of policy- and architecture-based self-adaptive systems that are modular and have the ability to change the specifications of the adaptation policy during system run-time. This paper assesses the feasibility of integrating those notations and tools with the robotics domain, develops novel self-adaptive capabilities for robotic systems, and identifies difficulties for such integration.

The last paper of this part, authored by W. Heaven, D. Sykes, J. Magee and J. Kramer and entitled "A Case Study in Goal-Driven Architectural Adaptation," presents an approach to constructing autonomous systems that synthesize tasks from high-level goals. In order to execute these tasks reliably in a changing environment, the software architecture of the system is adapted. The applicability of the proposed approach is demonstrated with a case study involving mobile robots.

Part three of the book is "Context-Aware and Model-Driven Self-Adaptation," and includes five papers centered around context-aware self-adaptability and how model-driven approaches can be applied in the development of self-adaptive software systems.

O. Nierstrasz, M. Denker and L. Renggli are the authors of "Model-Centric, Context-Aware Software Adaptation," the first paper of this part. This paper makes the case for model-centric and context-aware software adaptation, and shows through concrete examples how these design principles work at the level of application interface, programming language and run-time. The paper also discusses how the presence of sufficiently high-level models at run-time can enable very dynamic forms of context-dependent software adaptation. The authors also present a research agenda for model-centric development that supports dynamic software adaptation and evolution.

The second paper, entitled "Modeling of Context-Aware Self-Adaptive Applications in Ubiquitous and Service-Oriented Environments" written by K. Geihs, R. Reichle, M. Wagner and M. Ullah Khan, presents a modelling approach for the integration of service-based adaptation in a planning framework for compositional adaptation of context-aware applications. Based on semantic descriptions

that are associated to variation points in the component framework, the proposed modelling framework provides a harmonized view on QoS-properties of external discoverable services and conventional context properties of component-based applications.

The third paper "MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments," authored by R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli and U. Scholz, introduces an extension of the MUSIC component-based planning framework. This extension optimizes the overall utility of applications by plug-in interchangeable components and services when there are unexpected changes of the service provider landscape in a ubiquitous context. The planning framework is outlined and a motivating scenario is presented.

Paper four of this part, "Using Architecture Models to Support the Generation and Operation of Component-Based Adaptive Systems," authored by N. Bencomo and G. Blair, presents an approach that uses architectural models for supporting the generation and operation of component-based adaptive systems. The proposed approach supports the specification and validation of models based on abstractions of architectural or structural variability as well as environment and context variability.

Finally, the paper written by V. Grassi, R. Mirandola and E. Randazzo, and entitled "Model-Driven Assessment of QoS-Aware Self-Adaptation," presents an approach for supporting the QoS assessment of self-adaptable systems, which extends an intermediate modelling language for capturing the core features of a dynamically adaptable architecture model from a performance/dependability viewpoint. A model transformation chain is outlined that maps a "design oriented" model to an "analysis oriented" model for permitting the application of a suitable analysis methodology.

Part four of the book is "Self-Healing," and contains two papers related to error detection and system recovery.

The first paper in this part is authored by M. Pezzè and J. Wuttke with the title "Automatic Generation of Runtime Failure Detectors from Property Templates." The authors propose property templates to link requirements and design, and to generate automatically assertions using a model-based specification language. It targets self-healing software and permits detecting failures at low-cost at run-time while providing high detection precision and enough information about the detected failures to enable automatic healing actions.

The final paper of this part "Filtered Cartesian Flattening and Microrebooting to Build Enterprise Applications with Self-adaptive Healing," written by J. White, B. Dougherty, H.D. Strowd and D.C. Schmidt, considers the development of enterprise applications that can self-adapt to tolerate component failures. The paper describes a microrebooting technique based on feature models and a heuristic algorithm for deriving new configurations, as well as a container component that crashes the faulty subsystem and reboots the new configuration. For feature selection, the approach uses Filtered Cartesian Flattening and mul-

tidimensional multiple-choice knapsack heuristic algorithms in order to reduce self-healing time.

Although the self-adaptability of systems has been studied in a wide range of disciplines, from biology to robotics, only recently has the software engineering community recognized its key role in enabling the development of future software systems that are able to self-adapt to changes that may occur in the system, its requirements, or the environment in which it is deployed. In our understanding, this volume is one of the first books containing a collection of papers that looks specifically into the current state of the art in the field, describes a wide range of approaches coming from different strands of software engineering, and presents future challenges facing this always resurgent and challenging field of research. We are certain that this book will prove valuable both for practitioners and researchers that are involved with the development of self-adaptive software systems.

Our thanks go to the authors of the contributions for their excellent work, the participants of the Dagstuhl Seminar on Software Engineering for Self-Adaptive Systems for their active participation in the discussions and further contributions, and Alfred Hofmann and his team from Springer for believing in this important topic and for helping us to get the book published. Last but not least, we highly appreciate the efforts of our reviewers who have helped us in ensuring the high quality of the contributions. They are J. Andersson, L. Baresi, N. Bencomo, G. Blair, J. Bradbury, Y. Brun, C. Canal, G. Candea, S.-W. Cheng, C. Cuesta, G. Di Marzo Serugendo, Y. Ding, S. Dustdar, C. Gacek, K. Geihs, J. Georgas, K. M. Goeschka, V. Grassi, R. Hirschfeld, J. Kramer, E. Letier, M. Litoiu, J. Liu, P. Lollini, S. Malek, J. A. McCann, R. Mirandola, H. Müller, J. Noyé, O. Nierstrasz, P. Popov, P. Robertson, M. Sadjadi, G. Salaün, B. Schmerl, U. Scholz, J. P. Sousa, R. Taylor, M. Tichy, M. Tivoli, D. Weyns, and several anonymous reviewers.

March 2009

Betty H.C. Cheng
Rogério de Lemos
Holger Giese
Paola Inverardi
Jeff Magee

Table of Contents

Part 1: Research Roadmap

Part 2: Architecture-Based Self-Adaptation

Part 3: Context-Aware and Model-Driven Self-Adaptation

Software Engineering for Self-Adaptive Systems: A Research Roadmap

Betty H.C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee
(Dagstuhl Seminar Organizer Authors)

Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle
(Dagstuhl Seminar Participant Authors)

r.delemos@kent.ac.uk, holger.giese@hpi.uni-potsdam.de

Abstract. The goal of this roadmap paper is to summarize the state-of-the-art and to identify critical challenges for the systematic software engineering of self-adaptive systems. The paper is partitioned into four parts, one for each of the identified essential views of self-adaptation: modelling dimensions, requirements, engineering, and assurances. For each view, we present the state-of-the-art and the challenges that our community must address. This roadmap paper is a result of the Dagstuhl Seminar 08031 on "Software Engineering for Self-Adaptive Systems", which took place in January 2008.

1 Introduction

The simultaneous explosion of information, the integration of technology, and the continuous evolution from software-intensive systems to ultra-large-scale (ULS) systems require new and innovative approaches for building, running, and managing software systems [1]. A consequence of this continuous evolution is that software systems must become more versatile, flexible, resilient, dependable, robust, energy-efficient, recoverable, customizable, configurable, and self-optimizing by adapting to changing operational contexts, environments or system characteristics. Therefore, *self-adaptation* - systems that are able to adjust their behaviour in response to their perception of the environment and the system itself – has become an important research topic.

It is important to emphasize that in all the many initiatives to explore self-adaptive behaviour, the common element that enables the provision of self-adaptability is usually software. This applies to the research in several application areas and technologies such as adaptable user interfaces, autonomic

B.H.C. Cheng et al. (Eds.): Self-Adaptive Systems, LNCS 5525, pp. 1–26, 2009.

computing, dependable computing, embedded systems, mobile ad hoc networks, mobile and autonomous robots, multi-agent systems, peer-to-peer applications, sensor networks, service-oriented architectures, and ubiquitous computing. It also hold for many research fields, which have already investigated some aspects of self-adaptation from their own perspective, such as fault-tolerant computing, distributed systems, biologically inspired computing, distributed artificial intelligence, integrated management, robotics, knowledge-based systems, machine learning, control theory, etc. In all these case software's flexibility allows such heterogeneous applications; however, the proper realization of the self-adaptation functionality still remains a significant intellectual challenge and only recently have the first attempts in building self-adaptive systems emerged within specific application domains. Moreover, little endeavour has been made to establish suitable software engineering approaches for the provision of self-adaptation. In the long run, we need to establish the foundations that enable the systematic development of future generations of self-adaptive systems. Therefore it is worthwhile to identify the commonalities and differences of the results achieved so far in the different fields and look for ways to integrate them.

The goal of this roadmap paper is to summarize and point out the current state-of-the-art and its limitations, as well as to identify critical challenges for engineering self-adaptive software systems. Specifically, we intend to focus on development methods, techniques, and tools that we believe are required to support the systematic development of complex software systems with dynamic self-adaptive behaviour. In contrast to merely speculative and conjectural visions and ad hoc approaches for systems supporting self-adaptability, the objective of this paper is to establish a roadmap for research, and to identify the main research challenges for the systematic software engineering of self-adaptive systems.

To present and motivate these challenges, the paper divided into four parts, one for each of the four essential views of self-adaptation we have identified. For each view, we present the state-of-the-art and the challenges our community must address. The four views are: modelling dimensions (Section 2), requirements (Section 3), engineering (Section 4), and assurances (Section 5). Finally, we summarize our findings in Section 6.

2 Modelling Dimensions

Endowing a system with a self-adaptive property can take many different shapes. The way self-adaptation has to be conceived depends on various aspects, such as, user needs, environment characteristics, and other system properties. Understanding the problem and selecting a suitable solution requires precise models for representing important aspects of the self-adaptive system, its users, and its environment. A cursory review of the software engineering literature attests to the wide spectrum of software systems that are argued to be self-adaptive. Indeed, there is a lack of consensus among researchers and practitioners on the points of variation among such software systems. We refer to these points of variations as modelling dimensions.

In this section, we provide a classification of modelling dimensions for self-adaptive systems. Each dimension describes a particular aspect of the system that is relevant for self-adaptation. Note that it is not our ambition to be exhaustive in all possible dimensions, but rather to give an initial impetus towards defining a framework for modelling self-adaptive systems. The purpose is to establish a baseline from which key aspects of different self-adaptive system can be easily identified and compared. A more elaborated discussion of the ideas presented in this section can be found in [2].

In the following, we present the dimensions in term of four groups. First, the dimensions associated with self-adaptability aspects of the system goals, second, the dimensions associated with causes of self-adaptation, third, the dimensions associated with the mechanisms to achieve self-adaptability, and fourth, the dimensions related to the effects of self-adaptability upon a system. The proposed modelling framework is presented in the context of an illustrative case from the class of embedded systems, however, these dimensions can be equally useful in describing the self-adaptation properties, for example, of an IT change management system.

2.1 Illustrative Case

As an illustrative scenario, we consider the problem of obstacle/vehicle collisions in the domain of unmanned vehicles (UVs). A concrete application could be the DARPA Grand Challenge contest [3]. Each UV is provided with an autonomous control software system (ACS) to drive the vehicle from start to destination along the road network. The ACS takes into account the regular traffic environment, including the traffic infrastructure and other vehicles. The scenario we envision is the one in which there is a UV driving on the road through a region where people and animals can cross the road unexpectedly. To anticipate possible collisions, the ACS is extended with a self-adaptive control system (SCS). The SCS monitors the environment and controls the vehicle when a human being or an animal is detected in front of the vehicle. In case an obstacle is detected, the SCS manoeuvres the UV around the obstacle negotiating other obstacles and vehicles. Thus, the SCS extends the ACS with self-adaptation to avoid collisions with obstacles on the road.

2.2 Overview of Modelling Dimensions

We give overview of the important modelling dimensions per group. Each dimension is illustrated with an example from the illustrative case.

Goals. Goals are objectives the system under consideration should achieve. Goals could either be associated with the lifetime of the system or with scenarios that are related to the system. Moreover, goals can either refer to the self-adaptability aspects of the application, or to the middleware or infrastructure that supports that application. In the context of the case study mentioned above, amongst several possible goals, we consider, as an example, the following goal:

the system shall avoid collisions. This goal could be expressed in a way in which quantities are associated with the different attributes, and partitioned into sub-goals, with each sub-goal related to one of the attributes.

Evolution. This dimension captures whether the goals can change within the lifetime of the system. The number of goals may change, and the goals themselves may also change as the system as a whole evolves. Hence, goal evolution ranges from static in which changes are not expected, to dynamic in which goals can change at run-time, including the number of goals, i.e., the system is able to manage and create new goals during its lifetime. In the context of the case study since the goal is related to the safety of the UVs it is expected for the goal to be static.

Flexibility. This dimension captures whether the goals are flexible in the way they are expressed. This dimension is related to the level of uncertainty associated with the goal specification, which may range over three values: rigid, constrained, and unconstrained. A goal is rigid when it is prescriptive, while a goal is unconstrained when its statement provides flexibility for dealing with uncertainty. An example of a rigid goal is "the system shall do this..." while an unconstrained goal is "the system might do this..." A constrained goal provides a middle ground, where there is flexibility as long as certain constraints are satisfied, such as, "the system may do this... as long as it does this..." In the context of the case study, the goal is rigid.

Duration. This dimension is concerned with the validity of a goal throughout the system's lifetime. It may range from temporary to persistent. While a persistent goal should be valid throughout the system's lifetime, a temporary goal may be valid for a period of time: short, medium and long term. A persistent goal may restrict the adaptability of the system because it may constrain the system flexibility in adapting to change. A goal that is associated with a particular scenario can be considered a temporary goal. In terms of duration, the goal of the illustrative case can be considered persistent since it is related to the purpose of the system.

Multiplicity. This dimension is related to the number of goals associated with the self-adaptability aspects of a system. A system can either have a single goal or multiple goals. As a general rule of thumb, a single goal self-adaptive system is relatively easier to realize than systems with multiple goals. As discussed in the next dimension, this is particularly true for system where the goals are related. The illustrative case is presented in the context of a single goal.

Dependency. In case a system has multiple goals, this dimension captures how the goals are related to each other. They can be either independent or dependent. A system can have several independent goals (i.e., they don't affect each other). When the goals are dependent, goals can either be complementary with respect to the objectives that should be achieved or they can be conflicting. In the latter

case, trade offs have to be analyzed for identifying an optimal configuration of the goals to be met. In the illustrative case study there are no dependencies since there is a single goal.

Change. Changes are the cause of adaptation. Whenever the system's context changes the system has to decide whether it needs to adapt. In line with [4], we consider context as any information which is computationally accessible and upon which behavioural variations depend. Actors (entities that interact with the system), the environment (the part of the external world with which the system interacts [5]), and the system itself may contribute to the context that may influence the behaviour of the application. Actor-dependent, system-dependent, and environment-dependent variations can occur separately, or in any combination. We classify context-dependable changes of a self-adaptive system in terms of the place in which change has occurred, the type and the frequency of the change, and whether it can be anticipated. All these elements are important for identifying how the system should react to change that occurs during run-time. In the context of the illustrative case study, we consider the cause of adaptation the appearance of an obstacle in front of the ACS.

Source. This dimension identifies the origin of the change, which can be either external to the system (i.e., its environment) or internal to the system, depending on the scope of the system. In case the source of change is internal, it might be important to identify more precisely where change has occurred: application, middleware or infrastructure. The source of the change related to the ACS is external to the system.

Type. This dimension refers to the nature of change. It can be functional, non-functional, and technological. Technological refers to both software and hardware aspects that support the delivery of the services. Examples of the three types of change are, respectively: the purpose of the system has changed and services delivered need to reflect this change, system performance and reliability need to be improved, and the version of the middleware in which the application runs has been upgraded. In the illustrative case, since the change can lead ACS to collide against an obstacle the type of change is non-functional.

Frequency. This dimension is concerned with how often a particular change occurs, and it can range from rare to frequent. If for example a change happens quite often this might affect the responsiveness of the adaptation. Since the occurrence of obstacles is rare within the system lifetime, we consider changes are rare to occur.

Anticipation. This dimension captures whether change can be predicted ahead of time. Different self-adaptive techniques are necessary depending on the degree of anticipation: foreseen (taken care of), foreseeable (planned for), and unforeseen (not planned for) [6]. In the illustrative case study, the occurrence of obstacles should be foreseeable.

Mechanisms. This set of dimensions captures the system reaction towards change, which means that they are related to the adaptation process itself. The dimensions associated with this group refer to the type of self-adaptation that is expected, the level of autonomy of the self-adaptation, how self-adaptation is controlled, the impact of self-adaptation in terms of space and time, how responsive is self-adaptation, and how self-adaptation reacts to change.

Type. This dimension captures whether adaptation is related to the parameters of the system's components or to the structure of the system. Based on this, adaptation can be parametric or structural, or a combination of these. Structural adaptation could also be seen as compositional, since it depends on how components are integrated. In the illustrative case, to avoid collisions with obstacles, the SCS has to adjust the movements of the UV, and this might imply adjusting parameters in the steering gear.

Autonomy. This dimension identifies the degree of outside intervention during adaptation. The range of this dimension goes from autonomous to assisted. In the autonomous case, at run-time there is no influence external to the system guiding how the system should adapt. On the other hand, a system can have a degree of self-adaptability when externally assisted, either by another system or by human participation (which can be considered another system). In the illustrative case, for the foreseen type of changes the system is autonomous since the UV has to avoid collisions with animals without any human intervention.

Organization. This dimension captures whether adaptation is performed by a single component - centralized, or distributed amongst several components - decentralized. If adaptation is decentralized no single component has a complete control over the system. The SCS of the UV in the illustrative example seems to fit naturally with a weak organization.

Scope. This dimension identifies whether adaptation is localized or involves the entire system. The scope of adaptation can range from local to global. If adaptation affects the entire system then more thorough analysis is required to commit the adaptation. It is fundamental for the system to be well structured in order to reduce the impact that change might have on the adaptation. In the illustrative case, the adaptation is global to the UV since involves different components in the car, such as, steering gear and brakes.

Duration. This dimension refers to the period of time in which the system is self-adapting, or in other words, how long the adaptation lasts. The adaptation process can be for short (seconds to hours), medium (hours to months), or long (months to years) term. Note that time characteristics should be considered relative to the application domain. While scope dimension deals with the impact of adaptation in terms of space, duration deals with time. Considering that the time it takes for the UV to react to an obstacle is minimal compared with the lifetime of the system, the duration of the self-adaptation should be short term.

Timeliness. This dimension captures whether the time period for performing self-adaptation can be guaranteed, and it ranges from best-effort to guaranteed. For example, in case change occurs quite often, it may be the case that it is impossible to guarantee that adaptation will take place before another change occurs, in these situations best effort should be pursued. In the context of the case study, the upper bounds for the SCS to manoeuvre the UV should be identified for the timeliness associated with self-adaptation to be guaranteed.

Triggering. This dimension identifies whether the change that initiates adaptation is event-trigger or time-trigger. Although it is difficult to control how and when change occurs, it is possible to control how and when the adaptation should react to a certain change. If the time period for performing adaptation has to be guaranteed, then an event-trigger might not provide the necessary assurances when change is unbounded. Obstacles in the illustrative case appear unexpectedly and as such triggering of self-adaptation is event-based.

Effects. This set of dimensions capture what is the impact of adaptation upon the system, that is, it deals with the effects of adaptation. While mechanisms for adaptation are properties associated with the adaptation, these dimensions are properties associated with system in which the adaptation takes place. The dimensions associated with this group refer to the criticality of the adaptation, how predictable it is, what are the overheads associated with it, and whether the system is resilient in the face of change. In the context of the illustrative case study, a collision between an UV and an obstacle may ensue if the SCS fails.

Criticality. This dimension captures the impact upon the system in case the self-adaptation fails. There are adaptations that harmless in the context of the services provided by the system, while there are adaptations that might involve the loss of life. The range of values associated with this criticality is harmless, mission-critical, and safety-critical. The level of criticality of the application (and the adaptation process) is safety-critical since it may lead to an accident.

Predictability. This dimension identifies whether the consequences of self-adaptation can be predictable both in value and time. While timeliness is related to the adaptation mechanisms, predictability is associated with system. Since predictability is associated with guarantees, the degree of predictability can range from non-deterministic to deterministic. Given the nature of the illustrative case, the predictability of the adaptation should be deterministic.

Overhead. This dimension captures the negative impact of system adaptation upon the system's performance. The overhead can range from insignificant to system failure (e.g., thrashing). The latter will happen when the system ceases to be able to deliver its services due to the high-overhead of running the self-adaptation processes (monitoring, analyzer, planning, effecting processes). The

overheads associated with the SCS should be insignificant, otherwise the UV might not be able to avoid the obstacle.

Resilience. This dimension is related to the persistence of service delivery that can justifiably be trusted, when facing changes [6]. There are two issues that need to be considered under this dimension: first, it is the ability of the system to provide resilience, and second, it is the ability to justify the provided resilience. The degree of resilience can range from resilient to vulnerable. In the context of the illustrative case study, the system should be resilient.

2.3 Research Challenges in Modelling Dimensions

In spite of the many years of software engineering research, construction of self-adaptive software systems has remained a very challenging task. While substantial progress has been made in each of the discussed modelling dimensions, there are several important research questions that are remaining, and frame the future research in this area. We briefly elaborate on those below. The discussion is structured in line with the four presented groups of modelling dimensions.

Goals. A self-adaptive software system often needs to perform a trade-off analysis between several potentially conflicting goals. Practical techniques for specifying and generating utility functions, potentially based on the user's requirements, are needed. One promising direction is to use preferences that compare situations under Pareto optimal conditions.

Change. Monitoring a system, especially when there are several different QoS properties of interest, has an overhead. In fact, the amount of degradation in QoS due to monitoring could outweigh the benefits of improvements in QoS to adaptation. More research on lightweight monitoring techniques is needed.

Mechanisms. Researchers and practitioners have typically leveraged a single tactic to realize adaptation based on the characteristics of the target application. However, given the unique benefits of each approach, we believe a fruitful avenue of future research is a more comprehensive approach that leverages several adaptation tactics simultaneously.

The application of the centralized control loop pattern to a large-scale software system may suffer from scalability problems. There is a pressing need for decentralized, but still manageable, efficient, and predictable techniques for constructing self-adaptive software systems. A major challenge is to accommodate a systematic engineering approach that integrates both control-loop approaches with decentralized agent inspired approaches.

Responsiveness is a crucial property in real-time software systems, hence the need for adaptation models targeted for real-time systems that treat the duration and overhead of adaptation as first class entities.

Effects. Predicting the exact behaviour of a software system due to run-time changes is a challenging task. More advanced and predictive models of adaptation are needed for systems that could fail to satisfy their requirements due to side-effects of change.

In highly dynamic systems, such as mobile systems, where the environmental parameters change frequently, the overhead of adaptation due to frequent changes in the system could be so high that the system ends up thrashing. The trade-offs between the adaptation overhead and the accrued benefits of changing the system needs to be taken into consideration for such systems.

3 Requirements

A self-adaptive system is able to modify its behaviour according to changes in its environment. As such, a self-adaptive system must continuously monitor changes in its context and react accordingly. But what aspects of the environment should the self-adaptive system monitor? Clearly, the system cannot monitor everything. And exactly what should the system do if it detects less than optimal conditions in the environment? Presumably, the system still needs to maintain a set of high-level goals that should be satisfied regardless of the environmental conditions. But non-critical goals could well be relaxed, thus allowing the system a degree of flexibility during or after adaptation.

These questions (and others) form the core considerations for building self-adaptive systems. Requirements engineering is concerned with what a system should do and within which constraints it must do it. Requirements engineering for self-adaptive systems, therefore, must address what adaptations are possible and what constrains how those adaptations are realized. In particular, questions to be addressed include: what aspects of the environment are relevant for adaptation? Which requirements are allowed to vary or evolve at run-time, and which must always be maintained? In short, requirements engineering for self-adaptive systems must deal with uncertainty because the information about future execution environments is incomplete, and therefore the requirements for the behavior of the system may need to change (at run-time) in response to the changing environment.

3.1 Requirements State-of-the-Art

Requirements engineering for self-adaptive systems appears to be a wide open research area with only a limited number of approaches yet considered. Cheng and Atlee [7] report on some previous work on specifying and verifying adaptive software, and on run-time monitoring of requirements conformance [8,9]. They also explain how preliminary work on personalized and customized software can be applied to adaptive systems (e.g., [10,11]). In addition, some research approaches have successfully used goal models as a foundation for specifying the autonomic behaviour [12] and requirements of adaptive systems [13].

One of the main challenges that self-adaptation poses is that when designing a self-adaptive system, we cannot assume that all adaptations are known in advance — that is, we cannot anticipate requirements for the entire set of possible environmental conditions and their respective adaptation specifications. For example, if a system is to respond to cyber-attacks, one cannot possibly know all attacks in advance since malicious actors develop new attack types all the time.

As a result, requirements for self-adaptive systems may involve degrees of uncertainty or may necessarily be specified as "incomplete." The requirements specification therefore should cope with:

- the incomplete information about the environment and the resulting incomplete information about the respective behaviour that the system should expose
- the evolution of the requirements at run-time

3.2 Research Challenges in Requirements

This subsection highlights a number of short-term and long-term research challenges for requirements engineering for self-adaptive systems. We start with shorter-term challenges and progress to more visionary ideas. As far as the authors are aware, there is little or no research currently underway to address these challenges.

A New Requirements Language. Current languages for requirements engineering are not well suited to dealing with uncertainty, which, as mentioned above, is a key consideration for self-adaptive systems. We therefore propose that richer requirements languages are needed. Few of the existing approaches for requirements engineering provide this capability. In goal-modelling notations such as KAOS [14] and i* [15], there is no explicit support for uncertainty or adaptivity. Scenario-based notations generally do not explicitly support adaptation either, although live sequence charts [16] have a notion of mandatory versus potential behaviour that could possibly be used to specify adaptive systems. Of course, the most common notation for specifying requirements in industry is still natural language prose. Traditionally, requirements documents make statements such as "the system shall do this..." For self-adaptive systems, the prescriptive notion of "shall" needs to be relaxed and could, for example, be replaced with "the system may do this...or it may do that ..." or "if the system cannot do this...then it should eventually do that..." This idea leads to a new requirements vocabulary for self-adaptive systems that gives stakeholders the flexibility to account for uncertainty in their requirements documents. For example:

Traditional RE:
- "*The system shall do this...*"

Adaptive RE:
- "*The system might do this...*"
- "*But it may do this...as long as it does this...*"

– "*The system ought to do this... but if it cannot, it shall eventually do this...*"

Such a vocabulary would change the level of discourse in requirements from prescriptive to flexible. There would need to be a clear definition of terms, of course, as well as a composition calculus for defining how the terms relate to each other and compose. Multimodal logic and perhaps new adaptation-oriented logic [17] need to be developed to specify the semantics for what it means to have the possibility of conditions [18,19]. There is also a relationship with variability management mechanisms in software product lines [20], which also tackle built-in flexibilities. However, at the requirements level, one ideally would capture uncertainty at a more abstract level than simply enumerating alternatives. Some preliminary results in defining a new adaptation requirements language along these lines are being developed [21].

Mapping to Architecture. Given a new requirements language that explicitly handles uncertainty, it will be necessary to provide systematic methods for refining models in this language down to specific architectures that support run-time adaptation. A variety of technical options exist for implementing reconfigurability at the architecture level, including component-based, aspect-oriented and product-line based approaches, as well as combinations of these. Potentially, there could be a large gap in expressiveness between a requirements language that incorporates uncertainty and existing architecture structuring methods. One can imagine, therefore, a semi-automated process for mapping to architecture where heuristics and/or patterns are used to suggest architectural units corresponding to certain vocabulary terms in the requirements.

Managing Uncertainty. In general, once we start introducing uncertainty into our software engineering processes, we must have a way of managing this uncertainty and the inevitable complexity associated with handling so many unknowns. Certain requirements will not change (i.e., invariants), whereas others will permit a degree of flexibility. For example, a system cannot start out as a transport robot and self-adapt into a robot chef [22]! Allowing uncertainty levels when developing self-adaptive systems requires a trade-off between flexibility and assurance such that the critical high-level goals of the application are always met [23,24,25].

Requirements Reflection. As we said above, self-adaptation deals with requirements that vary at run-time. Therefore it is important that requirements lend themselves to be dynamically observed, i.e., during execution. Reflection [26,27,28] enables a system to observe its own structure and behaviour. A relevant research work is the ReqMon tools [29] which provides a requirements monitoring framework, focusing on temporal properties to be maintained. Leveraging and extending beyond these complementary approaches, Finkelstein [22] coins the term "requirements reflection" that would enable systems to be aware of their own requirements at run-time. This capability would require an

appropriate model of the requirements to be available online. Such an idea brings with it a host of interesting research questions, such as: Could a system dynamically observe its requirements? In other words, can we make requirements run-time objects? Future work is needed to develop technologies to provide such infrastructure support.

Online Goal Refinement. As in the case of design decisions that are eventually realized at run-time, new and more flexible requirement specifications like the one suggested above would imply that the system should perform the RE processes at run-time, e.g. goal-refinement [25].

Traceability from Requirements to Implementation. A constant challenge in all the topics shown above is dynamic traceability. For example, new operators of a new RE specification language should be easily traceable down to architecture, design, and beyond. Furthermore, if the RE process is performed at run-time we need to assure that the final implementation or behaviour of the system matches the requirements. Doing so is different from the traditional requirements traceability.

The above research challenges the requirements engineering (RE) community will face, as the demand for self-adaptive systems continues to grow, span RE activities during the development phases and run-time. In order to gain assurance about adaptive behaviour, it is important to monitor adherence and traceability to the requirements during run-time. Furthermore, it is also necessary to acknowledge and support the evolution of requirements at run-time. Given the increasing complexity of applications requiring run-time adaptation, the software artefacts with which the developers manipulate and analyze must be more abstract than source code. How can graphical models, formal specifications, policies, etc. be used as the basis for the evolutionary process of adaptive systems versus source code, the traditional artefact that is manipulated once a system has been deployed? How can we maintain traceability among relevant artefacts, including the code? How can we maintain assurance constraints during and after adaptation? How much should a system be allowed to adapt and still maintain traceability to the original system? Clearly, the ability to dynamically adapt systems at run-time is an exciting and powerful capability. The RE community, among other software engineering disciplines, need to be proactive in tackling these complex challenges in order to ensure that useful and safe adaptive capabilities are provided to the adaptive systems developers.

4 Engineering

The engineering of self-adaptive software systems is a major challenge, especially if predictability and cost-effectiveness are desired. However, in other areas of engineering and nature there is a well-known, pervasive notion that could be potentially applied to software systems as well: the notion of feedback.

The first mechanical system that regulated its speed automatically using feedback was Watt's steam engine that had a regulator implementing feedback

control principles. Also in nature plenty examples for positive and negative feedback can be found that help to regulate processes.

Even though control engineering [30,31] as well as feedback found in nature are not targeting software systems, mining the rich experiences of these fields and applying principles and findings to software-intensive adaptive systems is a most worthwhile and promising avenue of research for self-adaptive systems. We further strongly believe that self-adaptive systems must be based on this feedback principle and we advocate in this section to focus on the 'control loop' when engineering self-adaptive systems.

In this section we first examine the generic control loop and then analyze the control loop's role in control theory, natural systems, and software engineering, respectively. Finally, we describe the challenges whose resolutions are necessary to enable the systematic engineering of self-adaptive systems. A more detailed elaboration of the perspective presented in this section can be found in [32].

4.1 Control Loop Model

Self-adaptation aspects of software-intensive systems can often be hidden within the system design. What self-adaptive systems have in common is that (1) typically design decisions are partially made at run-time, and (2) the systems reason about their state and environment. This reasoning typically involves feedback processes with four key activities: collect, analyze, decide, and act, as depicted in Figure 1 [33].

Here, we concentrate on self-adaptive systems with feedback mechanisms controlling their dynamic behaviour. For example, keeping web services up and running for a long time requires collecting of information about the current state of the system, analyzing that information to diagnose performance problems or to detect failures, deciding how to resolve the problem (e.g., via dynamic load-balancing or healing), and acting on those decisions.

The generic model of a control loop based on [33] (cf. Figure 1) provides an overview of the main activities around the control loop but ignores properties of the control and data flow around the loop. When engineering a self-adaptive system, questions about these properties become important. We now identify such questions and argue that in order to properly design self-adaptive software systems, these questions must be brought to the forefront of the design process.

The feedback cycle starts with the *collection* of relevant data from environmental sensors and other sources that reflect the current state of the system. Some of the engineering questions that Figure 1 ignores with respect to collection but that are important to the engineering process are: What is the required sample rate? How reliable is the sensor data? Is there a common event format across sensors?

Next, the system *analyzes* the collected data. There are many approaches to structuring and reasoning about the raw data (e.g., using applicable models, theories, and rules). Some of the important questions here are: How is the current state of the system inferred? How much past state may be needed in the future? What data need to be archived for validation and verification? How faithful is the

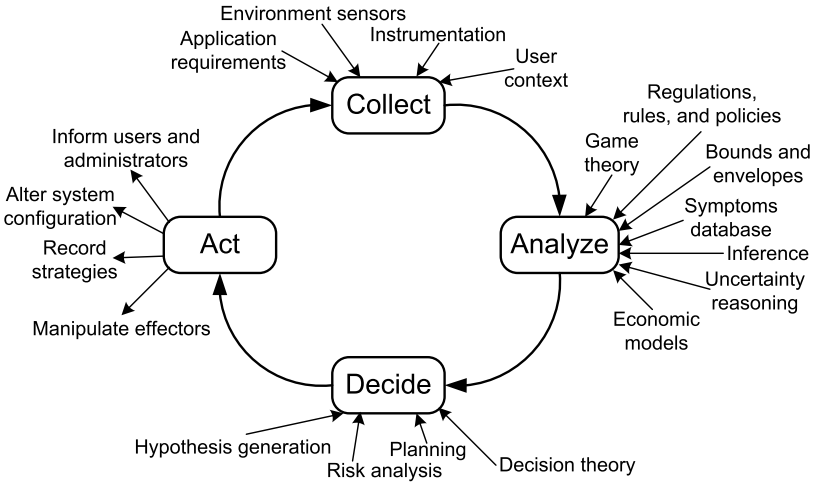


Fig. 1. Activities of the control loop

model to the real world? Can an adequate model be derived from the available sensor data?

Next, the system makes a *decision* about how to adapt in order to reach a desirable state. Approaches such as risk analysis can help make this decision. Here, the important questions are: How is the future state of the system inferred? How is a decision reached (e.g., with off-line simulation or utility/goal functions)? What are the priorities for adaptation across multiple control loops and within a single control loop?

Finally, to implement the decision, the system must *act* via available actuators and effectors. Important questions here are: When should the adaptation be safely performed? How do adjustments of different control loops interfere with each other? Does centralized or decentralized control help achieve the global goal? Does the control system have sufficient command authority over the process—that is, can the action be implemented using the available actuators and effectors?

The above questions—as well as others—regarding the control loop should be explicitly identified, recorded, and resolved during the development of the self-adaptive system.

4.2 Control Loops and Control Theory

The control loop is a central element of control theory, which provides well-established mathematical models, tools, and techniques to analyze system performance, stability, sensitivity, or correctness [34,35]. Researchers have applied results of control theory and control engineering to building self-adaptive systems. However, it is not clear if general principles of this discipline (e.g., open/closed-loop controller, observability, controllability, stability, or hysteresis) are applicable to self-adaptive software systems.

Control engineering has determined that systems with a single control loop are easier to reason about than systems with multiple loops. Unfortunately, the latter types of control loops are far more common. Good engineering practice calls for reducing multiple control loops to a single one, or making control loops independent of each other [36]. When such decoupling is impossible, the design must make the interactions of control loops explicit and expose how these interactions are handled.

Control engineering has also identified hierarchical organization of control loops as a fruitful way to decouple control-loop interactions. The different time scales of the different layers of the hierarchy can minimize the unexpected interference between control loops. This scheme is of particular interest if we distinguish between forms of adaptation such as change management and goal management [25] and can organize them hierarchically.

While mining control engineering for control-loop mechanisms applicable to software engineering can result in breakthroughs in engineering self-adaptive systems, one important obstacle is that different application areas of control engineering introduce distinct nomenclature and architectural diagrams for their realizations of the generic control loop depicted in Figure 1. It is useful to investigate how different application areas realize this generic control loop and to identify the commonalities in order to compare and leverage self-adaptive systems research from different application areas. For example, control engineering has developed standard approaches to model and reason about feedback such as the Model Reference Adaptive Control (MRAC) [31] and the Model Identification Adaptive Control (MIAC) [37].

Models such as MRAC and MIAC introduce well-defined elements such as controller, process, adjustment mechanism, and system identification or model reference along with prescribed dependencies among these elements. This form of separation of concerns suggests that these models are a solid starting point for the design of self-adaptive software-intensive systems. In fact, many participants of the Dagstuhl Seminar 08031 [38] presented self-adaptive systems that can be expressed in terms of standardized models from control engineering such as MRAC and MIAC. Examples of presented systems include a self-adaptive flight-control system that realizes a more robust aircraft control capable of handling multiple faults (e.g., change of aircraft dynamics due to loss of control surface, aileron, or stabilator) [39]; a system of autonomous shuttles that operate on demand and in a decentralized manner using a wireless network [40]; a multi-agent approach to an AGV transportation system that allows agents to flexibly adapt their behavior to changes in their context, realizing cooperative self-adaptation [41]; and the Rainbow system [42], whose architecture was mapped by Shaw to the classical control loop in control theory [43].

4.3 Control Loops and Natural Systems

In contrast to engineered self-adaptive systems, biologically or sociologically inspired systems do not often have clearly visible control loops. Furthermore, the

systems are often decentralized in such a way that the agents do not have a sense of the global goal but rather it is the interaction of their local behaviour that yields the global goal as an emergent property.

Nature is full of self-adapting systems that leverage mechanisms and types of control loops far removed from those we use today when engineering self-adaptive systems. Mining this rich collection of systems and creating a catalogue of feedback types and self-adaption techniques is an important and likely fruitful endeavour our community must undertake.

Some software systems that leverage mechanisms found in nature already exist and promise a bright future for nature-inspired software engineering techniques. For example, in systems built using the crystal-growth-inspired tile architectural style [44], components distributed around the Internet come together to "self-assemble" and "self-organize" into a solution to an NP-complete problem. These systems can self-adapt to exhibit properties of fault and adversary tolerance [45]. The self-adaptation control loop is not easily evident in the nature's process of crystal growth, but it does exist and increasing our understanding of such control loops will increase our ability to engineer self-adaptive software systems.

In addition to discovering new self-adaptation mechanisms, mining natural systems and creating a catalogue can facilitate engineering of new novel mechanisms as the combinations of existing ones. For example, while many systems in nature use bottom-up adaptation mechanisms, it may be possible to unify the self-adaptive top-down and self-organizing (bottom-up) mechanisms via software architecture by considering metadata and policies with adaptation properties and control-loop reasoning explicitly, both at design-time and run-time [46].

4.4 Control Loops and Software Engineering

We have observed that control loops are often hidden, abstracted, or internalized when presenting the architecture of self-adaptive systems [43]. However, the feedback behaviour of a self-adaptive system, which is realized with its control loops, is a crucial feature and, hence, should be elevated to a first-class entity in its modelling, design, and implementation.

When engineering a self-adaptive system, the properties of the control loops affect the system's design and architecture. Therefore, besides the control loops, those control loops' properties must be made explicit as well. In one approach, Cheng et al. [47] advocate making self-adaptation external, as opposed to internal or hard-wired, to separate the concerns of system functionality from the concerns of self-adaptation.

Despite recent attention to self-adaptive systems (e.g., several ICSE workshops), development and analysis methods for such systems do not yet provide sufficient explicit focus on the control loops and their associated properties that almost inevitably control self-adaptation.

The idea of increasing the visibility of control loops in software architectures and software methods is not new. Over a decade ago, Shaw compared a software design method based on process control to an object-oriented design method [48].

She introduced a new software organization paradigm based on control loops, one with an architecture that is dominated by feedback loops and their analyses, rather than by the identification of discrete stateful objects.

4.5 Research Challenges in Engineering

We have argued that control loops are essential for self-adaptive systems. Therefore, control loops must become first-class entities when engineering self-adaptive systems. Understanding and reasoning about the control loops of a self-adaptive systems is integral to advancing the engineering of self-adaptive systems' maturation from an ad-hoc, trial-and-error endeavour to a disciplined approach. We identify the following issues as the current most critical challenges that must be addressed in order to achieve a disciplined approach to engineering self-adaptive systems:

Modelling. Making the control loops explicit and exposing self-adaptive properties to allow the designer to reason about the system modelling support for control loops.

Architecture. Developing reference architectures for adaptive systems that address issues such as structural arrangements of control loops (e.g., sequential, parallel, hierarchy, decentralized), interactions among control loops, data flow around the control loops, tolerances, trade-offs, sampling rates, stability and convergence conditions, hysteresis specifications, and context uncertainty.

Design. Compiling a catalogue of common control-loop schemes and characterizing control-loop elements, along with associated obligations in the form of patterns to help classify specific kinds of interacting control loops, e.g., for manual vs. automatic control or for decoupling control loops from one another. These control-loop schemes should come from exploring existing knowledge in control engineering, as well as other fields that use feedback, and from mining naturally occurring systems that use adaptation.

Middleware Support. Developing middleware support to "allow researchers with different motivations and experiences to put their ideas in practice, free from the painful details of low-level system implementation" [49] by supporting a framework and standardized interfaces for self-adaptive functionality.

Verification & Validation. Creating validation and verification techniques to test and evaluate control loops' behaviour and automatically detect unintended interactions.

Reengineering. Exploring techniques for evolving existing systems and injecting self-adaptation into such systems.

Human-Computer Interaction. Analyzing feedback types from human-computer interaction and devising novel mechanisms for exposing the control loops to the users, keeping the users of self-adapting systems "in the loop" to ensure their trust.

5 Assurances

The goal of system assurance is simple. Developers need to provide evidence that the set of stated functional and non-functional properties are satisfied during system's operation. While the goal is simple, achieving it is not. Traditional verification and validation methods, static or dynamic, rely of stable descriptions of software models and properties. The characteristics of self-adaptive systems create new challenges for developing high-assurance systems. Current verification and validation methods do not align well with changing goals and requirements as well as variable software functionality. Consequently, novel verification and validation methods are required to provide assurance in self-adaptive systems.

In this section, we present a generalized verification and validation framework which specifically targets the characteristics of self-adaptive systems. Thereafter, we present a set of research challenges for verification and validation methods implied by the presented framework.

5.1 Assurances Framework

Self-adaptive systems are highly context dependent. Whenever the system's context changes the system has to decide whether it needs to adapt. Whenever the system decides to adapt, this may prompt the need for verification activities to provide continual assessment. Moreover, depending on the dynamics of change, verification activities may have to be embedded in the adaptation mechanism.

Due to the uniqueness of such assessment process, we find it necessary to propose a framework for adaptive system assurance. This framework is depicted in Figure 2. Over a period of operation, the system operates through a series of operational modes. Modes, represented in Figure 2 by index j, represent known and, in some cases, unknown phases in the computational lifecycle. Examples of known modes in flight control include altitude hold mode, flare mode and touchdown mode. Sequences of behavioural adjustments in the known modes are known. But, continuing with the same example, if failures change the airframe dynamics, the application's context changes and software control needs to sense and adapt to the conditions unknown prior to the deployment.

Such adaptations are reflected in a series of context - system state (whatever this is for a self-adaptive system) configurations. $(C+S)_{j_i}$ denotes the i^{th} combination of context and system state in a cycle which is related to the requirements of the system mode j. At the level of configurations it is irrelevant whether the context or the system state changes (transition t_{j_0}), the result always is a new configuration.

Goals and requirements of a self-adaptive system may also change during runtime. We abstract from the subtle differences between goals and requirements

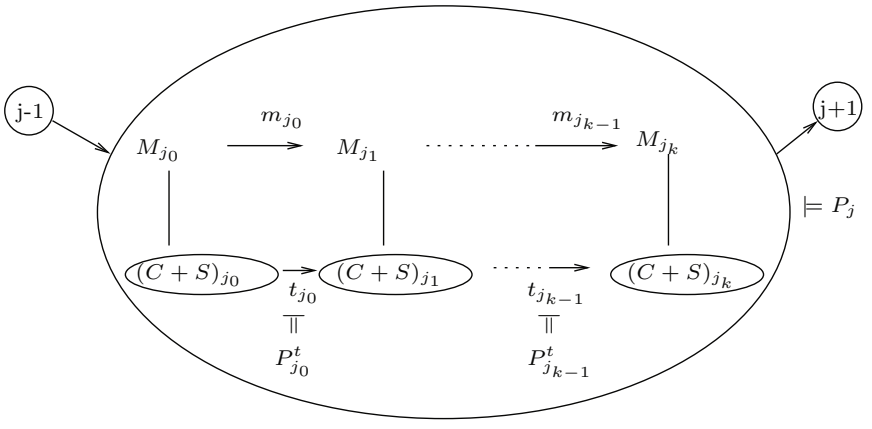


Fig. 2. V & V model

for the generalized framework and instead use the more generic term *properties*. In self-adaptive systems, properties may change over time in response to changes in the system context or the system state. Properties might be relevant in one context and completely irrelevant in some other. Properties might even be weighted against each other, resulting in a trade offs between properties and, thus, their partial fulfilment. Properties can also be related to each other. Global properties like safety requirements must be satisfied over the entire life time of a system, through all the system modes. Different local properties $P^t_{j_i}$ within a context might guarantee a global property. Similarly, a local property may guarantee that a global property is not invalidated by the changes.

Verification of the properties typically relies on the existence of models. System dynamics and changing requirements of self-adaptive systems make it impossible to build a steady model before system deployment and perform all verification tasks on such a model. Therefore, models need to be built and maintained at run-time. In Figure 2, M_{j_i} is the model that corresponds to configuration $(C+S)_{j_i}$. Each change in the system configuration needs to be reflected at model level as well, letting the model evolve accordingly from one configuration to the other, not necessarily linearly as depicted in Figure 2. Delays in model definition may also be inevitable. In Figure 2, the evolution of models from one configuration to the other is denoted by m_{j_i} . The complexity of such evolution moves along two dimensions. On one side the model must be efficiently updated to reflect the system changes, on the other it should still reflect an accurate representation of reality.

5.2 Research Challenges in Assurances

While verification and validation of properties in distributed systems is not a novel problem, a number of additional issues arise in the context of self-adaptation due to the nature of these applications. Self-adapting systems have

to contend with dynamic changes in modes and contexts as well as the dynamic changes in user requirements. Due to this high dynamism, V&V methods traditionally applied at requirements and design stages of development must be supplemented with run-time assurance techniques.

Dynamic Identification of Changing Requirements. System requirements can change implicitly, as a result of a change in context. Since in dynamic environments all eventualities cannot be anticipated, self-adapting systems have to be able to identify new contexts. There will inevitably be uncertainty in the process of context identification. Once the context is identified, utility functions evaluate trade-offs between the properties (goals) aligned with the context. The adequacy of context identification and utility functions is subject to verification. It appears that failure detection and identification techniques from distributed fault tolerant computing are a special case of context identification. Given that all such techniques incorporate uncertainty, probabilistic approaches to assurance seem to be the most promising research direction.

Adaptation-Specific Model-Driven Environments. To deal with the challenges of adaptation we envisage a model-driven development, where models play a key role throughout the development [50]. Models allow the application of verification and validation methods during the development process and can support self-adaptation at run-time. In fact, models can support estimation of system's status, so that the impact of a context change can be predicted. Provided that such predictions are reliable, it should be possible to perform model-based adaptation analysis as a verification activity [51]. A key issue in this approach is to keep the run-time models synchronized with the changing system. Any model based verification, therefore, presumes the availability of accurate change tracking algorithms that keep system model synchronized with the run-time environment. Uncertain model attributes can be described, for example, using probability distribution functions, the attribute value ranges, or using the analysis of historical attribute values. These methods can take advantage of probability theory and statistics that helped solve stochastic problems in the past.

Agile Run-Time Assurance. In situations when models that accurately represent the dynamic interaction between system context and state cannot be developed, performing verification activities that address verification at run-time are inevitable. The key requirement for run-time verification is the existence of efficient agile solution algorithms which do not require high space/time complexity. Self-adaptive systems may change their state quickly to respond to context or property changes. An interesting class of verification techniques is that inspired by Proof-Carrying Code (PCC). PCC is a technique by which a host platform can verify that code provided that needs to be executed adheres to a predefined, still limited, set of safety rules. The limitation of the PCC paradigm is that executed code must contain a formal safety proof that attests to the fact that the code respects the defined safety policy. Defining such kind of proofs

for code segments which are parameterized and undergo changes and for larger classes of safety properties is a challenge. When formal property proofs do not seem feasible, run-time assurance techniques may rely on demonstrable properties of adaptation, like convergence and stability. Adaptation is a transient behaviour and the fact that a sequence of observable states converge towards a stable state is always desirable. Transient states may not satisfy local or global properties (or we just cannot prove that they do). Therefore, the analysis of the rate of convergence may inspire confidence that system state will predictably quickly reach a desirable state. Here we intentionally use term "desirable" rather than "correct" state because we may not know what a correct adaptation is in an unforeseen context [52]. This problem necessitates investigation of scientific principles needed to move software assurance beyond current conceptions and calculations of correctness.

Liability and Social Aspects. Adaptive functionality in safety-critical systems is already a reality. Applications of adaptive computing in safety critical systems are on the rise [39,53]. Autonomous software adaptation raises new challenges in the legal and social context. Generally, if software does not perform as expected, the creator may be held liable. Depending on the legal theory, different issues will be relevant in a legal inquiry [54]. Software vendors may have a difficult time to argue that they applied the expected care when developing a critical application if the software is self-adaptive. Software may enter unforeseeable states that have never been tested or reasoned about. It can be also argued that current state-of-the-art engineering practices are not sufficiently mature to warrant self-adaptive functionality. However, certain liability claims for negligence may be rebutted if it can be show safety mechanisms could disable self-adaptive functionality in hazardous situations. Assurance of self-adaptive software is then not only a step to make the product itself safer, but should be considered a valid defence against liability claims.

6 Lessons and Challenges

In this section, we present the overall conclusions of the roadmap paper in the context of lessons learned and the major ensuing challenges for our community. First and foremost, we must point out that this exercise had no intention of being exhaustive. We made the choice to focus on the four major issues we identified as the key in the software engineering of self-adaptive systems process: modelling dimensions, requirements, engineering, and assurances.

The presentations of each of the four views intend not to cover all the related aspects, but rather focused theses as a means for identifying the challenges associated with each view. The four identified theses are:

– *modelling dimensions* - the need to enumerate and classify modelling dimensions for obtaining precise models to support run-time reasoning and decision making for achieving self-adaptability;

- *requirements* - the need to define a new requirements language for handling uncertainty to give self-adaptive systems the freedom to do adaptation;
- *engineering* - the need to consider feedback control loops as first-class entities during engineering of self-adaptive systems;
- *assurances* - the need to define novel verification and validation methods for the provision of assurances that cover the self-adaptation of systems.

We now summarize the most important challenges of each the views identified.

Modelling Dimensions. A major challenge in modelling dimensions is defining models that can represent a wide range of system properties. The more precise the models are, the more effective they should be in supporting run-time analysis and decision process. However, at the same time, models should be sufficiently general and simple to keep synthesis feasible. Defining utility functions for supporting decision making is a challenging task, and we need practical techniques to specify and generate such utility functions.

Requirements. The major challenge in requirements is defining a new language capable of capturing uncertainty at an abstract level. Once we consider uncertainty at the requirements stage, we must also find means of managing it. Thus, we need to represent the trade offs between the flexibility provided by the uncertainty and the assurances required by the application. Since requirements might vary at run-time, systems should be aware of their own requirements, creating a need for requirements reflection and online goal refinement. It is important to note that requirements should not be considered in isolation and we must develop techniques for mapping requirements into architecture and implementation.

Engineering. In order to properly engineer self-adaptive software systems, the feedback control loop must become a first-class entity throughout the process. To allow this, there is the need for modelling support to make the loop's role explicit. Explicit modelling of the loops will ease reifying system properties to allow their query and modification at run-time. In order to facilitate reasoning between system properties and the feedback control loop, reference architectures must highlight key aspects of the loop, such as, number, structural arrangements, interactions, and stability conditions. In order to maintain users' trust, certain aspects of the control must be exposed to the users. Finally, in order to facilitate organized use and reuse of self-adaptation mechanisms, the community must compile a catalog of feedback control loops, explicitly explaining their properties, benefits and shortcomings, and interaction possible methods for interaction with other loops.

Assurances. The major challenge in assurances is supplementing traditional methods applied at requirements and design stages of development with run-time assurances. Since system context is dynamic at run-time, systems must identify new contexts dynamically. In order to handle the uncertainty associated

with this process, models must include uncertainty via, e.g., probabilistic approaches. Further, adaptation-specific model-driven environments may facilitate modelling support of run-time self-adaptation; however, these environments must be lightweight, in order to allow run-time verification without impacting system performance. One approach to run-time verification of assurances is the labelling of such assurances as "desirable," rather than "required."

There are several aspects related to software engineering of self-adaptive systems that we did not cover. One of them is processes, which are an integral part of software engineering. Software engineering processes are essentially associated with design-time; however, engineering of self-adaptive systems will also require run-time processes for handling change. This may require re-evaluating how software should be developed for self-adaptive systems. For example, instead of a single process, two complementary processes may be required for coordinating the design-time and run-time activities of building software, which might lead to a whole new way of developing software. Technology should enable and influence the development of self-adaptive systems. Other aspects of software engineering related to self-adaptation are technologies like model-driven development, aspect-oriented programming, and software product lines. These technologies might offer new opportunities and offer new processes in the development of self-adaptive systems.

During the course of our work, we have learned that the area of self-adaptive systems is vast and multidisciplinary. Thus, it is important for software engineering to learn and borrow from other fields of knowledge that are working or have being working in the development and study of similar systems, or have already contributed solutions that fit the purpose of self-adaptive systems. We have already mentioned some of the fields, such as control theory and biology, but decision theory, non-classic computation, and computer networks may also prove to be useful. Finding a solution in one of these fields that fits our needs exactly is unlikely; however, studying a wide range of exemplars is likely to provide necessary knowledge of benchmarks, methods, techniques, and tools to solve the challenges of engineering self-adaptive software systems.

The four theses we have discussed in this paper outline new challenges that our community must face in engineering self-adapting software systems. These challenges all result from the dynamic nature of adaptation. This dynamic nature brings uncertainty that some traditional software engineering principles and techniques are the proper way to go about designing self-adaptive systems and will likely require novel solutions.

References

1. Feiler, P., Gabriel, R.P., Goodenough, J., Linger, R., Longstaff, T., Kazman, R., Klein, M., Northrop, L., Schmidt, D., Sullivan, K., Wallnau, K.: Ultra-large-scale systems: The software challenge of the future. Technical report, Software Engineering Institute (2006), `http://www.sei.cmu.edu/uls/`

2. Andersson, J., de Lemos, R., Malek, S., Weyns, D.: Towards a classification of self-adaptive software system. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525. Springer, Heidelberg (2009)
3. Seetharaman, G., Lakhotia, A., Blasch, E.P.: Unmanned Vehicles Come of Age: The DARPA Grand Challenge. Computer 39, 26–29 (2006)
4. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology 7, 125–151 (2008)
5. Jackson, M.: The meaning of requirements. Annals of Software Engineering 3, 5–21 (1997)
6. Laprie, J.C.: From dependability to resilience. In: International Conference on Dependable Systems and Networks (DSN 2008), Anchorage, AK, USA, pp. G8–G9 (2008)
7. Cheng, B.H.C., Atlee, J.M.: Research directions in requirements engineering. In: FOSE 2007: 2007 Future of Software Engineering, pp. 285–303. IEEE Computer Society, Minneapolis (2007)
8. Fickas, S., Feather, M.S.: Requirements monitoring in dynamic environments. In: IEEE International Symposium on Requirements Engineering (RE 1995), pp. 140–147 (1995)
9. Savor, T., Seviora, R.: An approach to automatic detection of software failures in realtime systems. In: IEEE Real-Time Technology and Applications Symposium, pp. 136–147 (1997)
10. Sutcliffe, A., Fickas, S., Sohlberg, M.M.: PC-RE a method for personal and context requirements engineering with some experience. Requirements Engineering Journal 11, 1–17 (2006)
11. Liaskos, S., Lapouchnian, A., Wang, Y., Yu, Y., Easterbrook, S.: Configuring common personal software: a requirements-driven approach. In: 13th IEEE International Conference on Requirements Engineering (RE 2005), pp. 9–18. IEEE Computer Society, Los Alamitos (2005)
12. Lapouchnian, A., Yu, Y., Liaskos, S., Mylopoulos, J.: Requirements-driven design of autonomic application software. In: CASCON 2006: Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research, p. 7. ACM, New York (2006)
13. Goldsby, H.J., Sawyer, P., Bencomo, N., Hughes, D., Cheng, B.H.C.: Goal-based modeling of dynamically adaptive system requirements. In: 15th Annual IEEE International Conference on the Engineering of Computer Based Systems (ECBS) (2008)
14. Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal directed requirements acquisition. In: Selected Papers of the Sixth International Workshop on Software Specification and Design (IWSSD), pp. 3–50 (1993)
15. Yu, E.S.K.: Towards modeling and reasoning support for early-phase requirements engineering. In: 3rd IEEE International Symposium on Requirements Engineering (RE 1997), Washington, DC, USA, p. 226 (1997)
16. Harel, D., Marelly, R.: Come Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer, Heidelberg (2005)
17. Zhang, J., Cheng, B.H.C.: Using temporal logic to specify adaptive program semantics. Journal of Systems and Software (JSS), Architecting Dependable Systems 79, 1361–1369 (2006)
18. Easterbrook, S., Chechik, M.: A framework for multi-valued reasoning over inconsistent viewpoints. In: Proceedings of International Conference on Software Engineering (ICSE 2001), pp. 411–420 (2001)

19. Sabetzadeh, M., Easterbrook, S.: View merging in the presence of incompleteness and inconsistency. Requirements Engineering Journal 11, 174–193 (2006)
20. Svahnberg, M., van Gurp, J., Bosch, J.: A taxonomy of variability realization techniques. Software: Practice and Experience 35, 705–754 (2005)
21. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.: A language for self-adaptive system requirement. In: SOCCER Workshop (2008)
22. Finkelstein, A.: Requirements reflection. Dagstuhl Presentation (2008)
23. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: Proceedings of International Conference on Software Engineering (ICSE 2006), Shanghai,China (2006)
24. Robinson, W.N.: Monitoring web service requirements. In: Proceedings of International Requirements Engineering Conference (RE 2003), pp. 65–74 (2003)
25. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: FOSE 2007: 2007 Future of Software Engineering, Minneapolis, MN, USA, pp. 259–268. IEEE Computer Society, Los Alamitos (2007)
26. Maes, P.: Computional reflection. PhD thesis, Vrije Universiteit (1987)
27. Kon, F., Costa, F., Blair, G., Campbell, R.H.: The case for reflective middleware. Communications of the ACM 45, 33–38 (2002)
28. Coulson, G., Blair, G., Grace, P., Joolia, A., Lee, K., Ueyama, J.: A generic component model for building systems software. ACM Transactions on Computer Systems (2008)
29. Robinson, W.: A requirements monitoring framework for enterprise systems. Requirements Engineering 11, 17–24 (2006)
30. Tanner, J.A.: Feedback control in living prototypes: A new vista in control engineering. Medical and Biological Engineering and Computing 1(3), 333–351 (1963), http://www.springerlink.com/content/rh7wx0675k5mx544/
31. Dumont, G., Huzmezan, M.: Concepts, methods and techniques in adaptive control. In: Proceedings American Control Conference (ACC 2002), Anchorage, AK, USA, vol. 2, pp. 1137–1150 (2002)
32. Brun, Y., Di Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litiou, M., Müller, H., Pezzè, M., Shaw, M.: Engineering self-adaptive systems through feedback loops. In: Cheng, B.H., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. Lecture Notes in Computer Science Hot Topics, vol. 5525 (2009)
33. Dobson, S., Denazis, S., Fernández, A., Gaïti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N., Zambonelli, F.: A survey of autonomic communications. ACM Transactions Autonomous Adaptive Systems (TAAS) 1(2), 223–259 (2006)
34. Burns, R.: Advanced Control Engineering. Butterworth-Heinemann (2001)
35. Dorf, R.C., Bishop, R.H.: Modern Control Systems, 10th edn. Prentice-Hall, Englewood Cliffs (2005)
36. Perrow, C.: Normal Accidents: Living with High-Risk Technologies. Princeton University Press, Princeton (1999)
37. Söderström, T., Stoica, P.: System Identification. Prentice-Hall, Englewood Cliffs (1988)
38. Schloss Dagstuhl Seminar 08031 Wadern, Germany: Software Engineering for Self-Adaptive Systems (2008), http://www.dagstuhl.de/08031/
39. Liu, Y., Cukic, B., Fuller, E., Yerramalla, S., Gururajan, S.: Monitoring techniques for an online neuro-adaptive controller. Journal of Systems and Software (JSS) 79(11), 1527–1540 (2006)

40. Burmester, S., Giese, H., Münch, E., Oberschelp, O., Klein, F., Scheideler, P.: Tool support for the design of self-optimizing mechatronic multi-agent systems. International Journal on Software Tools for Technology Transfer (STTT) 10 (2008) (to appear)
41. Weyns, D.: An architecture-centric approach for software engineering with situated multiagent systems. PhD thesis, Department of Computer Science, K.U. Leuven, Leuven, Belgium (2006)
42. Garlan, D., Cheng, S.W., Schmerl, B.: Increasing system dependability through architecture-based self-repair. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems. LNCS, vol. 2677. Springer, Heidelberg (2003)
43. Müller, H.A., Pezzè, M., Shaw, M.: Visibility of control in adaptive systems. In: Second International Workshop on Ultra-Large-Scale Software-Intensive Systems (ULSSIS 2008), ICSE 2008 Workshop (2008)
44. Brun, Y., Medvidovic, N.: An architectural style for solving computationally intensive problems on large networks. In: Proceedings of Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2007), Minneapolis, MN, USA (2007)
45. Brun, Y., Medvidovic, N.: Fault and adversary tolerance as an emergent property of distributed systems' software architectures. In: Proceedings of the 2nd International Workshop on Engineering Fault Tolerant Systems (EFTS 2007), Dubrovnik, Croatia, pp. 38–43 (2007)
46. Di Marzo Serugendo, G., Fitzgerald, J., Romanovsky, A., Guelfi, N.: Metaself - a framework for designing and controlling self-adaptive and self-organising systems. Technical Report BBKCS-08-08, School of Computer Science and Information Systems, Birkbeck College, London, UK (2008)
47. Cheng, S.W., Garlan, D., Schmerl, B.: Making self-adaptation an engineering reality. In: Babaoğlu, Ö., Jelasity, M., Montresor, A., Fetzer, C., Leonardi, S., van Moorsel, A., van Steen, M. (eds.) SELF-STAR 2004. LNCS, vol. 3460, pp. 158–173. Springer, Heidelberg (2005)
48. Shaw, M.: Beyond objects. ACM SIGSOFT Software Engineering Notes (SEN) 20(1), 27–38 (1995)
49. Babaoglu, Ö., Jelasity, M., Montresor, A., Fetzer, C., Leonardi, S., van Moorsel, A.P.A.: The self-star vision. In: Babaoglu, O., Jelasity, M., Montresor, A., Fetzer, C., Leonardi, S., van Moorsel, A. (eds.) SELF-STAR 2004. LNCS, vol. 3460, pp. 1–20. Springer, Heidelberg (2005)
50. Inverardi, P., Tivoli, M.: The future of software: Adaptation and dependability. In: ISSSE 2008, pp. 1–31 (2008)
51. Sama, M., Rosenblum, D., Wang, Z., Elbaum, S.: Model-based fault detection in context-aware adaptive applications. In: International Symposium on Foundations of Software Engineering (2008)
52. Liu, Y., Cukic, B., Gururajan, S.: Validating neural network-based online adaptive systems: A case study. Software Quality Journal 15(3), 309–326 (2007)
53. Hageman, J.J., Smith, M.S., Stachowiak, S.: Integration of online parameter identification and neural network for in-flight adaptive control. Technical Report NASA/TM-2003-212028, NASA (2003)
54. Kaner, C.: Software liability. Software QA 4 (1997)

Modeling Dimensions of Self-Adaptive Software Systems

Jesper Andersson[1], Rogério de Lemos[2], Sam Malek[3], and Danny Weyns[4]

[1] Department of Computer Science, Växjö University,
S-351 95 Växjö Sweden
jesper.andersson@msi.vxu.se
[2] Computing Laboratory, University of Kent,
Canterbury, Kent, CT2 7NF, UK
r.delemos@kent.ac.uk
[3] Department of Computer Science, George Mason University,
MS 4A4, 4400 University Drive, Fairfax, VA, 22030 U.S.A.
smalek@gmu.edu
[4] Departement Computerwetenschappen, Katholieke Universiteit Leuven,
Celestijnenlaan 200 A, B-3001 Leuven, Belgium
danny.weyns@cs.kuleuven.be

Abstract. It is commonly agreed that a self-adaptive software system is one that can modify itself at run-time due to changes in the system, its requirements, or the environment in which it is deployed. A cursory review of the software engineering literature attests to the wide spectrum of software systems that are described as self-adaptive. The way self-adaptation is conceived depends on various aspects, such as the users' requirements, the particular properties of a system, and the characteristics of the environment. In this paper, we propose a classification of modeling dimensions for self-adaptive software systems. Each modeling dimension describes a particular facet of the system that is relevant to self-adaptation. The modeling dimensions provide the engineers with a common set of vocabulary for specifying the self-adaptive properties under consideration and select suitable solutions. We illustrate how the modeling dimensions apply to several application scenarios.

Keywords: Self-Adaptive, Self-*, Dynamic Adaptation, Modeling.

1 Introduction

Over the past few decades we have witnessed an unrelenting pattern of growth in the size and complexity of software systems. This pattern of growth, which is very likely to continue well into the foreseeable future, has motivated software engineering researchers to develop techniques and tools that allow developers to deal with the complexity of designing, building and testing large-scale software systems. However, for the large part these advances have relied heavily on human reasoning or manual intervention.

At the same time, the emergence of highly distributed, mobile, and embedded systems that are often long-lived has made it increasingly infeasible to manually manage

B.H.C. Cheng et al. (Eds.): Self-Adaptive Systems, LNCS 5525, pp. 27–47, 2009.

and control such systems. This has prompted the development of a new class of software systems, namely self-adaptive software systems, which can modify their behavior at run-time due to changes in the system, its requirements, or the environment in which it is deployed. A cursory review of the software engineering literature attests to the wide spectrum of software systems that are argued to be self-adaptive. Indeed, there is a lack of consensus among researchers and practitioners on the points of variation among such software systems. We refer to these points of variations as *modeling dimensions*. The underlying insight guiding our study is that any self-adaptive system is built according to a conceptual model of adaptation, irrespective of the technologies and tools leveraged for its implementation. In fact, often the models of self-adaptation are represented implicitly in the form of domain knowledge or the engineer's expertise in the development of these systems. This in turn makes it harder to systematically, or even qualitatively, compare the different approaches.

In this paper, we identify modeling dimensions that describe various facets of self-adaptation, and classify these modeling dimensions in terms of four *groups*. This *classification* allows engineers to precisely specify the self-adaptive properties under consideration and select suitable solutions.

Note that it is not our ambition to be exhaustive, nor do we claim this is the only, or even the most appropriate classification. Our objective is to provide an initial impetus towards defining a comprehensive classification of key properties that are associated with self-adaptive systems. The purpose of such study is to establish a baseline from which key aspects of different self-adaptive system can be easily identified and compared. We demonstrate our classification's application in three different application domains. This exercise has served not only as a preliminary evaluation of the proposed classification, but has also helped us (the developers of these systems) to learn more about the specifics and in some cases intrinsically hidden characteristics of our systems. Finally, the classification of the modeling dimensions has aided us with identifying the current shortcomings of the state-of-the-art, which we propose to the software engineering community as future research challenges. We hope that it paves the way for focusing the future research efforts in this area.

The remainder of the paper is organized as follows. Section 2. describes an illustrative case, which serves as a motivating scenario for describing the classification of the modeling dimensions. Section 3. presents the details of the modeling dimensions. Section 4. discusses the application of the classification on two representative self-adaptive software systems. Section 5. provides an overview of open research challenges. Section 6. provides some pointers to related work. Finally, the paper concludes with a discussion of our contributions and our plans for extending this work.

2 Illustrative Case Study

As an illustrative case study, we consider the problem of obtaining dependable stock quotes from several, potentially unreliable, web sources [15]. The self-adaptation problem being considered is how to obtain reliable and available stock quotes through architectural reconfiguration. There are several web sources for stock quotes, for example, Yahoo, Google, CNN, Reuters and FT, but these sources might not be available all the time, and there are no guarantees that the values that are being provided

are correct, and moreover, their quality of services (QoS) may change. Based on the availability of resources, different fault-tolerant strategies, which rely on mechanisms, such as, voting, comparison and exception handling are employed in order to guarantee the delivery of dependable stock quotes. It is also assumed that the non-functional requirements (NFR) related to dependability may change during the system lifetime.

The system comprises (1) the application software that includes bridges for handling architectural mismatches and the fault tolerant strategies, (2) middleware that supports access to web services, and (3) the system infrastructure that includes computer hosts and the local area network. The user and the web sources for stock quotes are not considered to be part of the system.

3 Modeling Dimensions

We have grouped the identified key modeling dimensions for self-adaptive software systems into four groups: first, the dimensions associated with self-adaptability aspects of the system goals, second, the dimensions associated with causes of self-adaptation, third, the dimensions associated with the mechanisms to achieve self-adaptability, and fourth, the dimensions related to the effects of self-adaptability upon a system. Table 1 provides a summary of the modeling dimensions and their associated groups. Below we use different facets of the illustrative case study to exemplify the different modeling dimensions.

3.1 Goals

Goals are objectives the system under consideration should achieve [13]. Goals could either be associated with the lifetime of the system or with scenarios that are related to the system. Moreover, goals can either refer to the self-adaptability aspects of the application, or to the middleware or infrastructure that supports that application.

In the context of the case study mentioned above, amongst several possible goals, we consider, as an example, the following goal: "*the system shall deliver dependable (correct, responsive and available) stock quotes from the web*". This goal could be expressed in a way in which quantities are associated with the different attributes, and partitioned into sub-goals, with each sub-goal related to one of the attributes.

Evolution. This dimension captures whether the goals can change within the lifetime of the system. The number of goals may change, and the goals themselves may also change as the system as a whole evolves. Hence, goal evolution ranges from *static* in which changes are not expected, to *dynamic* in which goals can change at run-time, including the number of goals, i.e., the system is able to manage and create new goals during its lifetime.

In the context of the case study, the degree of goal evolution is static because a goal is not expected to change at run-time. However, if some stock quote providers start to charge for their services, then a new goal could be introduced to accommodate the need of the system to look for free services.

Table 1. Modeling dimensions for self-adaptive software systems

Dimensions	*Degree*	*Definition*
Goals – goals are objectives the system under consideration should achieve		
evolution	static to dynamic	whether the goals can change within the lifetime of the system
flexibility	rigid, constrained, unconstrained	whether the goals are flexible in the way they are expressed
duration	temporary to persistent	validity of a goal through the system lifetime
multiplicity	single to multiple	how many goals there are?
dependency	independent to dependent (complementary to conflicting)	how the goals are related to each other
Change – change is the cause for adaptation		
source	external (environmental), internal (application, middleware, infrastructure)	where is the source of change?
type	functional, non-functional, technological	what is the nature of change?
frequency	rare to frequent	how often a particular change occurs?
anticipation	foreseen, foreseeable, unforeseen	whether change can be predicted
Mechanisms – what is the reaction of the system towards change		
type	parametric to structural	whether adaptation is related to the parameters of the system components or to the structure of the system
autonomy	autonomous to assisted (system or human)	what is the degree of outside intervention during adaptation
organization	centralized to decentralized	whether the adaptation is done by a single component or distributed amongst several components
scope	local to global	whether adaptation is localized or involves the entire system
duration	short, medium, long term	how long the adaptation lasts
timeliness	best effort to guaranteed	whether the time period for performing self-adaptation can be guaranteed
triggering	event-trigger to time-trigger	whether the change that triggers adaptation is associated with an event or a time slot
Effects – what is the impact of adaptation upon the system		
criticality	harmless, mission-critical, safety-critical	impact upon the system in case the self-adaptation fails
predictability	non-deterministic to deterministic	whether the consequences of adaptation can be predictable
overhead	insignificant to failure	the impact of system adaptation upon the quality of services of the system
resilience	resilient to vulnerable	the persistence of service delivery that can justifiably be trusted, when facing changes

Flexibility. This dimension captures whether the goals are flexible in the way they are expressed [4]. This dimension is related to the level of uncertainty associated with the goal specification, which may range over three values: *rigid*, *constrained*, and *unconstrained*. A goal is rigid when it is prescriptive, while a goal is unconstrained when its statement provides flexibility for dealing with uncertainty. An example of a rigid goal is *"the system shall do this…"*, while an unconstrained goal is "*the system might do this…*" A constrained goal provides a middle ground, where there is flexibility as long as certain constraints are satisfied, such as, "*the system may do this… as long as it does this…*"

In the context of the case study, the goal as stated is rigid. However, if we consider a scenario in which the non-functional requirements (NFR) associated with a goal can change according to the quality of services (QoS) of the resources available, then the goal in terms of flexibility could be considered unconstrained. For example, if the NFR associated with the goal cannot be achieved, then the goal can be relaxed through some best effort analysis.

Duration. This dimension is concerned with the validity of a goal throughout the system's lifetime. It may range from *temporary* to *persistent*. While a persistent goal should be valid throughout the system's lifetime, a temporary goal may be valid for a period of time: *short*, *medium* and *long term*. A persistent goal may restrict the adaptability of the system because it may constrain the system flexibility in adapting to change. A goal that is associated with a particular scenario can be considered a temporary goal.

In terms of duration, the goal of the illustrative case can be considered persistent since it is related with the purpose of the system. On the other hand, a temporary goal could be "*the system shall deliver stock quotes more often when the volume of transactions go above a certain threshold*".

Multiplicity. This dimension is related to the number of goals associated with the self-adaptability aspects of a system. A system can either have a *single* goal or *multiple* goals. As a general rule of thumb, a single goal self-adaptive system is relatively easier to realize than systems with multiple goals. As discussed in the next dimension, this is particularly true for system where the goals are related.

In the illustrative case, since there are several NFRs associated with the system's overall objective, there are several goals that need to be satisfied. Therefore, we characterize the multiplicity dimension as multiple.

Dependency. In case a system has multiple goals, this dimension captures how the goals are related to each other. They can be either *independent* or *dependent*. A system can have several independent goals (i.e., they don't affect each other). When the goals are dependent, goals can either be *complementary* with respect to the objectives that should be achieved or they can be *conflicting*. In the latter case, tradeoffs have to be analyzed for identifying an optimal configuration of the goals to be met.

In the illustrative example, the goals that are extracted from the main objective can be considered dependent. Moreover, if cost is introduced as a NFR, then the goals can be considered as conflicting since those web sources that are able to provide better QoS might have a higher associated cost.

3.2 Change

Changes are the cause of adaptation. When there is a change in the system, its requirements, or the environment in which it is deployed, this may cause the system to self-adapt. There are changes in which the system is expected to act upon, while others can be masked from the system. Changes can be classified in terms of place in which change has occurred, the type and the frequency of the change, and whether it can be anticipated. All these elements are important for identifying how the system should react to change that occurs during run-time.

In the context of the illustrative case study mentioned above, we consider the cause of adaptation to be the failure of web sources, the reduced QoS from web sources, and changes in the NFR (expressed as goals) associated with the system.

Source. This dimension identifies the origin of the change, which can be either *external* to the system (i.e., its environment) or *internal* to the system, depending on the scope of the system. In case the source of change is internal, it might be important to identify more precisely where change has occurred: *application*, *middleware* or *infrastructure*.

The source of the two changes related to the service providers is external to the system. The change of Apache version, on which the application runs, is an internal change that happens in the middleware.

Type. This dimension refers to the nature of change. It can be *functional*, *non-functional*, and *technological*. Technological refers to both software and hardware aspects that support the delivery of the services. Examples of the three types of change are, respectively: the purpose of the system has changed and services delivered need to reflect this change, system performance and reliability need to be improved, and the version of the middleware in which the application runs has been upgraded.

In the illustrative case, since the changes are related to the QoS of the web sources, the type of change is non-functional, and the failure of a web source is also considered a non-functional change. An example of a technological change is the upgrade of the Apache version.

Frequency. This dimension is concerned with how often a particular change occurs, and it can range from *rare* to *frequent*. If for example a change happens quite often this might affect the responsiveness of the adaptation.

Failures in the web sources are expected to occur quite often, hence the frequency of change is frequent. On the other hand, if we consider changes in NFR, these should be quite rare to occur.

Anticipation. This dimension captures whether change can be predicted ahead of time. Different self-adaptive techniques are necessary depending on the degree of anticipation: *foreseen* (taken care of), *foreseeable* (planned for), and *unforeseen* (not planned for) [14].

Although faults are undesirable, they should be expected to occur, hence the failure of a web resource should be considered as foreseen. In contrast, the upgrade of the

Apache should be considered as a foreseeable change, and the provision of dependable weather forecast instead of stock quotes should be considered as unforeseen.

3.3 Mechanisms

This set of dimensions captures the system reaction towards change, which means that they are related to the adaptation process itself. The dimensions associated with this group refer to the type of self-adaptation that is expected, the level of autonomy of the self-adaptation, how self-adaptation is controlled, the impact of self-adaptation in terms of space and time, how responsive is self-adaptation, and how self-adaptation reacts to change.

In the context of the illustrative case study mentioned earlier, we consider the mechanism for self-adaptation to be the system's architectural reconfiguration in which the structure of the system is modified as a means to accommodate change.

Type. This dimension captures whether adaptation is related to the parameters of the system's components or to the structure of the system. Based on this, adaptation can be *parametric* or *structural*, or a combination of these. Structural adaptation could also be seen as compositional, since it depends on how components are integrated (e.g., dynamic weaving [20]).

The type of self-adaption considered in the illustrative case study is structural since configurations are changed and components and connectors are replaced. An example of structural adaptation is when a configuration based on majority voting has to be changed to a configuration based on comparison because of the lack of resources.

A parametric type self-adaptation would be to increase the time interval between two stock quote readings.

Autonomy. This dimension identifies the degree of outside intervention during adaptation. The range of this dimension goes from *autonomous* to *assisted*. In the autonomous case, at run-time there is no influence external to the system guiding how the system should adapt. On the other hand, a system can have a degree of self-adaptability when externally assisted, either by another system or by human participation (which can be considered another system).

In the illustrative case, for the foreseen type of changes the system is autonomous, but for the foreseeable type of changes, such as a change in the Apache version, human participation is likely to be required.

Organization. This dimension captures whether adaptation is performed by a single component – *centralized*, or distributed amongst several components – *decentralized*. If adaptation is decentralized no single component has a complete control over the system.

The self-adaptation in the case study relies on a complete model of the system, hence the organization is centralized.

Scope. This dimension identifies whether adaptation is localized or involves the entire system. The scope of adaptation can range from *local* to *global*. If adaptation affects the entire system then more thorough analysis is required to commit the adaptation. It

is fundamental for the system to be well structured in order to reduce the impact that change might have on the adaptation.

In the illustrative case, the current architectural configuration of the system and the web resource that has failed determines the scope of the self-adaptation. For instance, it may be global if it involves the reconfiguration of the whole system to come up with a new fault tolerance strategy.

Duration. This dimension refers to the period of time in which the system is self-adapting, or in other words, how long the adaptation lasts. The adaptation process can be for *short* (seconds to hours), *medium* (hours to months), or *long* (months to years) term. Note that time characteristics should be considered relative to the application domain. While scope dimension deals with the impact of adaptation in terms of space, duration deals with time.

Considering that the time it takes for architectural reconfiguration is minimal (in the scale of seconds) when compared with the lifetime of the system (months), the duration of the self-adaptation in the context of the case study should be short term.

Timeliness. This dimension captures whether the time period for performing self-adaptation can be guaranteed, and it ranges from *best-effort* to *guaranteed*. For example, in case change occurs quite often, it may be the case that it is impossible to guarantee that adaptation will take place before another change occurs, in these situations best effort should be pursued.

In the context of the case study, upper bounds on the process of architectural reconfiguration can be easily identified, hence the timeliness associated with self-adaptation can be guaranteed.

Triggering. This dimension identifies whether the change that initiates adaptation is *event-trigger* or *time-trigger*. Although it is difficult to control how and when change occurs, it is possible to control how and when the adaptation should react to a certain change. If the time period for performing adaptation has to be guaranteed, then an event-trigger might not provide the necessary assurances when change is unbounded.

In the illustrative case, the self-adaption mechanism is event-triggered, when a fault occurs, it is detected and the system starts the process of architectural reconfiguration.

3.4 Effects

This set of dimensions capture what is the impact of adaptation upon the system, that is, it deals with the effects of adaptation. While mechanisms for adaptation are properties associated with the adaptation, these dimensions are properties associated with system in which the adaptation takes place. The dimensions associated with this group refer to the criticality of the adaptation, how predictable it is, what are the overheads associated with it, and whether the system is resilient in the face of change.

In the context of the illustrative case study mentioned earlier, we consider that the system fails if it is not able to provide dependable stock quotes.

Criticality. This dimension captures the impact upon the system in case the self-adaptation fails. There are adaptations that harmless in the context of the services

provided by the system, while there are adaptations that might involve the loss of life. The range of values associated with this criticality is *harmless*, *mission-critical*, and *safety-critical*.

The level of criticality of the application (and the adaptation process) is mission-critical, since it may lead to some financial losses.

Predictability. This dimension identifies whether the consequences of self-adaptation can be predictable both in value and time. While timeliness is related to the adaptation mechanisms, predictability is associated with system. Since predictability is associated with guarantees, the degree of predictability can range from *non-deterministic* to *deterministic*.

Given that in the illustrative case there are no guarantees sufficient web sources will be available for the continued provisioning of services, the predictability of the adaptation is non-deterministic.

Overhead. This dimension captures the negative impact of system adaptation upon the system's performance. The overhead can range from *insignificant* to system *failure* (e.g., thrashing). The latter will happen when the system ceases to be able to deliver its services due to the high-overhead of running the self-adaptation processes (monitoring, analyzer, planning, effecting processes).

Since the architecture of the system that provides dependable stock quotes is based on web services, the overall overhead associated with the architectural reconfiguration is quite reasonable. In other words, although the system ceases to provide services for some time interval, this interval is acceptable.

Resilience. This dimension is related to the persistence of service delivery that can justifiably be trusted, when facing changes [14]. There are two issues that need to be considered under this dimension: first, it is the ability of the system to provide resilience, and second, it is the ability to justify the provided resilience. The degree of resilience can range from *resilient* to *vulnerable*.

In the context of the illustrative case study, the system is resilient to certain types of change (failures of web sources) because the self-adaptation which is responsible for the continuous provisioning of services can be analyzed for extracting the assurances that are needed for justifying resilience.

4 Evaluation – Case Studies

A classification framework is generally difficult to evaluate, mainly due to the process used to develop the classification. A formal evaluation, such as the one proposed by Gómez-Pérez [8], would require a formal specification of our classification framework, which is not feasible for our topic of study. Therefore, we adopt a more practical approach to validate our classification framework.

The main contribution of our work is the descriptive model of the modeling dimensions for self-adaptive software systems. The evaluation of the proposed classification was conducted through applying it to several previously developed self-adaptive software systems. The case studies represent different classes of application domains: (1) Traffic Jam Monitoring Systems [9], (2) Embedded Mobile Systems [16,17,19], and

(3) High Performance Computing and Sensor Networks [1]. The feedback from the case studies improved the classification in several iterations. Due to space constraints we only present the results of the first two studies below.

We use the classification in two different ways, In the Traffic Monitoring System, we apply the various modeling dimensions to a single scenario of self-healing. This approach provides detailed insight on one particular quality property of the system. In the Embedded mobile System, the modeling dimensions are applied to multiple QoS concerns. This approach provides insight on a set of related quality properties of the system.

4.1 Traffic Jam Monitoring System

Intelligent transportation systems (ITS) refer to systems that utilize advanced information and communication technologies to improve the safety, security and efficiency of transportation systems [6,9]. One particularly challenging problem in traffic is congestion. A first step to address this problem is monitoring the traffic. We describe an agent-based approach for traffic monitoring that enables the detection of traffic jams in a decentralized way, avoiding the bottleneck of a centralized control center. Our interest is in a particular scenario of self-healing that allows the system to deal with silent node failures (i.e., a type of failure that occurs when the failing node becomes unresponsive without sending any incorrect data). We introduce the application and briefly explain how self-healing is added to the system. Then we give an overview of the modeling dimensions for the self-healing scenario.

4.1.1 Application

The traffic monitoring system consists of a set of intelligent cameras which are distributed evenly along a highway. Each camera has a limited viewing range and cameras are placed to get an optimal coverage of the highway. A camera is able to measure the current congestion level of the traffic and decide whether there is a traffic jam or not in its viewing range. Each camera is equipped with a communication unit to interact with other cameras. The task of the cameras is to detect and monitor traffic jams on the highway in a decentralized way, i.e. without any centralized entity involved. Possible clients of the monitoring system are traffic light controllers and driver assistance systems such as systems that inform drivers about expected travel time delays. Since traffic jams can span the viewing range of multiple cameras and can dynamically

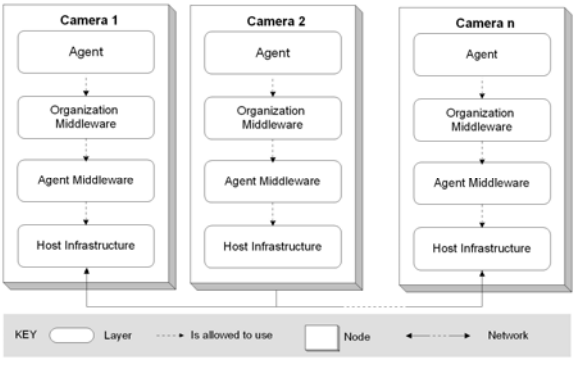


Fig. 1. Deployment view of the traffic monitoring system

grow and dissolve, the data observed by multiple cameras has to be aggregated. Without a central point of control, cameras have to collaborate and distribute the aggregated data to the clients. To support such dynamic organizations, we have applied an agent-based design for the system [9]. On each camera an agent is deployed that can play different roles in organizations. Example roles are "data pusher" and "data aggregator." Agents exploit a distributed middleware which provides support for dynamic organizations. The middleware encapsulates the management of dynamic evolution of organizations offering possible roles to agents based on the current context. Figure 1 shows the deployment view of the agent-based traffic monitoring system.

The software on each camera is structured in layers. The Host Infrastructure layer encapsulates common middleware services and basic support for distribution, hiding the complexity of the underlying hardware. The Agent Middleware layer provides basic services in multi-agent systems [22], including support for perception, action, and communication. The Organization Middleware layer provides support for dynamic organizations. The layer encapsulates the management of dynamic evolution of organizations and it provides role-specific services to the agents for perception, action, and communication. Finally the Agent layer encapsulates the agents that provide the associated functionality in the organizations for monitoring traffic jams.

4.1.2 Self-healing

When a node fails, the system may enter an inconsistent state in which agents and the organization middleware are no longer capable of working according to their specification. To deal with this kind of failures, an additional self-healing subsystem (SHS) is deployed on each node. The SHS interacts with the local agent middleware and organization middleware, and relies on the functionalities provided by the agent middleware to interact with SHSs on other nodes. Figure 2 shows the integration of the self-healing subsystem with the system software on one node.

SHSs periodically exchange alive signals using the communication service of the agent middleware (send and receive). Node failures are detected by monitoring the alive signals. When a SHS detects a failure, it adapts the local structure of the organizations in which the agent of the failing node is involved in and possibly interacts with SHSs on other involved nodes to bring the system in a consistent state from which it can continue its function in a degraded mode.

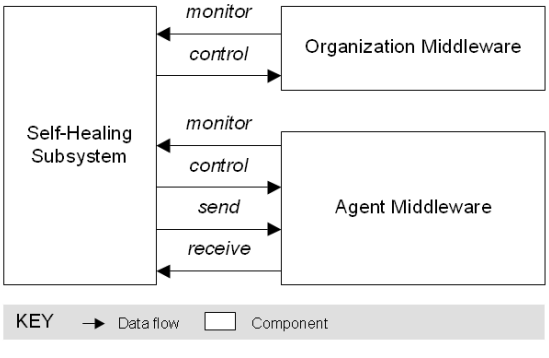


Fig. 2. Integration of the self-healing subsystem with the organization middleware and agent middleware on one node

4.1.3 Modeling Dimensions

We consider the scenario where the system is in normal operation mode (camera agents are collecting data and provide information to clients about possible traffic jams) and one of the nodes fails silently. Such event may result in corrupt organizations with lost or missing roles on the failed node. The self-healing subsystem (SHS) detects the failure and restores the system to a consistent state so that it can continue its operation.

Goals – The system shall recover from a silent node failure and continue its operation in a degraded mode.

- **Evolution:** *static* – Recovering from silent node failures is a goal that will not change over the life time of the system.
- **Flexibility:** *rigid* – A node failure compromises the consistency of the system and as such it threatens service delivery. In order to remain operational, the system must deal with node failures.
- **Duration:** *persistent* – Silent node failures can occur at any time during normal operation. As such, recovering from silent node failures is a persistent goal.
- **Multiplicity:** *multiple* – Besides dealing with node failures, the system has other goals as well. The primary goal of the system is to deliver a monitoring service to clients interested in traffic jams. Other goals refer to particular qualities of the system such as accuracy of observation and reaction time.
- **Dependency:** *dependent* – There is a dependency between the self-healing goal and the delivery of services. If the system fails to recover from a node failure, the quality of the services will significantly degrade.

Change – a node fails silently.

- **Source:** *external* (environment) and *internal* (application) – A silent node failure can be caused by an external trigger such as a hardware failure, or it can be caused by a crash of the software running on the node.
- **Type:** *technological* – The cause for self-adaptation is of a technologic nature: a node in the system fails. If the system reacts not properly, the failure will harm the system functionality.
- **Frequency:** *rare* – Silent node failures happen rarely.
- **Anticipation:** *foreseen* – Neither the place nor the time of a silent node failure can be predicted. Still, the system can anticipate how to react when a silent node failure occurs.

Mechanisms – the SHSs restore the system in a consistent state.

- **Type:** *parametric / structural* – From the point of view of a single node which is involved in a failure, the adaptation is parametric since the SHS will restore the affected local state of the system. From the point of view of the system, adaptation is structural since the changes applied by the SHSs on the nodes involved in a failure will change the structure of collaborating cameras (i.e. the failing camera will no longer be part of the collaboration).

- **Autonomy:** *autonomous* – The SHS acts fully autonomously. The self-adaptation process will take place without a human involved. However, restoring the failed node typically will require human intervention.
- **Organization:** *decentralized* – SHSs deployed on the different nodes collaborate to detect a node failure. The required adaptations are performed locally. No central monitor or controller is involved.
- **Scope:** *local* – The adaptation is performed locally. Only the nodes with cameras taking part in organizations with the camera of the failed node will be involved in the adaptation process.
- **Duration:** *short term* – The adaptation process should be completed in seconds. This is orders of magnitude faster as traffic jams arise or dissolve.
- **Timeliness:** *best effort* – The time period required for performing the adaptation depends on several factors, such as the current traffic conditions and the available bandwidth. Given the relative short duration of the adaptation (comparing to the duration of the traffic jam phenomena), best effort meets the required timeliness.
- **Triggering:** *event-trigger* – Adaptation is triggered by the detection of missing alive messages exchanged between SHSs.

Effects – the system will continue its functionality in degraded mode.

- **Criticality:** *harmless* – The services provided by the traffic monitoring system are in general not critical. If the adaptation fails, the functionality of the system may significantly degrade, however, no human lives are involved.
- **Predictability:** *deterministic* – The consequences of a node failure are clear. The information provided by the failed node will no longer be available. The SHSs will bring the system in a consistent state so that it can continue its operation.
- **Overhead:** *almost insignificant* – After adaptation, the quality of the services provided by the system will slightly degrade in case a traffic jam occurs in the neighborhood of the failed node. All traffic information collected outside the range of the camera of the failed node will not be affected.
- **Resilience:** *semi-resilient* – After adaptation, service delivery will persist with only minimal decrease of quality. In case of repeatable node failures, the quality of service delivery may become seriously affected, in particular when neighboring nodes fail.

4.2 Embedded Mobile System

Below we present the application of our classification model to another self-adaptive software system, which is representative of an emerging class of mobile, pervasive, and cyber physical systems. These systems are inherently different from traditional software systems. For instance, network failures and changes in the availability of resources are considered the norm, instead of an exception. As detailed further below, self-adaptation has been shown as a promising approach to deal with the unpredictability of such systems.

4.2.1 Application

Emergency Deployment System (EDS) is a mobile application intended for distributed management and deployment of personnel to deal with situations such as natural disasters and search-and-rescue efforts. An instance of EDS (shown in Figure 3) consists of *Headquarters*, *Team Leader*, and *Responder* applications that leverage the software services and wireless sensors provided by the system to achieve their tasks. The *Headquarters* computer is networked via secure links to a set of mobile devices used by *Leaders* during the operation. Each *Leader* is capable of controlling his own part of the crisis scene: deploying *Responders*, analyzing the deployment strategy, transferring *Responders* between *Leaders*, and so on. *Responders* can only view the segment of the operation in which they are located, receive direct orders from the *Leaders*, and report their status.

Fig. 3. An instance of EDS application

The domain of emergency and response is by its nature unpredictable. For instance, it is completely conceivable that due to some unforeseen event the *Headquarters* fail, in which case it is desirable for a designated *Leader* to assume the role of the *Headquarters*. On top of the unpredictability of the application domain, given that EDS is a mobile platform, it also needs to be able to deal with fluctuations in the availability of computing resources. For instance, the system should be able to deal with situations where as a result of user mobility the network connectivity or its throughput changes significantly.

4.2.2 Self-Adaptation

In response to the Quality of Service (QoS) challenges of mobile software systems, such as those faced by EDS, software engineers have previously developed a variety of run-time adaptation techniques, including caching [11] or hoarding [12] of data, and multi-modal components [21]. In our work [16,17,19] we have demonstrated that a software system's *deployment architecture* (i.e., allocation of the system's software components to its hardware hosts) has a significant impact on the mobile system's QoS properties. For example, a mobile system's latency can be improved if the system is deployed such that the most frequent and voluminous interactions among the components involved in delivering the functionality occur either locally or over reliable and capacious wireless network links. A key observation is that most system parameters (e.g., available bandwidth, reliability of networks, and frequency of interactions) that are required for finding a good deployment architecture are not known until run-time, and even then they can change. Therefore, a redeployment of the software system via migration of its components may be necessary to improve its QoS.

We have developed a self-adaptive infrastructure [16,17,19] for improving a mobile system's deployment architecture that consists of four phases: 1) monitoring the system parameters of interest (e.g., reliability of links, frequencies of interaction), 2) populating a deployment model of the system with the monitored system properties, 3) finding a new deployment architecture that improves the system's QoS properties, and 4) effecting the new deployment architecture via software component mobility [7]. In order to reason about multiple QoS dimensions we leverage a multivariate utility function. The utility function allows us to resolve the trade-offs among multiple QoS dimensions (e.g., when improvement in one QoS results in degradation in another QoS).

4.2.3 Modeling Dimensions

We have applied the above approach for improving a mobile software system's deployment architecture on several instances of EDS. Below we provide an analysis of this work in the context of the modeling dimensions from Section 3:

Goals – improve the users' preference, which is specified in the form of a utility function.

- **Evolution:** *dynamic* – New QoS concerns may be added, old ones may be removed or modified, and the users' preferences for the QoS concerns may change.
- **Flexibility:** *constrained* – The goal is to maximize the utility function as long as the system constraints are satisfied. An example of a system constraint is as follows: the amount of memory required for software components that are deployed on a host should be less than the amount of available memory on that host.
- **Duration:** *long term* – The system is always in pursuit of the optimal configuration.
- **Multiplicity:** *multiple* – Since most instances of EDS consist of multiple users with different roles (e.g., commanders, leaders, troops), and the fact that often there are multiple QoS dimensions of importance to each user, the goals are often multi-faceted.
- **Dependency:** *dependent* – The goals are dependent on one another. For example, a deployment architecture that maximizes the system's security often leverages complex encryption and authentication protocols, which have a negative impact on the system's latency.

Change – fluctuations in system parameters (e.g., network bandwidth, reliability) and changes in the system usage (i.e., load).

- **Source:** *external* (environment) and *internal* (application) – Source of change could be either external environment, such as a situation when a mobile user's connectivity is impacted severely due to his movement (e.g., when the user is behind thick walls). Alternatively, the source of change could be internal application, where some of the distributed components interact more frequently than others. In this case, a better deployment may be to collocate the components to minimize the amount of remote communication.
- **Type:** *non-functional* – Changes in the system could potentially degrade the level of QoS provisioned by the system.
- **Frequency:** *frequent* – System parameters are changing constantly. However, in order to avoid the system from constantly redeploying itself, changes in the

system are observed over a period of time, and only changes that are significant enough are reported to the adaptation modules.
- **Anticipation:** *foreseen* – In mobile systems changes in system parameters are the norm, rather than the exception. In EDS we foreseen such changes, and developed the appropriate mitigation capability.

Mechanisms – redeployment of software components.

- **Type:** *structural* – Through component redeployment the deployment architecture of the system is changed.
- **Autonomy:** *autonomous* – EDS is a long-lived and highly distributed system. At the same time, given that changes in such a system are frequent and unpredictable, it is infeasible for a manual control of adaptation at run-time.
- **Organization:** *centralized* analysis, *decentralized* adaptation – Finding (calculating) a new optimal deployment architecture is performed centrally, effecting the actual change (i.e., migrating and rebinding software components) is performed by individual platforms in a decentralized manner.
- **Scope:** *local* and *global* – The actual components that are redeployed depend on the results of the analysis. The result of adaptation could range from redeployment of a single software component to redeployment of the entire system.
- **Duration:** *short term* – The amount of time required to redeploy the system should be short. Redeployment impacts the availability of (some of) the system's services. This is in particular true for the EDS system that has stringent availability requirement.
- **Timeliness:** *best effort* – The time required for adaptation depends on a number of system parameters (e.g., network throughput) as well as the sizes of the software components that need to be redeployed. Therefore, it is not possible to provide any hard guarantees. For relatively stable systems it is feasible to determine a time bound.
- **Triggering:** *event-trigger* – The triggering usually depends on the patterns identified in the monitored data. If the monitored data indicates significant changes in the system, the analysis process is initiated.

Effects – system provisions its services with higher level of QoS.

- **Criticality:** *mission-critical*, *safety-critical* – Depending on the nature of emergency scenario EDS could be either a mission or safety critical system.
- **Predictability:** *non-deterministic* – An underlying assumption in EDS is that changes in the past are a good indicator of the system's future behavior. However, this assumption may not hold, in particular if the users' usages of the system's services or its parameters change dramatically.
- **Overhead:** *reasonable* – In EDS we have developed a mechanism to ensure the system does not constantly redeploy itself. This is realized by ensuring that the adaptation is triggered only if there are significant changes in the monitored data over a prespecified period of time. However, there is a considerable overhead in terms of wasted resources (e.g., battery) for the redeployment of the components, and if the components are large this overhead may be prohibitive.

- **Resilience:** *semi-resilient* – A services (functionality) becomes temporary unavailable if some of the software components involved in provisioning that service are currently being redeployed. However, there is no impact to the services that do not depend on the part of the system that is being redeployed.

5 Challenges of Modeling Self-Adaptive Systems

Substantial progress has been made by the software engineering community in tackling the challenges posed by each of the discussed modeling dimensions. However, there are several important research questions that are remaining. Our study of the modeling dimensions, in particular the exercise of applying it to several self-adaptive software systems, helped us to identify some important research questions that should be the focus of future research in this area. We briefly elaborate on those below based on the categories of the modeling dimensions.

5.1 Goals

A self-adaptive software system often needs to perform a trade-off analysis between several potentially conflicting goals. Current state-of-the-art techniques leverage a utility function to map the trade-offs among several conflicting goals to a scalar value, which is then used for making decisions about adaptation. However, in practice, defining such a utility function is a challenging task. Practical techniques for specifying and generating utility functions, potentially based on the user's requirements, are needed. One promising direction is to use preferences that compare situations under Pareto optimal conditions.

5.2 Change

Often the adaptation is triggered by the occurrence of a pattern in the data that is gathered from a running system. For example, the system is monitored to determine when a particular level of QoS is not satisfied, which then initiates the adaptation process. However, monitoring a system, especially when there are several different QoS properties of interest, has an overhead. In fact, the amount of degradation in QoS due to monitoring could outweigh the benefits of improvements in QoS to adaptation. We believe that more research on light-weight monitoring techniques, as well as more advanced models that take the monitoring overhead of the approach into account are needed.

5.3 Mechanisms

Many types of adaptation techniques have been developed: architecture-based adaptation that is mainly concerned with structural changes at the level of software components, parametric based adaptation that leverages policy files or input parameters to configure the software components, aspect-oriented-based adaptation that changes the behavior of a running system via dynamic weaving techniques. Researchers and practitioners have typically leveraged a single tactic to realize adaptation based on the characteristics of the target application. However, given the unique benefits of each

approach, we believe a fruitful avenue of future research is a more comprehensive approach that leverages several adaptation tactics simultaneously.

Most state-of-the-art adaptive systems are built according to the centralized control loop pattern. Thereby, if applied to a large-scale software system, many such techniques suffer from scalability problems. The field of multi-agent systems has developed a large body of knowledge on decentralized systems, where each agent (software component) adapts its behavior at run-time. Related are biologically inspired adaptation systems that tend to further push the limits of decentralization. While these approaches are promising, practical experiences with these approaches in real-world settings are limited. Methods used in systems engineering, like hierarchical organization and coordination schemes could also be applicable. There is a pressing need for decentralized, but still manageable, efficient, and predictable techniques for constructing self-adaptive software systems. A major challenge is to accommodate a systematic engineering approach that integrates both control-loop approaches with decentralized agent inspired approaches.

Responsiveness is a crucial property in real-time software systems, which are becoming more prevalent with the emergence of embedded and cyber-physical systems. These systems are often required to deterministically respond within pre-specified (often short) time intervals, making it extremely difficult to adapt the system, while satisfying the deadline requirements. There is a need for adaptation models targeted for real-time systems that treat the duration and overhead of adaptation as first class entities.

5.4 Effects

Adapting safety-critical software systems, while ensuring the safety requirements, has remained largely an out-of-reach goal for the practitioners and researchers. There is a need for verification and validation techniques that guarantee safe and sound adaptation of safety-critical systems, under all foreseen and unforeseen events.

Finally, predicting the exact behavior of a software system due to run-time changes is a challenging task. For example, while it may be possible to predict the new functionality that will become available as a result of replacing a software component, it may not be possible to determine what will be the impact of the replaced software component on the other components that are sharing the same resources (e.g., CPU, memory, and network). More advanced and predictive models of adaptation are needed for systems that could fail to satisfy their requirements due to side-effects of change.

In highly dynamic systems, such as mobile systems, where the environmental parameters change frequently, the overhead of adaptation due to frequent changes in the system could be so high that the system ends up thrashing. This overhead includes the frequent execution of the reasoning algorithms (e.g., finding a new configuration), the downtime of a portion of the system due to making changes, or simply the resource cost (e.g., wasted CPU cycles, battery power) of changing the system. The trade-offs between the adaptation overhead and the accrued benefits of changing the system needs to be taken into consideration for such systems.

6 Related Work

This work defines a classification of modeling dimensions that should be considered when modeling self-adaptive software systems. Similar classifications exist but our survey reveals that none is suitable for characterizing the modeling variations among self-adaptive software systems. Dobson et al. provide a survey of techniques applied to autonomic computing [5]. Buckley et al. define a taxonomy for software change [3], which unlike our approach is not focused on run-time adaptation (change) of software. Another major difference is that goals are not made explicit in Buckeley's taxonomy. Implementation of adaptability requires support from middleware or languages, Bradbury et al. classify support from dynamic software architecture languages [2]. This work has a focus on architecture specification and does not consider the goals. Similarly, the taxonomy by McKinley et al. [18] specifically targets compositional software adaptation, and is not applicable to other types of self-adaptive software. The work by Laprie on a classification of resilience [14] has also inspired some of the dimensions and their values in our work.

7 Discussions and Future Work

The classification model presented in this paper applies to any self-adaptive software system. We believe that this classification will be useful in several different development situations. It can be used as a driver for traditional forward engineering, but also useful in a reverse engineering context where engineers comprehend the existing solutions.

The classification brings more structure to the area of self-adaptive software systems. With the classification in mind it is more likely that an individual engineer as well as groups of engineers to be able to understand the technology domain better, thus avoiding situations where two or more interpretations of a technique affect the development process. Our intention with the classification has been to create a vocabulary that can be used within, for instance, a design team.

The classification can also be used to drive development. Despite its rather high level of abstraction, the groups, dimensions, and degrees can be used as a requirements statement for the self-adaptive scenarios in a system. This information supports decision making about tools, languages, and middleware platforms.

The classification could also be utilized in reverse engineering activities. Part of this process is "understanding structures" that are currently present in a self-adaptive application, and then go forward and change. The classification provides a check-list for conceptual and physical concepts concerned with structural and behavioral properties that can be identified in existing application documentation, hence assist in classifying an application's self-adaptive behavior.

While our experience with the classification model has been positive so far, we believe the classification model can be refined further. In particular, we would like to provide a more detailed enumeration of possible values for the classification's degree attribute (i.e., the middle column of Table 1). We also hypothesize that the majority of self-adaptive software systems are developed according to a handful of architectural patterns. We intend to leverage the proposed classification model, which allows for

systematically identifying the variations among different self-adaptive software systems, to study and document the most commonly used architectural patterns for such systems.

Acknowledgments

This work is partially supported by grant CCF-0820060 from the National Science Foundation. Danny Weyns is funded by the Research Foundation Flanders (FWO).

References

1. Andersson, J., et al.: An Adaptive High-Performance Service Architecture. In: ETAPS Workshop on Software Composition Electronic Notes Theoretical Computer Science, vol. 114 (2005)
2. Bradbury, J.S., et al.: A Survey Of Self-Management In Dynamic Software Architecture Specifications. In: Garlan, D., Kramer, J., Wolf, A. (eds.) ACM SIGSOFT Workshop on Self-Managed Systems (WOSS 2004), pp. 28–33 (2004)
3. Buckley, J., Mens, T., Zenger, M., Rashid, A., Kniesel, G.: Towards A Taxonomy of Software Change. Journal of Software Maintenance and Evolution, 309–332 (September 2005)
4. Cheng, B.H.C., et al.: Software Engineering for Self-Adaptive Systems: A Research Road Map. In: Cheng, B.H.C., et al. (eds.) Software Engineering for Self-Adaptive Systems. 08031 Dagstuhl Seminar. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2008)
5. Dobson, S., Denazis, S., et al.: A Survey of Autonomic Communications. ACM Transactions on Autonomous and Adaptive Systems 1(2), 223–259 (2006)
6. ERTICO. Intelligent Transportation Systems for Europe, http://www.ertico.com/
7. Fuggetta, A., Picco, G.P., Vigna, G.: Understanding Code Mobility. IEEE Trans. on Software Engineering 24, 342–361 (1998)
8. Gómez-Pérez, A.: Evaluation of Ontologies. International Journal of Intelligent Systems 16, 391–409 (2001)
9. Haesevoets, R., et al.: Managing Agent Interactions with Context-Driven Dynamic Organizations. In: Weyns, D., Brueckner, S.A., Demazeau, Y. (eds.) EEMMAS 2007. LNCS, vol. 5049, pp. 166–186. Springer, Heidelberg (2008)
10. ITS. Intelligent Transportation Society of America, http://www.itsa.org/
11. Kistler, J.J., Satyanarayanan, M.: Disconnected Operation in the Coda File System. ACM Transactions on Computer Systems 10(1) (Feburary 1992)
12. Kuenning, G.H., Popek, G.J.: Automated Hoarding for Mobile Computers. In: ACM Symp. on Operating Systems Principles, St. Malo, France (October 1997)
13. van Lamsweerde, A.: Goal-Oriented Requirements Engineering: A Guided Tour. In: IEEE International Symposium on Requirements Engineering, Toronto, Canada (August 2001)
14. Laprie, J.C.: From Dependability to Resilience. In: International Conference on Dependable Systems & Networks (DSN 2008), Anchorage, Alaska, June 2008, pp. G8–G9 (2008)
15. de Lemos, R.: Architecting Web Services Applications for Improving Availability. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems III. LNCS, vol. 3549, pp. 69–91. Springer, Heidelberg (2005)

16. Malek, S., et al.: A Framework for Ensuring and Improving Dependability in Highly Distributed Systems. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems III. LNCS, vol. 3549, pp. 173–193. Springer, Heidelberg (2005)
17. Malek, S., Seo, C., Ravula, S., Petrus, B., Medvidovic, N.: Reconceptualizing a Family of Heterogeneous Embedded Systems via Explicit Architectural Support. In: International Conference on Software Engineering (ICSE 2007), Minneapolis, Minnesota (May 2007)
18. Mckinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C.: Composing Adaptive Software. IEEE Computer 37(7), 56–64 (2004)
19. Mikic-Rakic, M., Malek, S., Medvidovic, N.: Architecture-Driven Software Mobility in Support of QoS Requirements. In: International Workshop on Software Architectures and Mobility (SAM), Leipzig, Germany (May 2008)
20. Popovici, A., et al.: Dynamic Weaving for Aspect-oriented Programming. In: International Conference on Aspect-Oriented Software Development (AOSD 2002), Enschede, Netherlands, April 2002, pp. 141–147 (2002)
21. Weinsberg, Y., Ben-Shaul, I.: A Programming Model and System Support for Disconnected-Aware Applications on Resource-Constrained Devices. In: International Conference on Software Engineering (ICSE 2002), Orlando, FL (2002)
22. Weyns, D., et al.: Environments for multiagent systems, state-of-the-art and research challenges. In: Weyns, D., Van Dyke Parunak, H., Michel, F. (eds.) E4MAS 2004. LNCS, vol. 3374, pp. 1–47. Springer, Heidelberg (2005)

Engineering Self-Adaptive Systems through Feedback Loops

Yuriy Brun[1], Giovanna Di Marzo Serugendo[2], Cristina Gacek[3], Holger Giese[4], Holger Kienle[5], Marin Litoiu[6], Hausi Müller[5], Mauro Pezzè[7], and Mary Shaw[8]

[1] University of Southern California, Los Angeles, CA, USA
ybrun@usc.edu
[2] Birkbeck, University of London, London, UK
dimarzo@dcs.bbk.ac.uk
[3] University of Newcastle upon Tyne, Newcastle upon Tyne, UK
cristina.gacek@ncl.ac.uk
[4] Hasso Plattner Institute at the University of Potsdam, Germany
holger.giese@hpi.uni-potsdam.de
[5] University of Victoria, British Columbia, Canada
{kienle,hausi}@cs.uvic.ca
[6] York University and IBM Canada Ltd., Canada
marin@ca.ibm.com
[7] University of Milano Bicocca, Italy and University of Lugano, Switzerland
mauro.pezze@unisi.ch
[8] Carnegie Mellon University, Pittsburgh, PA, USA
mary.shaw@cs.cmu.edu

Abstract. To deal with the increasing complexity of software systems and uncertainty of their environments, software engineers have turned to self-adaptivity. Self-adaptive systems are capable of dealing with a continuously changing environment and emerging requirements that may be unknown at design-time. However, building such systems cost-effectively and in a predictable manner is a major engineering challenge. In this paper, we explore the state-of-the-art in engineering self-adaptive systems and identify potential improvements in the design process.

Our most important finding is that in designing self-adaptive systems, the feedback loops that control self-adaptation must become first-class entities. We explore feedback loops from the perspective of control engineering and within existing self-adaptive systems in nature and biology. Finally, we identify the critical challenges our community must address to enable systematic and well-organized engineering of self-adaptive and self-managing software systems.

1 Introduction

The complexity of current software systems and uncertainty in their environments has led the software engineering community to look for inspiration in diverse related fields (e.g., robotics, artificial intelligence, control theory, and biology) for new ways to design and manage systems and services [1,2,3,4]. In

B.H.C. Cheng et al. (Eds.): Self-Adaptive Systems, LNCS 5525, pp. 48–70, 2009.

this endeavor, the capability of the system to adjust its behavior in response to the environment in the form of self-adaptation has become one of the most promising research directions. The "self" prefix indicates that the systems decide autonomously (i.e., without or with minimal interference) how to adapt or organize to accommodate changes in their contexts and environments. While some self-adaptive system may be able to function without any human intervention, guidance in the form of higher-level objectives (e.g., through policies) is useful and realized in many systems.

The landscapes of software engineering domains and computing environments are constantly evolving. In particular, software has become the bricks and mortar of many complex systems (i.e., a system composed of interconnected parts that as a whole exhibits one or more properties (behaviors among the possible properties) not obvious from the properties of the individual parts). The hallmarks of such complex or ultra-large-scale (ULS) systems [5] are self-adaptation, self-organization, and emergence [6]. Engineers in general, and software engineers in particular, design systems according to requirements and specifications and are not accustomed to regulating requirements and orchestrating emergent properties. Ottino argues that the landscape is bubbling with activity and engineers should be at the center of these developments and contribute new theories and tools [6].

In order for the evolution of software engineering techniques to keep up with these ever-changing landscapes, software engineers must innovate in the realm of building, running, and managing software systems. Software-intensive systems must be able to adapt more easily to their ever-changing surroundings and be flexible, fault-tolerant, robust, resilient, available, configurable, secure, and self-healing. Ideally, and necessarily for sufficiently large systems, these adaptations must happen autonomously. The research community that has formed around self-adaptive systems has already generated many encouraging results, helping to establish self-adaptive systems as a significant, interdisciplinary, and active research field.

Self-adaptive systems have been studied within the different research areas of software engineering, including requirements engineering [7], software architecture [8,9], middleware [10], and component-based development [11]; however, most of these initiatives have been isolated. Other research communities that have also investigated self-adaptation and feedback from their own perspectives are even more diverse: control theory, control engineering, artificial intelligence, mobile and autonomous robots, multi-agent systems, fault-tolerant computing, dependable computing, distributed systems, autonomic computing, self-managing systems, autonomic communications, adaptable user interfaces, biology, distributed artificial intelligence, machine learning, economic and financial systems, business and military strategic planning, sensor networks, or pervasive and ubiquitous computing. Over the past decade several self-adaptation-related application areas and technologies have grown in importance. It is important to emphasize that in all these initiatives software has become the common element

that enables the provision of self-adaptability. Thus, it is imperative to investigate systematic software engineering approaches for developing self-adaptive systems, which are—ideally—applicable across multiple domains.

Self-adaptive systems can be characterized by how they *operate* or how they are *analyzed*, and by multiple dimensions of properties including *centralized* and *decentralized*, *top-down* and *bottom-up*, *feedback latency* (slow vs. fast), or *environment uncertainty* (low vs. high). A top-down self-adaptive system is often centralized and operates with the guidance of a central controller or policy, assesses its own behavior in the current surroundings, and adapts itself if the monitoring and analysis warrants it. Such a system often operates with an explicit internal representation of itself and its global goals. By analyzing the components of a top-down self-adaptive system, one can compose and deduce the behavior of the whole system. In contrast, a cooperative self-adaptive system or self-organizing system is often decentralized, operates without a central authority, and is typically composed bottom-up of a large number of components that interact locally according to simple rules. The global behavior of the system *emerges* from these local interactions. It is difficult to deduce properties of the global system by analyzing only the local properties of its parts. Such systems do not necessarily use internal representations of global properties or goals; they are often inspired by biological or sociological phenomena.

Most engineered and nature-inspired self-adaptive systems fall somewhere between these two extreme poles of self-adaptive system types. In practice, the line between these types is rather blurred and compromises will often lead to an engineering approach incorporating techniques from both of these two extreme poles. For example, ULS systems embody both top-down and bottom-up self-adaptive characteristics (e.g., the Web is basically decentralized as a global system, but local sub-webs are highly centralized or server farms are both centralized and decentralized) [5].

Building self-adaptive software systems cost-effectively and in a predictable manner is a major engineering challenge. New theories are needed to accommodate, in a systematic engineering manner, traditional top-down approaches and bottom-up approaches. A promising starting point to meet these challenges is to mine suitable theories and techniques from control engineering and nature and to apply those when designing and reasoning about self-adaptive software systems. Control engineering emphasizes feedback loops, elevating them to first-class entities [12,13]. In this paper we argue that feedback loops are also essential for understanding all types of self-adaptive systems.

Over the years, the discipline of software engineering strongly emphasized the static architecture of a system and, to a certain extent, neglected the dynamic aspects. In contrast, control engineering emphasized the dynamic feedback loops embedded in a system and its environment and neglected the static architecture. A notable exception is the seminal paper by Magee and Kramer on dynamic structure in software architecture [14], which formed the foundation for many subsequent research projects [9,15,16,17]. However, while these research projects realized feedback systems, the actual feedback loops were hidden or abstracted.

Feedback loops have been recognized as important factors in software process management and improvement or software evolution. For example, the feedback loops at every stage in Royce's waterfall model [18] or the risk feedback loop in Boehm's spiral model [19] are well known. Lehman's work on software evolution showed that "the software process constitutes a multilevel, multiloop feedback system and must be treated as such if major progress in its planning, control, and improvement is to be achieved." Therefore, any attempt to make parts of this "multiloop feedback system" self-adaptive necessarily also has to consider feedback loops [20].

With the proliferation of self-adaptive software systems, it is imperative to develop theories, methods and tools around feedback loops. Mining the rich experiences and theories from control engineering as well as taking inspiration from nature and biology where we can find systems that adapt in rather complex ways, and then adapting and applying the findings to software-intensive self-adaptive systems is a most worthwhile and promising avenue of research.

In the remainder of this paper, we therefore investigate *feedback loops* as a key aspect of engineering self-adaptive systems. Section 2 outlines basic principles of feedback loops and demonstrates their importance and potential benefits for understanding self-adaptive systems. Sections 3 and 4 describe control engineering and biologically inspired approaches for self-adaptation. In Section 5, we present selected challenges for the software engineering community in general and the SEAMS community in particular for engineering self-adaptive computing systems.

2 The Role of Feedback Loops

Self-adaptation in software-intensive systems comes in many different guises. What self-adaptive systems have in common is that design decisions are moved towards runtime to control dynamic behavior and that an individual system reasons about its state and environment. For example, keeping web services up and running for a long time requires collecting information that reflects the current state of the system, analyzing that information to diagnose performance problems or to detect failures, deciding on how to resolve the problem (e.g., via dynamic load-balancing or healing), and acting to effect the planning decisions made.

Feedback loops provide the generic mechanism for self-adaptation. Positive feedback occurs when an initial change in a system is reinforced, which leads toward an amplification of the change. In contrast, negative feedback triggers a response that counteracts a perturbation. Further, natural environments with synergistic and antagonistic relationships between components sometimes produce more complex forms of feedback loops that can neither be classified as positive nor negative feedback.

2.1 Generic Feedback Loop

A feedback loop typically involves four key activities: collect, analyze, decide, and act. Sensors or probes collect data from the executing system and its context

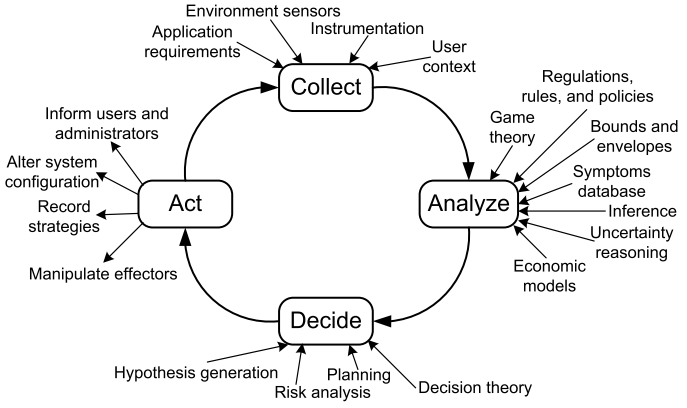


Fig. 1. Autonomic control loop [21]

about its current state. The accumulated data are then cleaned, filtered, and pruned and, finally, stored for future reference to portray an accurate model of past and current states. The diagnosis then analyzes the data to infer trends and identify symptoms. Subsequently, the planning attempts to predict the future to decide on how to act on the executing system and its context through actuators or effectors.

This generic model of a feedback loop, often referred to as the autonomic control loop as depicted in Figure 1 [21], focuses on the activities that realize feedback. This model is a refinement of the AI community's sense-plan-act approach of the early 1980s to control autonomous mobile robots [22,23]. While this model provides a good starting point for our discussion of feedback loops, it does not detail the flow of data and control around the loop. However, the flow of control among these components is unidirectional. Moreover, while the figure shows a single control loop, multiple separate loops are typically involved in a practical system.

When engineering a self-adaptive system, questions about these properties become important. The feedback cycle starts with the *collection* of relevant data from environmental sensors and other sources that reflect the current state of the system. Some of the engineering questions that need be answered here are: What is the required sample rate? How reliable is the sensor data? Is there a common event format across sensors? Do the sensors provide sufficient information for system identification?

Next, the system *analyzes* the collected data. There are many approaches to structuring and reasoning about the raw data (e.g., using models, theories, and rules). Some of the applicable questions here are: How is the current state of the system inferred? How much past state may be needed in the future? What data need to be archived for validation and verification? How faithful will the model be to the real world and whether an adequate model can be obtained from the available sensor data? How stable will the model be over time?

Next, a *decision* must be made about how to adapt the system in order to reach a desirable state. Approaches such as risk analysis help in choosing among various alternatives. Here, the important questions are: How is the future state of the system inferred? How is a decision reached (e.g., with off-line simulation, utility/goal functions, or system identification)? What are the priorities for self-adaptation across multiple feedback loops and within a single feedback loop?

Finally, to implement the decision, the system must *act* via available actuators or effectors. Important questions that arise here are: When should and can the adaptation be safely performed? How do adjustments of different feedback loops interfere with each other? Do centralized or decentralized feedback help achieve the global goal? An important additional applicable question is whether the control system has sufficient command authority over the process—that is, whether the available actuators or effectors are sufficient to drive the system into the desired directions.

The above questions—and many others—regarding the feedback loops should be explicitly identified, recorded, and resolved during the development of a self-adaptive system.

2.2 Feedback Loops in Control Engineering

An obvious way to address some of the questions raised above is to draw on control theory. Feedback control is a central element of control theory, which provides well-established mathematical models, tools, and techniques for analysis of system performance, stability, sensitivity, or correctness [24,25]. The software engineering community in general and the SEAMS community in particular are exploring the extent to which general principles of control theory (i.e., feedforward and feedback control, observability, controllability, stability, hysteresis, and specific control strategies) are applicable when reasoning about self-adaptive software systems.

Control engineers have invented many variations of control and adaptive control. For many engineering disciplines, these types of control systems have been the bread and butter of their designs. While the amount of software in these control systems has increased steadily over the years, the field of software engineering has not embraced feedback loops as a core design element. If the computing pioneers and programming language designers were control engineers—instead of mathematicians—by training, modern programming paradigms might feature process control elements [26].

We now turn our attention to the generic data and control flow of a feedback loop. Figure 2 depicts the classical feedback control loop featured in numerous control engineering books [24,25]. Due to the interdisciplinary nature of control theory and its applications (e.g., robotics, power control, autopilots, electronics, communication, or cruise control), many diagrams and variable naming conventions are in use. The system's goal is to maintain specified properties of the output, y_p, of the process (also referred to as the plant or the system) at or sufficiently close to given reference inputs u_p (often called set points). The process output y_p may vary naturally; in addition external perturbations d may disturb

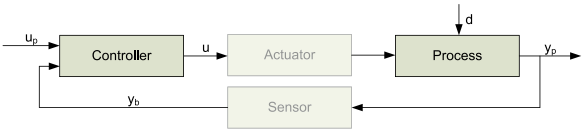


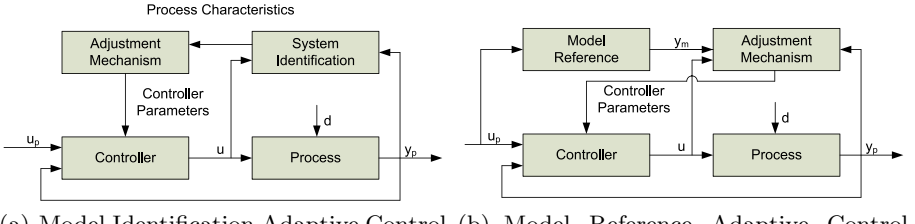
Fig. 2. Feedback control loop

the process. The process output y_p is fed back by means of sensors—and often through additional filters (not shown in Figure 2)—as y_b to compute the difference with the reference inputs u_p. The controller implements a particular control algorithm or strategy, which takes into account the difference between u_p and y_b to decide upon a suitable correction u to drive y_p closer to u_p using process-specific actuators. Often the sensors and actuators are omitted in diagrams of the control loops for the sake of brevity.

The key reason for using feedback is to reduce the effects of uncertainty which appear in different forms as disturbances or noise in variables or imperfections in the models of the environment used to design the controller [27]. For example, feedback systems are used to manage QoS in web server farms. Internet load, which is difficult to model due to its unpredictability, is one of the key variables in such a system fraught with uncertainty.

It seems prudent for the SEAMS community to investigate how different application areas realize this generic feedback loop, point out commonalities, and evaluate the applicability of theories and concepts in order to compare and leverage self-adaptive software-intensive systems research. To facilitate this comparison, we now introduce the organization of two classic feedback control systems, which are long established in control engineering and also have wide applicability.

Adaptive control in control theory involves modifying the model or the control law of the controller to be able to cope with slowly occurring changes of the controlled process. Therefore, a second control loop is installed on top of the main controller. This second control loop adjusts the controller's model and operates much slower than the underlying feedback control loop. For example, the main feedback loop, which controls a web server farm, reacts rapidly to bursts of Internet load to manage QoS. A second slow-reacting feedback loop



(a) Model Identification Adaptive Control (MIAC)

(b) Model Reference Adaptive Control (MRAC)

Fig. 3. Two standard schemes for adaptive feedback control loops

may adjust the control law in the controller to accommodate or take advantage of anomalies emerging over time.

Model Identification Adaptive Control (MIAC) [28] and *Model Reference Adaptive Control* (MRAC) [27], depicted in Figures 3(a) and 3(b), are two important manifestations of adaptive control. Both approaches use a reference model to decide whether the current controller model needs adjustment. The MIAC strategy builds a dynamical reference model by simply observing the process without taking reference inputs into account. The MRAC strategy relies on a predefined reference model (e.g., equations or simulation model) which includes reference inputs.

This MIAC system identification element takes the control input u and the process output y_p to infer the model of the current running process (e.g., its unobservable state). Then, the element provides the system characteristics it has identified to the adjustment mechanism which then adjusts the controller accordingly by setting the controller parameters. This adaptation scheme has to take also into account that a disturbances d might affect the process behavior and, thus, usually has to observe the process for multiple control cycles before initiating an adjustment of the controller.

The MRAC solution, originally proposed for the flight-control problem [27,29], is suitable for situations in which the controlled process has to follow an elaborate prescribed behavior described by the model reference. The adaptive algorithm compares the outputs of the process y_p which results from the control value u of the Controller to the desired responses from a reference model y_m for the goal u_p, and then adjusts the controller model by setting controller parameters to improve the fit in the future. The goal of the scheme is to find controller parameters that cause the combined response of the controller and process to match the response of the reference model despite present disturbances d.

The MIAC control scheme observes only the process to identify its specific characteristics using its input u and output y_p. This information is used to adjust the controller model accordingly. The MRAC control scheme in contrast provides the desired behavior of the controller and process together using a model reference and the input u_p. The adjustment mechanism compares this to y_p. The MRAC scheme is appropriate for achieving robust control if a solid and trustworthy reference model is available and the controller model does not change significantly over time. The MIAC scheme is appropriate when there is no established reference model but enough knowledge about the process to identify the relevant characteristics. The MIAC approach can potentially accommodate more substantial variations in the controller model.

Feedback loops of this sort are used in many engineered devices to bring about desired behavior despite undesired disturbances [24,27,28,29]. Hellerstein et al. provide a more detailed treatment of the analysis capabilities offered by control theory and their application to computing systems [2,13]. As pointed out by Kokar et al. [30], rather different forms of control loops may be employed for self-adaptive software and we may even go beyond classical or even adaptive control and use reconfigurable control for the software where besides the parameters also structural changes are considered (cf. compositional adaptation [31]).

2.3 Feedback Loops in Natural Systems

In contrast to self-adaptive systems built using control engineering concepts, self-adaptive systems in nature do not often have a single clearly visible control loop. Often, there is no clear separation between the controller, the process, and the other elements present in advanced control schemes. Further, the systems are often highly decentralized in such a way that the entities have no sense of the global goal but rather it is the interaction of their local behavior that yields the global goal as an emergent property.

Nature provides plenty of examples of cooperative self-adaptive and self-organizing systems: social insect behaviors (e.g., ants, termites, bees, wasps, or spiders), schools of fish, flocks of birds, immune systems, and social human behavior. Many cooperative self-adaptive systems in nature are far more complex than the systems we design and build today. The human body alone is orders of magnitude more complex than our most intricate designed systems. Further, biological systems are decentralized in such a way that allows them to benefit from built-in error correction, fault tolerance, and scalability. When encountering malicious intruders, biological systems typically continue to execute, often reducing performance as some resources are rerouted towards handling those intruders (e.g., when the flu virus infects a human, the immune system uses energy to attack the virus while the human continues to function). Despite added complexity, human beings are more resilient to failures of individual components and injections of malicious bacteria and viruses than engineered software systems are to component failure and computer virus infection. Other biological systems, for example worms and sea stars, are capable of recovering from such serious hardware failures as being cut in half (both worms and sea stars regenerate the missing pieces to form two nearly identical organisms), yet we envision neither a functioning laptop computer, half of which was crushed by a car, nor a machine that can recover from being installed with only half of an operating system. It follows that if we can extract certain properties of biological systems and inject them into our software design process, we may be able to build complex and dependable self-adaptive software systems. Thus, identifying and understanding the feedback loops within natural systems is critical to being able to design nature-mimicking self-adaptive software systems.

Two types of feedback in nature are positive and negative feedback. Positive feedback reinforces a perturbation in systems in nature and leads to an amplification of that perturbation. For example, ants lay down a pheromone that attracts other ants. When an ant travels down a path and finds food, the pheromone attracts other ants to the path. The more ants use the path, the more positive feedback the path receives, encouraging more and more ants to follow the path to the food. Negative feedback triggers a response that counteracts a perturbation. For example, when the human body experiences a high concentration of blood sugar, it releases insulin, resulting in glucose absorption, and bringing the blood sugar back to the normal concentration.

Negative and positive feedback combine to ensure system stability: positive feedback alone would push the system beyond its limits and ultimately out of

control, whereas negative feedback alone prevents the system from searching for optimal behavior.

Decentralized self-organizing systems are generally composed of a large number of simple components that interact locally — either directly or indirectly. An individual component's behavior follows internal rules based only on local information. These rules can support positive and negative feedback at the level of individual components. The numerous interactions among the components then lead to global control loops.

2.4 Feedback Loops in Software Engineering

For software engineering we have observed that feedback loops are often hidden, abstracted, dispersed, or internalized when the architecture of an adaptive system is documented or presented [26]. Certainly, common software design notations (e.g., UML) do not routinely provide views that lend themselves to describing and analyzing control and reason about uncertainty. Further, we suspect that the lack of a notation leads to the absence of an explicit task to document the control, which leads in turn in failure to explicitly designing, analyzing, and validating the feedback loops.

However, the feedback behavior of a self-adaptive system, which is realized with its control loops, is a crucial feature and, hence, should be elevated to a first-class entity in its modeling, design, implementation, validation, and operation. When engineering a self-adaptive system, the properties of the control loops affect the system's design, architecture, and capabilities. Therefore, besides making the control loops explicit, the control loops' properties have to be made explicit as well. Garlan et al. also advocate to make self-adaptation external, as opposed to internal or hard-wired, to separate the concerns of system functionality from the concerns of self-adaptation [9,16].

Explicit feedback loops are common in software process improvement models [19] and industrial IT service management [32], where the system management activities and products are decoupled by the software development cycle. A major breakthrough in making feedback loops explicit came with IBM's autonomic computing initiative [33] with its emphasis on engineering self-managing systems. One of the key findings of this research initiative is the blueprint for building autonomic systems using MAPE-K (monitor-analyze-plan-execute over a knowledge base) feedback loops [34] as depicted in Figure 4. The phases of the MAPE-K loop or autonomic element map readily to the generic autonomic control loop as depicted in Figure 1. Both diagrams highlight the main activities of the feedback loop while abstracting away characteristics of the control and data flow around the loop. However, the blueprint provides extensive instructions on how to architect and implement the four phases, the knowledge bases, sensors, and actuators. It also outlines how to compose autonomic elements to orchestrate self-management.

Software engineering for self-adaptive systems has recently received considerable attention with a proliferation of journals, conferences, workshops (e.g., TASS, SASO, ICAC, or SEAMS). Many of the papers published in these venues

dealing with the development, analysis and validation methods for self-adaptive systems do not yet provide sufficient explicit focus on the feedback loops, and their associated properties, that almost inevitably control the self-adaptations.

The idea of increasing the visibility of control loops in software architectures and software methods is not new. Over a decade ago, Shaw compared a software design method based on process control to an object-oriented design method [35]. She introduced a new software organization paradigm based on control loops with an architecture that is dominated by feedback loops and their analysis rather than by the identification of discrete stateful objects. Hellerstein et al. in their ground-breaking book provide a first practical treatment of the design and application of feedback control of computing systems [13]. Recently, Shaw, together with Müller and Pezzè, advocated the usefulness of a design paradigm based on explicit control loops for the design of ULS systems [26]. The preliminary ideas presented in this position paper contributed to ignite the discussion that led to the contribution of this paper.

To manage uncertainty in computing systems and their environments, we need to introduce feedback loops to control the uncertainty. To reason about uncertainty effectively, we need to elevate feedback loops to be visible and first class. If we do not make the feedback loops visible, we also will not be able to identify which feedback loops may have major impact on the overall system behavior and apply techniques to predict their possible severe effects. More seriously, we will neglect the proof obligations associated with the feedback, such as validating that y_b (i.e., the estimate of y_p derived from the sensors) is sufficiently good, that the control strategy is appropriate to the problem, that all necessary corrections can be achieved with the available actuators, that corrections will preserve global properties such as stability, and that time constraints will be satisfied. Therefore, if feedback loops are not visible we will not only fail to understand these systems but also fail to build them in such a manner that crucial properties for the adaptation behavior can be guaranteed.

ULS systems may include many self-adaptive mechanisms developed independently by different working teams to solve several classes of problems at different abstraction levels. The complexity of both the systems and the development processes may result in the impossibility of coordinating the many self-adaptive mechanisms by design, and may result in unexpected interactions with negative effects on the overall system behavior. Making feedback loops visible is an essential step toward the design of distributed coordination mechanisms that can prevent undesirable system characteristics—such as various forms of instability and divergence—due to interactions of competing self-adaptive systems.

3 Solutions Inspired by Explicit Control

The *autonomic element*—introduced by Kephart and Chess [33] and popularized with IBM's architectural blueprint for autonomic computing [34]—is the first architecture for self-adaptive systems that explicitly exposes the feedback control loop depicted in Figure 2 and the steps indicated in Figure 1, identifying

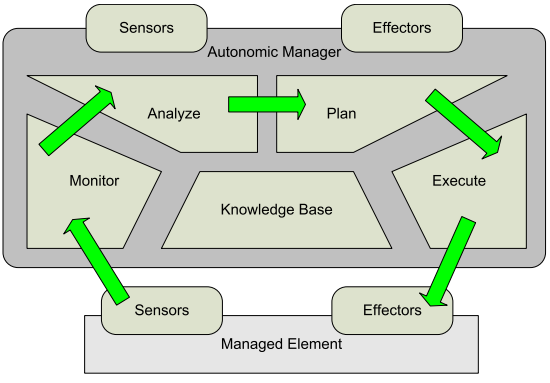


Fig. 4. IBM's autonomic element [34]

functional components and interfaces for decomposing and managing the feedback loop. To realize an autonomic system, designers compose arrangements of collaborating autonomic elements working towards common goals. In particular, IBM uses the autonomic element as a fundamental building block for realizing self-configuring, self-healing, self-protecting and self-optimizing systems [33,34].

An autonomic element, as depicted in Figure 4, consists of a managed element and an autonomic manager with a feedback control loop at its core. Thus, the autonomic manager and the managed element correspond to the controller and the process, respectively, in the generic feedback loop. The manager or controller is composed of two manageability interfaces, the sensor and the effector, and the monitor-analyze-plan-execute (MAPE-K) engine consisting of a monitor, an analyzer, a planner, and an executor which share a common knowledge base. The monitor senses the managed process and its context, filters the accumulated sensor data, and stores relevant events in the knowledge base for future reference. The analyzer compares event data against patterns in the knowledge base to diagnose symptoms and stores the symptoms for future reference in the knowledge base. The planner interprets the symptoms and devises a plan to execute the change in the managed process through its effectors. The manageability interfaces, each of which consists of a set of sensors and effectors, are standardized across managed elements and autonomic building blocks, to facilitate collaboration and data and control integration among autonomic elements. The autonomic manager gathers measurements from the managed element as well as information from the current and past states from various knowledge sources and then adjusts the managed element if necessary through its manageability interface according to its control objective.

An autonomic element itself can be a managed element [34,36]. In this case additional sensors and effectors at the top of the autonomic manager are used to manage the element (i.e., provide measurements through its sensors and receive control input—rules or policies—through its effectors). If there are no such effectors, then the rules or policies are hard-wired into the control loop. Even

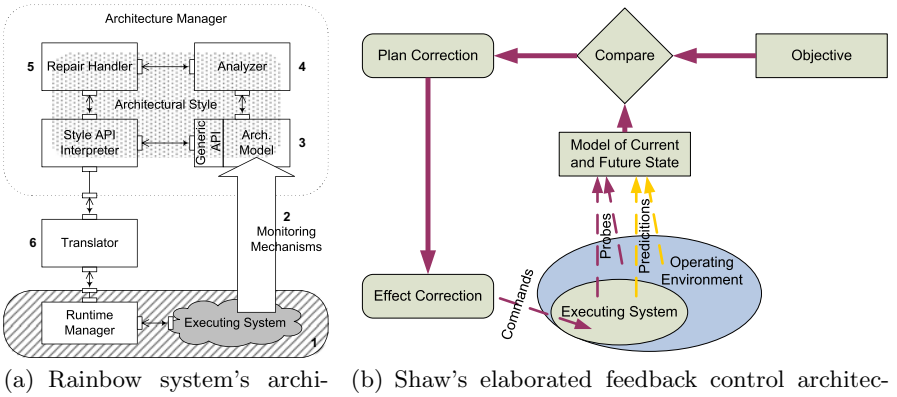
if there are no effectors at the top of the element, the state of the autonomic element is typically still exposed through its top sensors. Thus, an autonomic element constitutes a self-adaptive system because it alters the behavior of an underlying subsystem—the managed element—to achieve the overall objectives of the system.

While the autonomic element, as depicted in Figure 4, was originally proposed as a solution for architecting self-managing systems for autonomic computing [33], conceptually, it is in fact a feedback control loop from classic control theory.

Garlan et al. have developed a technique for using feedback for self-repair of systems [9]. Figure 5(a) shows their system. They add an external controller (top box) to the underlying system (bottom box), which is augmented with suitable actuators. Their architecture maps quite naturally to the generic feedback control loop (cf. Figure 2).

To see this, Figure 5(b) introduces two elaborations to the generic control loop. First, we separate the controller into three parts (compare, plan correction, and effect correction). Second, we elaborate the value of y_b, showing that sensors can sense both the executing system and its operating environment and by explicitly adding a component to convert observations to modeled value. In redrawing the diagram, we have arranged the components so that they overlay the corresponding components of the Rainbow architecture diagram. To show that the feedback loop is clearly visible in the Rainbow architecture, we provide the mapping between both architectures in Table 1.

An example for a self-adaptive system following the MIAC scheme applied to software is the robust feedback loop used in self-optimization that is becoming prevalent in performance-tuning and resource-provisioning scenarios (cf. Figure 6(a)) [37,38]. Robust feedback control tolerates incomplete knowledge about the system model and assumes that the system model has to be frequently rebuilt. To accomplish this, the feedback control includes an Estimator



(a) Rainbow system's architecture [9]

(b) Shaw's elaborated feedback control architecture [26]

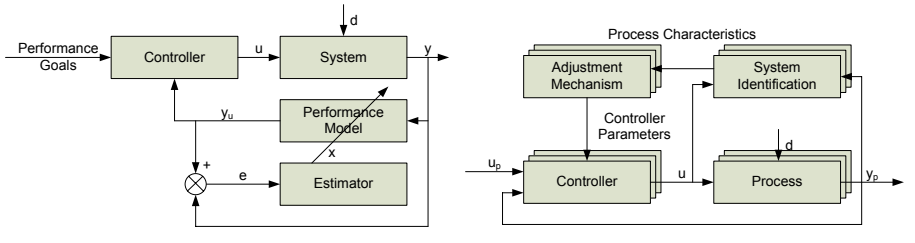
Fig. 5. The Rainbow system and Shaw's feedback control loop

Table 1. Mapping showing the correspondence between the elements of the Rainbow system's architecture and Shaw's architecture

Rainbow system (cf. Figure 5(a))	Model (cf. Figure 5(b))
(1) Executing sys. with runtime manager	Executing sys. in its operating environment
(2) Monitoring mechanisms	Probes
(–) (Rainbow is not predictive)	Predictions
(3) Architectural model	Objective, model of current state
(4) Analyzer	Compare
(5) Repair handler	Plan correction
(6) Translator, runtime manager	Effect correction, commands

that estimates state variables (x) that cannot be directly observed. The variables x are then used to tune a performance model (i.e., queuing network model) online, allowing that model to provide a quantitative dependency law between the performance outputs and inputs (y_u) of the system around an operational point. This dependence is dynamic and captures the influence of perturbations w, as well as time variations of different parameters in the system (e.g., due to software aging, caching, or optimizations). A Controller uses the y_u dependency to decide when and what resources to tune or provision. The Controller uses online optimization algorithms to decide what adaptation to perform. Since performance is affected by many parameters, the Controller chooses to change only those parameters that achieve the performance goals with minimum resource consumption. Since the system changes in time, so does the performance dependence between the outputs and inputs. However, the scheme still works because the Estimator and the Performance Model provide the Controller with an accurate reflection of the system.

Another example of an MIAC scheme is a mechatronics system consisting of a system of autonomous shuttles that operate on demand and in a decentralized manner using a wireless network [39]. Realizing such a mechatronics system makes it necessary to draw from techniques offered by the domains of control engineering as well as software engineering. Each shuttle that travels along a specific track section approaches the responsible local section control to obtain data about the track characteristics. The shuttle optimizes the control behavior for passing that track section based on that data and the specific characteristics of the shuttle. The new experiences are then propagated back to the section control



(a) Self-optimization via feedback loop (b) Multiple intertwined MIAC schemes

Fig. 6. Applications of adaptive control schemes for self-adaptive systems

such that other shuttles may benefit from them (i.e., improve their model). The shuttles implement the MIAC control loop as a group as depicted in Figure 6(b). Each shuttle traveling along a track only realizes the Adjustment Mechanism and Controller while the shuttles, which have reported on the track characteristics before, and the section control collectively realize the System Identification. Note that this constitutes a form of cooperative self-adaptation where multiple elements are involved in a single adaptive control loop.

4 Solutions Inspired by Natural Systems

Because nature-inspired engineering is a younger area of research than control theory, emerging nature-inspired solutions for self-adaptive software have not yet been classified and contrasted against one another. In this section, we present some of the existing nature-inspired solutions for self-adaptive software systems. While some current work deals with building biologically inspired self-adapting software systems [40,41], even more work has gone into studying biological systems to inspire the design of software and hardware systems in robotics [1,42,43,44]. It remains a challenge to employ biological knowledge to develop an understanding of how to build software systems that function the way biological systems do, and to design appropriate architectures, design tools, and programming tools to create such systems.

In nature, the process of crystal growth can result in well-formed regular crystals or high-error irregular crystals. The key aspect that determines which type of crystal will form is the speed at which the crystal grows. If the crystal grows slowly, then badly and weakly attached molecules detach from the crystal and the final result has very few, if any, errors. However, if the crystal grows quickly, badly attached molecules are locked in by other attachments before they can detach, and the final result has many errors. The tile architectural style [40], a software architectural style inspired by crystal growth, leverages the feedback exhibited by crystal growth to allow fault and adversary tolerance [4]. The tile style allows distributing computation of NP-complete problems on a large network in a secure, dependable, and scalable manner [40]. The control loops within tile-style systems are difficult to classify as positive or negative; however, they do fit nicely into the feedback loop described in Figure 2. The individual components attach to one another, *collecting* information on what other components may attach. After a faulty or malicious agent attaches an illegal component, future attachment cannot happen, and *analysis* reveals that the assembly became locally "stuck." (Since the system is only affected locally, the computational resources are rerouted and the system as a whole makes progress, incurring only negligible reduction in computation speed.) The surrounding components *decide* to detach a few most-recently attached neighbors, and resume attaching new components. As this *action* selects prospective components at random, faulty or malicious agents are unlikely to be able to penetrate the assembly two or more times, thus resulting in a fault- and adversary-tolerant software system.

Schools of fish and flocks of birds adapt their behavior by using direct communication. They follow a set of attraction and repulsion rules: maintaining a

minimum distance from other objects in the environment, matching own velocity with that of neighbors, and moving toward the perceived center of mass in one's neighborhood. These rules provoke a wave of reactions that are communicated progressively to all components of the school or flock. Certain software systems use similar mechanisms—for example, process schedulers and network routing protocols.

In contrast, ants and wasps use stigmergy as an indirect communication mechanism by leaving clues in the environment for the others. Ants add pheromone to their environments to denote paths to food and wasp-nest construction follows a work-in-progress mechanism (each cell added to the nest creates a new nest configuration and each configuration triggers a particular response in the wasps). Research in swarm robotics has used stigmergy extensively to solve static and dynamic optimization problems. More generally, stigmergy as an indirect communication medium is being used for coordinating unmanned vehicles [45].

Mammalian immune systems provide a defense mechanism by detecting antigens (intruders) and by coordinating a collective decentralized response to destroy them. These distributed, decentralized systems balance detection and removal of malicious agents against interference with normal cell processes and employ learning techniques. Software intrusion detection research already leverages some immune-system ideas [46], but understanding these intricate self-organizing defenses can offer much more insight into engineering self-adaptive systems.

More generally, current practice in engineering self-organizing systems encompass the use of autonomous components (agents), establishment of behavior interactions rules following adaptive mechanisms inspired by nature or use of middleware with built-in features supporting adaptive mechanisms (such as digital pheromone propagation).

5 Challenges Ahead

We have argued that the feedback loop should be a first-class entity when thinking about engineering of self-adaptive systems. We believe that understanding and reasoning about the control loop is key for advancing the construction of self-adaptive systems from an ad-hoc, trial-and-error endeavor towards a more disciplined approach. To achieve this goal, the following issues, possibly among others, have to be addressed.

Modeling: There should be modeling support to make the control loop explicit and to expose self-adaptive properties so that the designer can reason about the system. The models have to capture what can be observed and what can be influenced. It would be desirable to have a widely agreed upon standard (e.g., in the form of reference models with domain-specific notations) for self-adaptive systems including the control loop. Highly decentralized self-organizing systems, such as swarms, need to have proper models of control loops, even though today, that control loop is only implicitly present in the models.

The nature of a self-adaptive system requires to reify properties that would otherwise be encoded implicitly. These reified properties need to be modeled

appropriately so that they can be queried and modified during runtime. Examples of such properties are system state that is used to reason about the system's behavior, and policies and business goals that govern and constrain how the system can and will adapt.

Control Loops: We have described a number of types of control loops found in control-engineering, natural, robotics, and software systems. Our list is by no means comprehensive and other types of control loops, and self-adaption techniques that leverage control loops and interactions between control loops, exist. One challenge to advancing the engineering of self-adaptive software systems is creating a reference library of control-loop types and mechanisms of control-loop interactions. To create this library, we must mine, understand, and leverage existing systems and then classify and catalog their self-adaptation mechanisms. In particular, natural systems are rich sources of distinct and novel control loops and control-loop interactions.

Architecture and Design: Decisions concerning feedback loops in the architecture and design of self-adaptive systems can leverage past experience. Control theory research found that systems with a single control loop are easier to reason about than systems with multiple loops, although the latter are far more common. Since good engineering practice calls for simple design, engineers should consider minimizing the number of control loops or decoupling control loops from each other. Such a decoupling can happen with respect to time, ensuring that the loops operate at different time scales, or with respect to space, weakening the dependencies between variables. When complete decoupling is not possible, the design must make the control-loop interactions, and their handling, explicit. Thus, designs containing multiple control loops must be carefully considered and analyzed, as in, for example, the MIAC or MRAC designs (cf. Figures 3(b) and 3(b)).

Control engineering research has also identified that hierarchical organization of control loops reduces the design complexity. In this scheme, the loops influence each other top-down and operate at different time scales, avoiding unexpected interference between the hierarchy levels. Hierarchical organization is of particular interest if it is possible to distinguish different time scales and different controlled variables [37] or different adaptation domains within a software system, such as change management and goal management [17].

Reference architectures for adaptive systems should therefore highlight key aspects of feedback loops, including their number, structural arrangements (e.g., sequential, parallel, hierarchical, decentralized), interactions, data flow, tolerances, trade-offs, sampling rates, stability and convergence conditions, hysteresis specifications, and context uncertainty [36]. It is highly desirable that such architectures can be used to reason about the properties of the system and its control loop. In other words, we must determine whether it is possible to build Attribute-Based Architectural Styles for control loops in self-adaptive systems [47].

Unintended-Interaction Detection: For some systems (e.g., ULS systems), their complexity may limit the possibility of hierarchical organization of control loops or other methods of control-loop decoupling. Control loops developed independently to cope with different problems at various abstraction levels may result in unexpected interactions with negative effects on the system behavior.

Combining distinct subsystems with seemingly independent goals can often result in emergent behaviors, some of which can be desirable, while others are undesirable. Nature suggests two avenues of research toward potential solutions to detecting and avoiding unintended interactions between control loops: (1) mining the abundant systems from nature with control loops that do not interact in undesirable ways to understand how to develop such control loops in engineered systems; and (2) understanding the process that has arrived at systems with only desirably interacting control loops (e.g., evolution) and applying a similar process to select or decouple control loops automatically in engineered systems.

Maintenance: Since maintenance constitutes a significant portion of a software system's life cycle, understanding maintainability concerns specific to self-adaptive systems poses an important challenge. Examples of issues that should be tackled are how the maintainability concerns of self-adaptive systems and traditional systems compare and whether a system designed for dynamic variability or adaptation is easier to maintain than a static system [36,48]. It is reasonable to expect differences in maintenance of the two kinds of systems because some self-adaptive systems add a reflective layer that enables runtime analysis and adaptation. Consequently, a maintenance activity may involve changes to either, or both, the system's *meta level* or *base level* [49].

Middleware Support: Currently, the building of self-adaptive systems is tedious because of the lack of a reusable code base. A dedicated development and execution environment (e.g., in the form of a framework or library) for building systems with self-adaptive features would go a long way in resolving this challenge. As a vision, good middleware support should "allow researchers with different motivations and experiences to put their ideas into practice, free from the painful details of low-level system implementation" [50]. Such an infrastructure should define standardized interfaces and services, support different heterogeneous platforms, allow for the rapid prototyping of self-adaptive features, and involve hybrid architectures (e.g., combining top-down and bottom-up or centralized and decentralized approaches).

Verification and Validation: Development of self-adaptive systems requires techniques to validate the effects of feedback loops. Classical control engineering provides sophisticated solutions for the analysis of continuous-feedback control loops [2,13]. However, some phenomena relevant to software and self-adaptation have a discrete or hybrid nature (e.g., architectural changes). In addition to control engineering, discrete event systems [51], switched systems [52], and hybrid systems [53] may provide mechanisms relevant to self-adaptive systems.

Reengineering: Today, most engineering issues for self-adaptive systems are approached from the perspective of greenfield development. However, many legacy applications can benefit from self-adaptive features. Reengineering of existing systems with the goal of making them more self-adaptive in a cost-effective and principled manner poses an important challenge. Of particular concern is the question of how to inject a control loop into an existing system. Technologies and tools that allow an engineer to (semi-)automatically augment an existing system with sensors and effectors can begin to answer this challenge. Further, existing systems should be gradually migrated towards self-adaptive capabilities (a.k.a. the chicken little approach), for example, by increasing the scope of self-adaptive control from subcomponents towards the entire business infrastructure or by increasing self-adaptive functionality in a single component by substituting high-level-goal-based self-adaptive behavior for manual configuration.

Human-Computer Interaction: Even though self-adaptive systems act autonomously in many respects, they have to keep the user in the loop. Providing the user with feedback about the system state is crucial to establish and keep users' trust. To that effect, a self-adaptive system needs to expose aspects of its control loop to the user. For example, if a web server is reconfigured in response to a load change, the human administrator needs (visual) feedback that the performed adaptation has a positive effect.

Also, users should be given the option to disable self-adaptive features and the system should take care not to contradict explicit choices made by users [54]. Furthermore, users might want feedback from the system about the information collected by sensors and how this information is used to adapt the system. In fact, if the collected information is personal data there might be even a legal obligation to do so [55].

6 Conclusions

We have outlined in this paper that feedback loops are a key factor in software engineering of self-adaptive systems. In the case of top-down self-adaptive architectures employing explicitly-engineered feedback control loops, these loops are of paramount importance to guide engineering of the self-adaptive part of those systems. In case of systems inspired by biological and natural systems, identifying the feedback loops and understanding their impact is essential. While notions of the feedback loop that can be found in the areas of control theory and natural systems can provide valuable insight, software engineering needs to develop its own unique notion of feedback loop that is suitably aligned with its own problem domain. Therefore, we argue for the necessity of well-founded approaches for the models, architectures, design, implementation, maintenance, and verification techniques of self-adaptation, while taking into account the notion of reengineering existing systems to contain self-adaptation. We think that aligning our efforts with the key concept of feedback loops, which has been somewhat ignored in software engineering, will bring our community closer to the goal

of building complex self-adapting systems. Satisfying the challenges we outlined above is the first step toward this endeavor.

Acknowledgments

This paper is the result of stimulating discussions among the authors and other participants during the seminar on Software Engineering for Self-Adaptive Systems at Schloss Dagstuhl in January 2008. Some of the ideas developed in this paper have initially been presented elsewhere [26,36,56].

References

1. Abelson, H., Allen, D., Coore, D., Hanson, C., Homsy, G., Thomas, F., Knight, J., Nagpal, R., Rauch, E., Sussman, G.J., Weiss, R.: Amorphous computing. Communications of the ACM 43(5), 74–82 (2000)
2. Diao, Y., Hellerstein, J.L., Parekh, S., Griffith, R., Kaiser, G., Phung, D.: Control theory foundation for self-managing computing systems. IEEE Journal on Selected Areas in Communications 23(12), 2213–2222 (2005)
3. Di Marzo-Serugendo, G., Gleizes, M.P., Karageorgos, A.: Self-organisation in MAS. Knowledge Engineering Review 20(2), 165–189 (2005)
4. Brun, Y., Medvidovic, N.: Fault and adversary tolerance as an emergent property of distributed systems' software architectures. In: 2nd ACM International Workshop on Engineering Fault Tolerant Systems (EFTS 2007), Dubrovnik, Croatia, pp. 38–43 (2007)
5. Feiler, P., Gabriel, R.P., Goodenough, J., Linger, R., Longstaff, T., Kazman, R., Klein, M., Northrop, L., Schmidt, D., Sullivan, K., Wallnau, K.: Ultra-large-scale systems: The software challenge of the future. Technical report, Software Engineering Institute (2006), http://www.sei.cmu.edu/uls/
6. Ottino, J.M.: Engineering complex systems. Nature 427(6973), 399–400 (2004)
7. Brown, G., Cheng, B.H., Goldsby, H., Zhang, J.: Goal-oriented specification of adaptation requirements engineering in adaptive systems. In: ACM 2006 International Workshop on Self-Adaptation and Self-Managing Systems (SEAMS 2006), Shanghai, China, pp. 23–29 (2006)
8. Richter, U., Mnif, M., Branke, J., Müller-Schloer, C., Schmeck, H.: Towards a generic observer/controller architecture for organic computing. In: Hochberger, C., Liskowsky, R. (eds.) INFORMATIK 2006: Informatik für Menschen. GI-Edition – Lecture Notes in Informatics, vol. P-93, pp. 112–119. Gesellschaft für Informatik (2006)
9. Garlan, D., Cheng, S.W., Schmerl, B.: Increasing system dependability through architecture-based self-repair. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems. LNCS, vol. 2677. Springer, Heidelberg (2003)
10. Liu, H., Parashar, M.: Accord: a programming framework for autonomic applications. IEEE Transactions on Systems, Man, and Cybernetics 36(3), 341–352 (2006)
11. Peper, C., Schneider, D.: Component engineering for adaptive ad-hoc systems. In: ACM 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2008), Leipzig, Germany, pp. 49–56 (2008)
12. Tanner, J.A.: Feedback control in living prototypes: A new vista in control engineering. Medical and Biological Engineering and Computing 1(3), 333–351 (1963), http://www.springerlink.com/content/rh7wx0675k5mx544/

13. Hellerstein, J.L., Diao, Y., Parekh, S., Tilbury, D.M.: Feedback Control of Computing Systems. John Wiley & Sons, Chichester (2004)
14. Magee, J., Kramer, J.: Dynamic structure in software architectures. In: 4th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE 1996), San Francisco, CA, USA, pp. 3–14. ACM Press, New York (1996)
15. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An architecture-based approach to self-adaptive software. IEEE Intelligent Systems 14(3), 54–62 (1999)
16. Cheng, S.W., Garlan, D., Schmerl, B.: Making self-adaptation an engineering reality. In: Babaoğlu, Ö., Jelasity, M., Montresor, A., Fetzer, C., Leonardi, S., van Moorsel, A., van Steen, M. (eds.) SELF-STAR 2004. LNCS, vol. 3460, pp. 158–173. Springer, Heidelberg (2005)
17. Kramer, J., Magee, J.: Self-managed systems: An architectural challenge. In: Future of Software Engineering (FOSE 2007), Minneapolis, MN, USA, pp. 259–268. IEEE Computer Society, Los Alamitos (2007)
18. Royce, W.W.: Managing the development of large software systems. In: 9th ACM/IEEE International Conference on Software Engineering (ICSE 1970), pp. 328–338 (1970)
19. Boehm, B.W.: A spiral model of software development and enhancement. IEEE Computer 21(5), 61–72 (1988)
20. Lehman, M.M.: Software's future: Managing evolution. IEEE Software 15(1), 40–44 (1998)
21. Dobson, S., Denazis, S., Fernández, A., Gaïti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N., Zambonelli, F.: A survey of autonomic communications. ACM Transactions Autonomous Adaptive Systems (TAAS) 1(2), 223–259 (2006)
22. Nilsson, N.J.: Principles of Artificial Intelligence. Tioga Press, Palo Alto (1980)
23. Gat, E.: Three-layer Architectures, pp. 195–210. MIT/AAAI Press, Cambridge (1997)
24. Burns, R.: Advanced Control Engineering. Butterworth-Heinemann (2001)
25. Dorf, R.C., Bishop, R.H.: Modern Control Systems, 10th edn. Prentice-Hall, Englewood Cliffs (2005)
26. Müller, H.A., Pezzè, M., Shaw, M.: Visibility of control in adaptive systems. In: Second International Workshop on Ultra-Large-Scale Software-Intensive Systems (ULSSIS 2008), Workshop at 30th IEEE/ACM International Conference on Software Engineering (ICSE 2008), Leipzig, Germany (May 2008)
27. Astrom, K., Wittenmark, B.: Adaptive Control, 2nd edn. Addison-Wesley, Reading (1995)
28. Söderström, T., Stoica, P.: System Identification. Prentice-Hall, Englewood Cliffs (1988)
29. Dumont, G., Huzmezan, M.: Concepts, methods and techniques in adaptive control. In: 2002 IEEE American Control Conference (ACC 2002), Anchorage, AK, USA, vol. 2, pp. 1137–1150 (2002)
30. Kokar, M.M., Baclawski, K., Eracar, Y.A.: Control theory-based foundations of self-controlling software. IEEE Intelligent Systems 14(3), 37–45 (1999)
31. McKinley, P.K., Sadjadi, M., Kasten, E.P., Cheng, B.H.: Composing adaptive software. IEEE Computer 37(7), 56–64 (2004)
32. Brittenham, P., Cutlip, R.R., Draper, C., Miller, B.A., Choudhary, S., Perazolo, M.: IT service management architecture and autonomic computing. IBM Systems Journal 46(3), 565–581 (2007)
33. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer 36(1), 41–50 (2003)

34. IBM Corporation: An architectural blueprint for autonomic computing. White Paper, 4th edn., IBM Corporation, http://www-03.ibm.com/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf
35. Shaw, M.: Beyond objects. ACM SIGSOFT Software Engineering Notes (SEN) 20(1), 27–38 (1995)
36. Müller, H.A., Kienle, H.M., Stege, U.: Autonomic computing: Now you see it, now you don't. In: Lucia, A.D., Ferrucci, F. (eds.) Software Engineering: International Summer Schools, ISSSE 2006-2008, Salerno, Italy, Revised Tutorial Lectures. LNCS, vol. 5413, pp. 32–54. Springer, Heidelberg (2009)
37. Litoiu, M., Woodside, M., Zheng, T.: Hierarchical model-based autonomic control of software systems. In: ACM ICSE Workshop on Design and Evolution of Autonomic Software, St. Louis, MO, USA, pp. 1–7 (2005)
38. Litoiu, M., Mihaescu, M., Ionescu, D., Solomon, B.: Scalable adaptive web services. In: Development for Service Oriented Architectures (SD-SOA 2008), Workshop at 30th IEEE/ACM International Conference on Software Engineering (ICSE 2008), Leipzig, Germany (2008)
39. Burmester, S., Giese, H., Münch, E., Oberschelp, O., Klein, F., Scheideler, P.: Tool support for the design of self-optimizing mechatronic multi-agent systems. International Journal on Software Tools for Technology Transfer (STTT) 10(3) (2008)
40. Brun, Y., Medvidovic, N.: An architectural style for solving computationally intensive problems on large networks. In: Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2007), Workshop at 29th IEEE/ACM International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA (2007)
41. Di Marzo-Serugendo, G., Fitzgerald, J., Romanovsky, A., Guelfi, N.: A generic framework for the engineering of self-adaptive and self-organising systems. Technical report, School of Computer Science, University of Newcastle, Newcastle, UK (2007)
42. Nagpal, R.: Programmable Self-Assembly: Constructing Global Shape Using Biologically-Inspired Local Interactions and Origami Mathematics. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA (2001)
43. Clement, L., Nagpal, R.: Self-assembly and self-repairing topologies. In: Workshop on Adaptability in Multi-Agent Systems, First RoboCup Australian Open (AORC 2003), Sydney, Australia (2003)
44. Shen, W.M., Krivokon, M., Chiu, H., Everist, J., Rubenstein, M., Venkatesh, J.: Multimode locomotion via superbot reconfigurable robots. Autonomous Robots 20(2), 165–177 (2006)
45. Sauter, J.A., Matthews, R., Parunak, H.V.D., Brueckner, S.A.: Performance of digital pheromones for swarming vehicle control. In: 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005), The Netherlands, pp. 903–910. ACM, New York (2005)
46. Hofmeyr, S., Forrest, S.: Immunity by design: An artificial immune system. In: Genetic and Evolutionary Computation Conference (GECCO 1999), Orlando, Florida, USA, pp. 1289–1296. Morgan-Kaufmann, San Francisco (1999)
47. Klein, M., Kazman, R.: Attribute-based architectural styles. Technical Report CMU/SEI-99-TR-022, Software Engineering Institute (SEI) (1999), http://www.sei.cmu.edu/pub/documents/99.reports/pdf/99tr022.pdf
48. Zhu, Q., Lin, L., Kienle, H.M., Müller, H.A.: Characterizing maintainability concerns in autonomic element design. In: 24th IEEE International Conference on Software Maintenance (ICSM 2008), Beijing, China, pp. 197–206 (2008)

49. Andersson, J., de Lemos, R., Malek, S., Weyns, D.: Reflecting on self-adaptive software systems. In: 2009 International Workshop on Self-Adaptation and Self-Managing Systems (SEAMS 2009), Vancouver, BC, Canada (to be published, 2009)
50. Babaoglu, Ö., Jelasity, M., Montresor, A., Fetzer, C., Leonardi, S., van Moorsel, A.P.A.: The self-star vision. In: Babaoğlu, Ö., Jelasity, M., Montresor, A., Fetzer, C., Leonardi, S., van Moorsel, A., van Steen, M. (eds.) SELF-STAR 2004. LNCS, vol. 3460, pp. 1–20. Springer, Heidelberg (2005)
51. Passino, K.M., Burgess, K.L.: Stability analysis of discrete event systems. Adaptive and Learning Systems for Signal Processing Communications, and Control. John Wiley & Sons, Inc., New York (1998)
52. Liberzon, D., Morse, A.: Basic problems in stability and design of switched systems. IEEE Control Systems Magazine 19(5), 59–70 (1999)
53. Decarlo, R.A., Branicky, M.S., Pettersson, S., Lennartson, B.: Perspectives and Results on the Stability and Stabilizability of Hybrid Systems. Proceedings of the IEEE 88(7), 1069–1082 (2000)
54. Lightstone, S.: Seven software engineering principles for autonomic computing development. Innovations in Systems and Software Engineering 3(1), 71–74 (2007)
55. Sackmann, S., Strüker, J., Accorsi, R.: Personalization in privacy-aware highly dynamic systems. Communications of the ACM 49(9), 32–38 (2006)
56. Cheng, B.H., de Lemos, R., Giese, H., et al.: Software engineering for self-adaptive systems: A research roadmap. In: Cheng, B.H., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525. Springer, Heidelberg (2009)

Improving Architecture-Based Self-Adaptation through Resource Prediction

Shang-Wen Cheng, Vahe V. Poladian, David Garlan, and Bradley Schmerl

Carnegie Mellon University, School of Computer Science
5000 Forbes Avenue, Pittsburgh, PA 15213
{zensoul,vahe.poladian,garlan,schmerl}@cs.cmu.edu

Abstract. An increasingly important concern for modern systems design is how best to incorporate self-adaptation into systems so as to improve their ability to dynamically respond to faults, resource variation, and changing user needs. One promising approach is to use architectural models as a basis for monitoring, problem detection, and repair selection. While this approach has been shown to yield positive results, current systems use a reactive approach: they respond to problems only when they occur. In this paper we argue that self-adaptation can be improved by adopting an anticipatory approach in which predictions are used to inform adaptation strategies. We show how such an approach can be incorporated into an architecture-based adaptation framework and demonstrate the benefits of the approach.

Keywords: self-adaptation, resource prediction, autonomic computing, software architecture.

1 Introduction

As computing systems become more and more integral to our daily activities, it becomes increasingly important for those systems to provide reliable and uninterrupted service, even in the presence of system faults, changing resources and loads, and different user needs. In the past this capability has largely been provided through human oversight. As a result, the cost of managing such systems has grown to 70-90% of the total cost of system ownership [8], while the burden of managing the many aspects of computing has surpassed the capacity of human attention [24].

In response there has been considerable recent interest in supporting automated system self-adaptation, whereby the system takes increasing responsibility for dynamically detecting problems and repairing itself. Most systems that support self-adaptation adopt a control systems perspective: a system is monitored and the resulting observations are used to determine system health, and the system is adapted to fix any existing problems.

One particularly promising form of this approach is to use architectural models of a system as the basis for problem detection, diagnosis, and repair. Architecture-based self-adaptation has had considerable success in providing adaptation support for

B.H.C. Cheng et al. (Eds.): Self-Adaptive Systems, LNCS 5525, pp. 71–88, 2009.

legacy systems and in providing flexibility for tailoring adaptation to business needs [1,6,10,11,12,13,19,25].

One outstanding problem with such systems is that they are strictly reactive: they respond to the current environment and system state, invoking adaptation strategies if and only if an immediate problem arises. The goal of a reactive approach is to select an adaptation that optimizes the instantaneous utility of the system at that time. However, from a global perspective, several instantaneously optimal decisions may be sub-optimal when considered together. For example, if we adapt a web system reactively to a short, temporary spike in bandwidth by reducing the fidelity of the content, this may be sub-optimal in hindsight because a short delay may be less offensive to the client than low fidelity.

In this paper we argue that self-adaptation can be dramatically improved if we use *future predictions* of the environment, and specifically its resources, to make better choices about whether and how to adapt a system. In other work [21], we have developed a resource prediction framework that provides predictions on resource availability from a variety of prediction models, in the context of continually adapting ubiquitous computing. We can use this framework to provide predictive information to help architecture-based self-adaptation. In particular, we observe that prediction offers four kinds of improvement to the existing self-adaptation approach:

1. Prediction prevents unnecessary self-adaptation.
2. Prediction reduces disruption from incremental adaptation, for example, enlisting servers 4 at once rather than one at a time.
3. Prediction enables pre-adaptation to seasonal behavior.
4. Prediction improves overall choice of adaptation.

At first glance, it seems obvious that using predicted information will improve self-adaptation – if you know it is going to rain, don't turn on the sprinklers. But, making choices about when and how to consider this predicted information is crucially important. Accordingly, the contributions of this chapter are:

1. A framework for generic use of predictive information. The framework is agnostic to methods used for deriving predictions;
2. Flexibility in using predictions for self-adaptation. Our framework has several points of integration where predictions can be useful; and
3. Some rules-of-thumb for how to incorporate predictive information into a self-adaptive framework.

In the remainder of this chapter we describe our resource prediction framework and show how it achieves the improvements listed above. In Section 2, we describe the overall framework of our architecture-based self-adaptation approach and identify core challenges of incorporating prediction. We then introduce the anticipatory model for adaptation in Section 3. In Section 4 we present initial results of applying an anticipatory model to adaptation and describe future applications. In Section 5 we describe related work on architecture-based self-adaptation and prediction. In the final section, we conclude with a brief discussion of additional ways in which prediction could be used to improve architecture-based self-adaptation.

2 Framework for Architecture-Based Self-Adaptation

In this section we provide a high-level overview of our self adaptation framework, illustrate its use with an example, and discuss opportunities for enhancement via resource prediction. In particular, making use of prediction requires addressing a few challenges: What kinds of predictive information are useful? What can be predicted? How would it be used? This section addresses the first question of requirements for prediction. In the next two sections we address the questions of what and how.

To illustrate our approach, consider an example news service, Znn.com, inspired by real sites like cnn.com and RockyMountainNews.com, which serves multimedia news content to its customers. Architecturally, Znn.com is a web-based client-server system that conforms to an N-tier style. As illustrated in Fig. 1, Znn.com uses a load balancer (LB) to balance requests across a pool of replicated servers, the size of which is dynamically adjusted to balance server utilization against service response time. A set of client processes (represented by the C component) makes stateless content requests to the servers. Let us assume we can monitor the system for information such as server load and the bandwidth of server-client connections. Assume further that we can modify the system, for instance, to add more servers to the pool or to change the fidelity of the content. We want to add self-adaptation capabilities that will take advantage of the monitored system and adapt the system to fulfill Znn.com objectives.

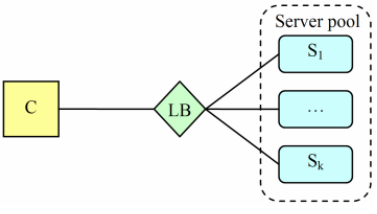


Fig. 1. Architecture model of Znn.com

The business objectives at Znn.com are to serve news content to its customers with reasonable response, while keeping the cost of the server pool within its operating budget. From time to time, due to highly popular events, Znn.com experiences spikes in news requests that it cannot serve adequately, even at maximum pool size. To prevent unacceptable latencies, Znn.com opts to serve minimalist textual content during such peak times in lieu of providing its customers zero service. Assume that two actions are possible to adapt the system: adjust the server pool size (enlist or remove) or switch content mode (multimedia or textual). While seemingly simple, an adaptation decision requires a tradeoff between the multiple objectives.

2.1 Overview of the Rainbow Framework

Our architecture-based self-adaptive approach is embodied in an engineering framework, called Rainbow, which provides mechanisms to monitor a target system and its executing environment, reflect observations in an architecture model, detect opportunities for improvements, select a course of action, and effect changes. By leveraging the notion of *architectural style* to exploit commonality between systems, the framework provides general and reusable infrastructures with well-defined customization points to cater to a wide range of systems. It also provides a useful set of abstractions to focus engineers on adaptation concerns, facilitating the systematic customization of Rainbow to particular systems. Details can be found in [3,4].

The Rainbow framework (Fig. 2) uses a component-and-connector architecture model of the target system to monitor and reason about appropriate strategies for adapting the system. Monitoring mechanisms—*probes* and *gauges*—observe the running *target system*. Observations are reported to update properties of the architecture model managed by the *Model Manager*. The *Architecture Evaluator* evaluates the model upon update to ensure that the system is operating within an acceptable range, as determined by architectural constraints. If the Evaluator determines that the system is not operating within the accepted range, it triggers the *Adaptation Manager* to initiate the adaptation process and choose an appropriate adaptation strategy. The *Strategy Executor* then executes the strategy on the running system via system-level *effectors*.

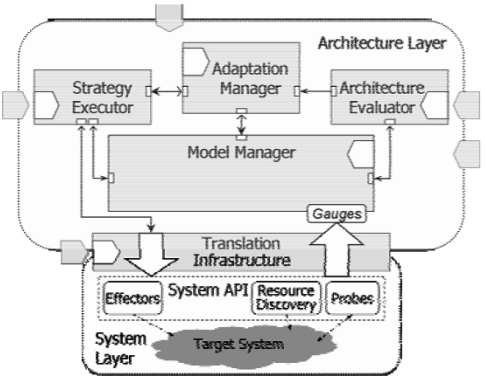


Fig. 2. The Rainbow Framework

To apply Rainbow to the Znn.com example, we use probes and gauges to monitor response time and server load, reflecting those as properties in the architecture model. The architecture evaluator triggers adaption when any client experiences request-response latencies above some threshold. The Adaptation manager determines whether to activate more servers or decrease content fidelity, as specified in a repair script. The strategy executor effects the change in Znn.com using provided hooks.

When the system comes under high load, Rainbow may opt to increase the server pool size until a cost-determined maximum is reached, at which point Rainbow would switch the servers to serve textual content. If the system load drops, Rainbow may switch the servers back to multimedia mode to make customers happy, in combination with reducing the pool size to reduce operating cost. In general, the adaptation decision is determined by both the business objectives and observations of system conditions, including average response time, server load, and available bandwidth.

2.2 Elements of Rainbow

The Rainbow framework uses models of the architecture and environment to make adaptation decisions. A component-and-connector (C&C) architecture model reflects abstract, runtime states of a target system, including what entities are present and how they communicate [5]. An environment model provides contextual information about the system, including its executing environment and the resources used. For example, when additional servers are needed, the environment model indicates what spare servers are available. When a better connection is required, the environment model contains information about the available bandwidth of other communication paths.

In order to get information out of the target system into an abstract model for management, and then to push changes back into the system, we need mechanisms that hook into the target system and understand what is represented in the model. Gauges

process system-specific information from Probes to populate architectural properties. Associated with architectural operators in the Rainbow Architecture Layer, effectors carry out change operations on the target system via mechanisms that range in complexity from a system-call, to a script, to an elaborate workflow.

The Architecture Evaluator evaluates model conformance against architectural constraints, which are specified using first-order predicate logic to identify problems in the system. When triggered by the Architecture Evaluator, the Adaptation Manager uses information about the state of the system, embodied in the architecture, the business quality-of-service concerns and utility functions to decide which remedial strategy to execute. A strategy is chosen from a set of specified strategies that have been engineered for the system and/or domain. A strategy specifies conditions and contexts in which it applies, and captures a pattern of adaptation steps.

Business quality-of-service concerns for the target system (e.g., system reliability, service availability, or performance) are represented as quality dimensions. A quality dimension provides a notion of utility, or happiness, for particular values of a quality attribute. Each adaptation action has a specified impact in cost or benefit on each dimension. By tallying the *cost-benefit attributes* over the actions in a strategy, an expected aggregate impact can be computed for each strategy. A strategy can then be scored using utility preferences specified for the quality dimensions. The Adaptation Manager then selects the highest-scoring strategy.

Utility preferences define the relative importance between the quality dimensions. Specifically, we use a von Neumann-Morgenstern utility function $u_d : X_d \rightarrow \Re$ that assigns a real number to each quality dimension d, normalized to the range [0,1]. Across multiple dimensions, we attribute a percentage weight to each dimension to account for its relative importance compared to other dimensions. These weights form the utility preferences. The overall utility is then given by the utility preference function, $U = \Sigma w_d u_d$. An example utility preference with three objectives, u_1, u_2, u_3, of decreasing importance might be quantified as [w_1:0.6, w_2:0.3, w_3:0.1].

The utility preference function gives us a way to compute the *instantaneous utility* of the target system given its current conditions, as well as the *accrued utility* of the target system over time. If we assume coverage of system conditions, accrued utility provides a measure of optimality of the target system, giving us a way to compare the relative optimality of a system under different combinations of conditions.

2.3 Opportunities for Improving Self-Adaptation

To date, Rainbow's adaptation has been *reactive* in nature. Reactive adaptation has the advantage of requiring only a small set of recent system conditions to choose an adaptation, allowing for timely decisions. However, reactive adaptation has a number of well-known disadvantages. First, following the decision to perform an adaptation, time is needed to carry out and propagate the necessary changes on the target system. At times, the conditions that trigger an adaptation may be more short-lived than the duration for propagating the adaptation changes, resulting in an unnecessary adaptations that incur potential resource costs and service disruption, which we term *penalty*.

Second, reactive adaptation lags behind current system conditions, and the degree of that lag depends on the sensitivity of the system sensors to present (versus historical) values of a system condition (e.g., CPU load, link bandwidth). If the system

condition undergoes a dramatic and rapid shift, it may take numerous adaptation cycles for sensors to "catch up," resulting in more than one incremental adaptation change where a single adaptation might have sufficed. Again, this is problematic since each adaptation potentially incurs some penalty.

If a similar shift in system conditions recurs "seasonally"—once every *period* of time, such as every day at 8 AM—then the same undesirable pattern of incremental adaptations would repeat every period. (One workaround is to learn the seasonal pattern from historical data and predicate adaptations on time; however, this is a form of prediction.) In fact, executing adaptation while the system is under duress usually will take more time and is more likely to fail because of lack of resources. Having such prediction will help ensure sufficient resources are available for the adaptation.

Finally, knowledge of future availability of some required resource might result in a different adaptation choice that moves the system into a higher level of overall utility. To illustrate using a simplified Znn.com example, assume three levels of utility—*happy*, *somewhat happy*, *unhappy* – and three levels of values corresponding to resource conditions: *low*, *medium*, *high*. Assume that both high response time and zero service (i.e., no content) makes the customer unhappy, while low-fidelity content makes the customer somewhat happy. Assume further that an adaptation cycle takes one unit of time to effect its changes. We will represent the conditions of the system at a particular time-point with a tuple: (utility, response time, server load, available bandwidth, content fidelity). Now imagine a scenario lasting 3 time units, where Rainbow reacts to the conditions at time unit *1* by lowering the content fidelity:

0. (*happy* utility, *low* response time, *low* load, *high* available bandwidth, *high* fidelity)
1. (*unhappy*, *high*, *high*, *low*, *high*)
2. (*somewhat happy*, *medium*, *medium*, *low*, *low*)
3. (*somewhat happy*, *medium*, *medium*, *high*, *low*)

However, with perfect hindsight, knowing that the available bandwidth would recover to *high* might have led Rainbow to adapt by enlisting more servers to lower the average server load and to keep the fidelity high, thus achieving better overall utility:

3. (***happy***, *medium*, *medium*, *high*, *high*)

This example demonstrates how a reactive strategy of adaptation that optimizes instantaneous utility may often be sub-optimal over a long period of time. This deficiency results from two properties of reactive adaptation: (1) information used for decision making does not extend into the future, and (2) the planning horizon of the strategy is short and does not consider the effect of current decisions on future utility.

By analyzing its reactive nature, we have thus identified four opportunities for improving the current self-adaptation capabilities:

1. Preventing unnecessary self-adaptation
2. Reducing disruption from incremental adaptations.
3. Enabling pre-adaptation to seasonal behavior.
4. Improving overall choice of adaptation.

These opportunities for improving self-adaptation highlight the need for predictive information, particularly predictions of resources the target system environment. In

the following section, we characterize a number of different kinds of prediction and types of information that are amenable to prediction.

3 Resource Prediction

In the previous section we identified four opportunities for using prediction to improve self-adaptation of systems. For the purpose of this chapter, *prediction* is an informed estimation of the future random values of a system or environment variable, e.g., the future available level of some resource required by the system. By leveraging predictive information, a self-adapting system is able to analyze adaptation alternatives slightly, or even significantly, ahead of real-time, make forward-looking decisions based on those predictions, and potentially improve the performance according to some objective metric. In this section, we describe the types of prediction that we use, discuss their applicability and limitations, and then describe a generic prediction framework that was developed for use in a ubiquitous computing context, but which can be co-opted for use within Rainbow.

Poladian defined and described an anticipatory model of self-adaptation in the context of a ubiquitous computing system that makes resource allocation decisions based on predictions of three inputs: (1) predictions of user's tasks, e.g., what type of applications the user needs and for how long, (2) predictions of resource demand by resource- and fidelity-aware applications, and (3) predictions of the available supply of resources such as network bandwidth and battery. He developed a calculus and framework that can synthesize different categories of prediction about a resource to produce a single combined predictive value. The types of predictive models that can be synthesized with this approach are: (1) *linear recent history*, which is a kind of predictor that uses recent history and a linear time-series model; we use auto-regressive moving average (ARMA) models for this kind of resource prediction, which is consistent with [7]. (2) *Relative move,* which models seasonal variations in resource availability (e.g., knowing that network usage will be high at the beginning of a work day). (3) *bounding*, which specifies the maximum and minimum values of a resource for a union of time intervals (for example, knowing that bandwidth cannot be above 10Mbps). In this chapter, we are concerned with how to integrate the prediction architecture with a self-adaptive system, rather than the particular models of prediction used. For details of the types of predictive models, and the calculus for combining them, we refer readers to [20,21].

Because predictions are rarely perfect, a model of prediction must be prepared to address *uncertainty*. Broadly, *uncertainty* describes both measurement and estimation error when making predictions. Consequently, we differentiate between two types of uncertainty. The first type of uncertainty arises when estimating future, random values of variables. One familiar example of such uncertainty is forecasting tomorrow's weather. Predicting (forecasting) tomorrow's temperature is generally imprecise, and a good prediction would provide an estimate for the uncertainty (error) in the forecast. Moreover, the error increases the further into the future one is predicting. Examples from computer systems include predicting the number of clients connected to the system or the available supply of network bandwidth in ten minutes. The second type of uncertainty arises when measuring the magnitude of past and present values of

variables. An example from the physical sciences is the measurement of voltage. Here the uncertainty (error) is the result of imprecision, rather than randomness, that can only be resolved by waiting until some future time. An example from computer systems includes measuring the current available bandwidth between two network nodes.

Prediction and uncertainty in the context of self-adaptive systems must be modeled and addressed together. Typically, making predictions requires a statistical model that estimates (calculates) future values of a variable based on available information to the system. The uncertainty in the prediction is a rigorous description of the predictive error based upon that statistical model. In other words, prediction and uncertainty are described by the predictive distribution of the variable being estimated, conditional on all available data, e.g., the past values of the variable as well as the past values of the prediction errors and any other information.

The types of prediction models and the way of combining them can be extended to a certain class of self-adaptive systems that (a) monitor and predict resource availability, and (b) make resource allocation decisions as part of self-adaptive behavior. Typically, such systems are concerned with measuring or estimating both the *demand* for computational resources by the system under consideration and the *supply* of resources available to that system. In practice, the demand and the supply might be dependent. Therefore, it is important to identify when those are interdependent and express the dependence. Essentially this means whether each critical resource in the environment of the self-adaptive system is shared among many systems or entirely dedicated to the system under consideration. If the resource is not under our control, then we can simply use the aggregate predictions of that resource where the future value of that resource is based on the historical values of that resource. However, if the resource is being managed wholly by the self-adaptive system, then the prediction is more complicated; we need to predict how each element under our control uses that resource. In either case, the predictive framework can be applied equally effectively.

The kinds of predictions that can be handled by the prediction framework are for resources that have historical data that can be analyzed statistically and that match our statistical model of the resource in question. For example, if the historical data fits a Poisson distribution then it is obviously not applicable for an ARMA predictor that assumes Gaussian distribution. So, predictions that assume uncertainty is normally distributed may fail to detect the arrival of a so-called "Slashdot effect," when a rapid increase of web clients are connected to the server due to a sudden surge in the popularity of the web server. This is especially the case if the historical data does not contain evidence of a Slashdot event.

Our approach to anticipatory adaptation is based on optimizing the match between system needs and the environment capabilities. In practice, finding such a match corresponds to maximizing system utility. Poladian's thesis defines an analytical model that formalizes the notion of utility for user's tasks and expresses automatic configuration as a mathematical problem of maximizing the expected utility of the user from the running state of the environment under the constraints of the computing environment.

The analytical model provides a carefully crafted structure for the problem, allowing efficient runtime configuration algorithms to search the problem space for good solutions. That structure is used to define a configuration strategy for prediction that takes as input (1) the amount of historical information about the resource being

predicted, (2) the temporal horizon of the decisions, and (3) treatment of uncertainty in the available information explicitly quantifying the uncertainty of future events and coping with uncertainty by planning for future changes.

Using this analytical model, Poladian designed and implemented a software infrastructure for automatic configuration with three important contributions: (1) a central component that makes near optimal configuration decisions, (2) a prediction framework that provides resource prediction on demand, and (3) a programming interface between the centralized decision maker and the prediction framework. The central decision-making component leverages the structure of the analytical models to implement efficient and near-optimal configuration algorithms. In particular, the framework consists of the following four types of components, the architecture of which is defined in Fig. 3. For each resource, there will be one instantiation of this framework.

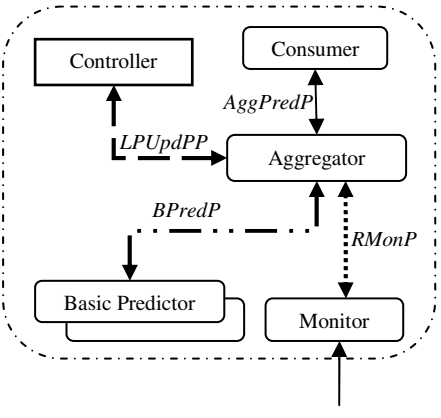


Fig. 3. The resource-prediction framework

Aggregator: the centerpiece of the prediction framework, is responsible for combining information from all available Basic Predictors and calculating aggregate predictions. The Aggregator maintains an up-to-date list of currently available Basic Predictors. It aggregates the information from the predictors and produces a time series of predictions with increasing uncertainty further into the future.

Controller: allows setting the model parameters of the linear recent history predictor in the Aggregator. The model parameters are expected to be relatively stable over time, changing only infrequently. There is one Controller resource instance.

Basic Predictors: these components implement a wrapper around either known pattern or bounding predictors. Multiple Basic Predictors can be used. Upon startup, a Basic Predictor registers with the Aggregator. As new sources of predictions become available, additional Basic Predictors can be added to the framework,

Monitor: probes the environment for actual resource availability and provides periodic monitoring reports to the Aggregator. These monitored values correspond to the the historical values used by the predictors. A Monitor provides a uniform interface to the aggregator, encapsulating platform, network, and resource-specific details,

Consumer: the recipient and beneficiary of aggregate predictions. A Consumer is implemented by the coordinating entity of an adaptive resource management system. The prediction framework allows multiple concurrent Consumers to co-exist, each with its own aggregate prediction session. The Consumer specifies prediction parameters to the Aggregator including the sampling window to make prediction observations and how far into the future to predict.

In summary, the resource prediction framework quantifies the future level of resource availability by combining predictive information from multiple sources. More details can be found in [21].

4 Incorporating Resource Predictions in Rainbow

Poladian's work on resource prediction is both practical in terms of algorithm speed, and useful in terms of manageable parameter space. In this section, we show how resource predictions can be incorporated into the Rainbow self-adaptation framework. Rainbow must satisfy the following requirements of Poladian's framework:

1. *Utility:* to evaluate the quality of the various possible adaptations on the system. Rainbow has a notion of utility as a central concept for strategy selection;
2. *Penalty*: to quantify costs of performing adaptations. If there is no penalty associated with adapting the system, this would obviate the need for using prediction – we will do a much better job with a reactive approach. In Rainbow, the penalties reflect the impact of temporary disruptions to system utility and the also time it takes to propagate changes throughout the system; and
3. *Historical information:* to facilitate prediction, past observed values need to be fed to the prediction framework.

4.1 Integration Points to Make Predictive Information Available

In Rainbow resource predictions can provide additional leverage in evaluating and choosing between alternate strategies of adaptation. For example, by knowing the probability that the available level of a critical resource, such as bandwidth, will be below a certain threshold 5 minutes from now, Rainbow can choose a strategy that quiesces lower priority client sessions so that the remaining client requests will continue to be satisfied within a tolerable latency. If, on the other hand, the probability is high that the bandwidth will be restored to levels that will naturally bring the system back within its desired state, Rainbow can choose to reduce the fidelity of some or all of the client sessions. Rainbow can even choose to do nothing.

To leverage resource predictions, it is important to consider how predictive information adds to the existing information flow of adaptation decisions in the Rainbow framework. The following points in Rainbow are potential sites for integrating resource predictions:

- Monitoring: predictor gauges
- Detection: prediction of architectural properties
- Strategy: conditions based on predicted value and actions with time cost
- Effector: addition or removal of prediction data streams

Monitoring: To make adaptation decisions, Rainbow reads gauge output to determine target-system conditions. We can integrate resource predictions in Rainbow by encapsulating, as gauges, instances of the entire prediction runtime from Poladian's system. It provides output to the gauge bus consistent with the gauge infrastructure API. Rainbow uses the standard gauge control interface to configure parameters of the

prediction runtime. The gauge performs the role of the consumer, providing parameters for the prediction, then processing the time series returned from the aggregator to produce a single predicted value for one future time, as requested by Rainbow.

Because uncertainty is inherent in resource prediction, we must incorporate the probability of error in a predicted measurement, as supplied by predictors. We can choose to ignore predicted measurements and fallback to current measurements when the confidence level is below some threshold. We can also incorporate confidence level directly in utility computation to give lower consideration to strategies that use low-confidence predictive information.

Detection: Rainbow uses architectural constraints to identify opportunities for adaptation. Conditions based on predicted resource states, such as the anticipated load in the next 500 milliseconds, may indicate opportunities for adaptation. Thus, architectural constraints should support predicates over predicted values of architectural properties, perhaps in the form of a supplied architectural function, such as predictedProperty(p : Property, dur : int) : float (similarly for functions providing basic statistical operations, e.g., max/min/average). The predictedProperty() function returns the value of the architectural property identified by p, at a time point dur milliseconds from now. Recall that gauges are associated with specific architectural properties to update their values. So the function can compute predicted values by querying the predictor gauge mapped to the requested property.

Strategy: At adaptation time, Rainbow uses current system conditions (reflected in the model) to score and select strategies based on their expected utility. A strategy has two important ingredients: system *conditions* and adaptation *actions*. System conditions are used to (a) determine the applicability of strategies during strategy selection and (b) decide the next adaptation step during strategy execution. Adaptation actions change the target system to move the system toward a better state. New capabilities are required in the mechanisms for strategy selection, applicability condition, and actions to incorporate resource predictions.

An example strategy to reduce system response time is shown in Fig. 4, specified in Rainbow's adaptation language. The function defined on line 1, cPredViolation(), uses the architectural function predictedProperty() to compute client experienced response time at some future time, specified by dur. (cViolation defines the same predicate without using a predicted value.) Line 4 shows the use of this predicted value to determine the applicability of this strategy, in this case, when the client experienced response time is above threshold now and in the future. Lines 5-9 specify what the strategy

```
01  define boolean cPredViolation (dur : int)=
      exists c : T.ClientT in M.components |
      Model.predictedProperty(c.experRespTime,
      dur) > M.MAX_RESPTIME;
02  ...
03  strategy VariedReduceResponseTime
04   [ cViolation && cPredViolation(self.dur) ] {
05    t0: (cViolation) -> enlistServers(1)
                                  @[1000 /*ms*/] {
06      t1: (!cViolation) -> done;
07      t2: (cViolation) -> lowerFidelity(2, 100)
                                  @[3000 /*ms*/] {
08        t2a: (!cViolation) -> done;
09        t2b: (default) -> TNULL;  // give up
10  } } }
```

Fig. 4. Sample snippet of an adaptation strategy

does. In this case, it first enlists a server. Failing that, it then lowers the fidelity. And if that doesn't work, it gives up.

Timing plays a crucial role in prediction. We add the ability to calculate forward-looking expected utilities based on future system conditions. We augment Rainbow with the notion of *future variable value* so that a strategy can specify dependency on the future value of a condition. An adaptation action takes some time to execute, and estimating this duration is needed to determine how far into the future to predict. So, using settling time information specified in strategies (see @[ms] in Fig. 4, lines 5 and 7), Rainbow estimates the amount of *time* that a strategy would take to execute successfully. It then measures actual execution times to improve estimation.

For prediction to improve the performance of Rainbow, recall that there needs to be some cost, or *penalty,* to doing a particular adaptation. To capture this, we model penalty as a separate utility dimension, called disruption, that can be applied in utility-based strategy selection like other dimensions, such as average response time (see Table 1). There are two parts to disruption: one is how jarring it is to the user, and the other is how long the user is disrupted. We collect information about the disruption level in the same way as other dimensions, specified as part of the strategy specification. The second one we track automatically by measuring how long it takes to execute an adaptation step.

Effector: Finally, changes to the target system, particularly changes that add or remove resource components, will likely have significant effects on resource predictions. Therefore, we rely on system-level effectors to be augmented so that, when adding or removing system elements with associated resources, the effectors also take care of the addition or removal of the corresponding prediction data streams. Additionally, because prediction usually requires a series of input before the first output of predictive data, gauges may have to be coordinated with the addition of prediction data streams to produce useful output immediately.

4.2 Illustration of Rainbow with Resource Predictions

To illustrate resource predictions in Rainbow, let us revisit the Znn.com example to examine in more detail the four scenarios of prediction introduced in Section 2.3. Recall that in the Znn.com example, the customers care about quick response time and high content fidelity for their news requests. While aware of customer preferences on content fidelity, Znn.com as the provider is constrained by infrastructure provisioning costs. We also consider service disruption as a penalty of performing an adaptation: avoiding penalties is important to improving overall system utility, which is a major benefit to having predictive information.

Accordingly, we define four quality dimensions and determine the corresponding measurable properties in the target system. We capture each dimension as a discrete set of values (for example, we use an ordinal scale of 1 to 5 to express the degree of disruption). We then elicit from the service providers the utility values and preferences for these dimensions, summarized in Table 1.

Table 1. Znn.com quality dimensions and utility preferences

Label	Description	Architectural Property	Utility Function	Weight
uR	Avg Response Time	ClientT.experRespTime	((low,1), (med,0.5), (high,0))	25%
uF	Avg Content Fidelity	ServerT.fidelity	((textual,0), (multi-media,1))	10%
uC	Avg Budget	ServerT.cost	((within,1), (over,0))	15%
uD	Disruption	ServerT. droppedReqs	((1,0.8), (2,0.6), (3,0.4), (4,0.2), (5,0))	50%

A rule specifies the acceptable bound of request-response latencies experienced by a client: exceeding the threshold indicates a problem. A set of operators correspond to available effectors in Znn.com to enlist or remove servers, or to change content fidelity. We define a number of adaptation strategies for Znn.com and specify cost-benefit attribute vectors, not shown here, that specify the impact of each strategy to the four quality dimensions. For example, strategy VariedReduceResponseTime is expected to lower response time and fidelity level, not affect cost, and incur some disruption.

We now consider how prediction could improve Rainbow's choices of adaptation for the four opportunities outlined in Sec. 2.3. For evaluation, we set up Znn.com in a simulation environment that allows us to experiment with prediction-enabling design points in Rainbow's Architecture Layer (cf. Fig. 2). The states of Znn.com are simulated using an M/M/k queuing model. The simulation environment acts as gauges that update corresponding Znn.com architectural properties in Rainbow. This setup enables prediction of future states to an arbitrary precision.

Scenario 1: Avoiding Unnecessary Adaptation

In the first scenario, if a client experiences an above-threshold request-response time for only 500 ms, but the chosen adaptation requires at least one second to complete, this adaptation is unnecessary. Avoiding adaptation requires knowing the predicted request-response time (using the architectural function predictedProperty()) and the estimated execution time of an adaptation strategy, which Rainbow collects.

To evaluate how well prediction improves overall system utility in this scenario, we designed two Znn.com configurations, one in which the bandwidth drops briefly, and another in which incoming requests (load) spike briefly. The data is summarized in Table 2. In both cases, Rainbow with prediction successfully avoided making unnecessary adaptations, improving the normalized accrued utility over no prediction by 2.5% in the transient bandwidth-drop case, and 15.7% in the transient peak-load case. The much greater improvement in the second case can be attributed to the high level of disruption incurred by the strategy that is unnecessarily invoked without future

Table 2. Summary of data from 3 experiments (each averaged over 30 trials)

Scenario Configuration	Normalized Accrued Utility (AU)		ΔAU	Improved
	No Prediction	With Prediction		
1: transient bandwidth-drop	0.889	0.911	0.022	2.5%
1: transient peak-load	0.731	0.846	0.115	15.7%
2: ramp-up to peak load	0.734	0.770	0.036	4.9%

knowledge This outcome underscores the role of *penalty* in determining whether prediction is useful. We discuss some choices of prediction usage in Section 4.3.

Scenario 2: Reducing Incremental Disruptions
In the second scenario, Znn.com experiences a dramatic increase in client requests, ramped up over seconds to minutes. In reaction, Rainbow provisions by invoking a strategy that adds one server. However, by the time the server is added, the request load has surpassed the capacity of the added server, so Rainbow adds another server in response. This gradual adaptation is undesirable because it disrupts the system multiple times. Eliminating this ramp-up requires knowing the peak of the ramp-up and computing cost-benefit attributes based on input arguments to an adaptation step (e.g., *k* in enlistServers(*k*)).

To evaluate this scenario, we designed a Znn.com configuration that ramps up requests over four seconds. We added a *leap* strategy similar to VariedReduceResponseTime but enlists 3 servers in one step. We then configured Rainbow to compute utilities that look five seconds ahead, and compute the load at its peak. Rainbow successfully selected the leap strategy and showed a 4.9% improvement in AU.

The results of these experiments show that there is improvement when using predictive information. Perhaps not surprisingly, the most improvement is achieved when the potential disruption to the user is high. For the other cases, the room for improvement is not as great, but our numbers are significant when measured against the available margin for attaining perfect utility.

Additional Scenarios: Seasonal Pre-adaptation and Choosing Better Adaptations
We have shown two scenarios that exercised the new capabilities added to Rainbow to incorporate predictive information, with supporting data from experiments. We now consider two other scenarios that use the same set of capabilities; for these we have not performed additional experiments.

In a third scenario, Znn.com periodically experiences a significant increase in client requests at 9 AM every Monday through Friday. Reacting to the increase each time it occurs is undesirable because the adaptation potentially disrupts the system and adds stress to a system already under load. In contrast, pre-adapting has the benefit of reducing disruption while introducing system *slack* to prepare for the upcoming load. Pre-adapting for seasonal behavior requires detecting seasonal patterns, which can be provided by predictors in Vahe's framework. Then, by adding an architectural constraint that checks for predicted load at fixed future time points, configuring utility computation to look ahead to the same time, and specifying a strategy that is applicable for violation at that future time point, Rainbow can seasonally pre-adapt.

A fourth scenario is already described in Section 2.3, where a client experiences an above-threshold request-response time due to increased visitor traffic, coupled with a transient drop in available bandwidth. Given the low bandwidth and a choice between the a strategy to lower fideltiy and another to enlist more servers, Rainbow chooses the former to use less bandwidth while fulfilling the increased request load. However, when the available bandwidth recovers shortly afterward, Rainbow would then adapt again to restore the content fidelity and perhaps also enlarge the server pool if traffic remains high. Thus, Rainbow's reaction results in at least one additional disruption and an overall lower system utility. With advanced knowledge that the bandwidth drop is transient (as in scenario 1), Rainbow would have chosen to enlist servers.

4.3 Deciding When to Use Predictive Information

Once predictive information is available for use in self-adaptation, the questions still remain of when and how to use the information in the decision process. In our application of predictive information, we encountered the following design choices, which we have addressed in a variety of ways.

How far into the future do we look ahead? The predictive framework requires parameterization for how far ahead to predict a resource property. The predictive framework actually returns a time series of values, but to make use of this information in Rainbow, we must pick one particular value. The choice of this depends on the context. For example, in Scenario 1 where we are trying to decide whether to avoid an adaption, a reasonable choice is to use a duration equivalent to the estimated time of completing the adaptation. For Scenario 2 on the other hand, the look ahead could be far longer than the duration of adaptation. Note that by using estimated completion time to choose how far into the future to look, we are comparing different prediction ranges for different strategies in a single adaptation cycle. An alternative is to look ahead to the same time in the future, perhaps by using the maximum completion time of all strategies under consideration.

Should predictive information be used at strategy selection time, utility evaluation time, strategy execution time, or a combination of these? There are several steps in Rainbow's process of selecting a repair strategy: 1) decide the set of strategies that may fix a problem; 2) determine which strategy is the best to use; and 3) execute the chosen strategy. Predicted information can be used in Rainbow at any of these times. For example, the strategy in Fig. 4 uses predicted information in step 1. In line 4, we are checking if response time is high now and in the future. If the condition is transient the strategy will not be chosen, and so there is no need to use prediction in lines 5-9 (strategy execution time). Alternatively, to anticipate seasonal changes, the strategy writer would write the strategy to consider only predictions in line 4, and also to use prediction in lines 5-9. Rainbow gives the strategy writer the power to decide how and when to use the prediction. Currently, in Rainbow, the second step (using prediction in the utility calculation) is provided as a parameter to the framework, because there is no way in the strategy language to refer to this. We are investigating a more agile way to specify the use of prediction in this case.

How much weight should be given to the penalty dimension? When we experimented with a 10% weight for the penalty dimension, the first configuration yielded a utility improvement of 0.4%, whereas a 50% weight yielded 2.4% improvement. This data reinforced Poladian's results that anticipatory adaptation yields increasing gains at penalty levels above 7%. On the flip side, a penalty weight above 50% makes it difficult to distinguish the relative importance of the other utility dimensions. A sweet spot should be found between 10 and 50%.

5 Related Work

To date, several dynamic software architecture-based adaptation approaches and frameworks have been proposed and developed [12, 19]. Related approaches focus on formalism and modeling, mechanisms for adaptation, or distribution and

decentralization of control. These include Darwin with π-calculus semantics to specify distributed systems [16], ArchWare with architectural reflection and dynamic coevolution [17], Weaves for construction and analysis of data-flow systems [13], ArchStudio for self-adaptation of C2 hierarchical publish-subscribe systems [6], Plastik targeting performance properties [1], and CASA for resource availability concerns in mobile network environments [18]. These approaches share a few common characteristics: They generally apply a closed-loop control, use an architecture model for reasoning about the target system, assume certain structures in the target system, and adapt for a fixed set of quality attributes.

Notable in industry, IBM's Autonomic Computing tackles the challenges of emergent autonomic behavior with the MAPE control loop—to monitor, analyze, plan, and execute changes for self-management. The AC toolkit provides consoles and tools to diagnose problems and engineer autonomic systems. We apply a similar approach.

One of the differentiators of this work from prior self-adaptive systems is the use of resource prediction. The anticipatory strategy uses predictions of the future values of input variables to make forward-looking decisions about adaptation selection. Forward-looking approaches have been proposed and used in other domains. For example, the online stochastic combinatorial optimization approach is similar to our anticipatory strategy [2,14]. Various combinatorial optimization problems such as optimal vehicle dispatch and network packet routing are solved by leveraging probabilistic priors of the future values of problem inputs. There is equivalence between the algorithms for automatic configuration in this chapter and the algorithms described in [14]. The Active Virtual Network Management Prediction System uses simulation models running ahead of real time to predict resource demand among network nodes. Such predictions can be used to allocate network capacity in anticipation of demand increase, and to ensure adequate quality of service to different network flows [9]. Our work shares theoretical foundations with these, but the problem domains are different.

There is a body of work that uses various kinds of prediction to improve self-adaptation. For example, Clockwork [22] introduces the concept of *predictive autonomicity* that uses statistical modeling to forecast cyclic variations in system load and uses these predictions to reconfigure systems in anticipation of need. They prescribe a method for implementing a predictive autonomic system. In spirit, we share the same steps for incorporating predictive information. However, our notion of controllable parameters are enriched with strategies and utility preferences, and we use predicted information in strategy selection. Solomon [23] uses predictions about workload to adapt the control component of an autonomic system to be more suited to that workload. For example, if the workload is linear, then simple thresholding can be used in the controller, but if the workload on the system changes to be Gaussian, then a more sophisticated statistical controller based on Kalman filters is swapped in to manage the system. Their adaptation layer shares the same principle components as Rainbow, with the selection of controllers analogous to selection of strategies. They show encouraging results in using predicted information for Gaussian workloads to provision servers. This is one of many types of prediction sources that could be incorporated into our prediction framework.

Rather than using auto-regressive techniques to predict resource availability, Lu [15] uses knowledge about the domain being controlled to predict behavior. They use queuing-theoretic models in the domain of web servers to infer expected delays

directly from input load. Again, this is another form of predictive model that could theoretically be incorporated as a Basic Predictor in our framework, although it is of a type that we have not fully considered.

6 Conclusion and Future Work

In this chapter, we presented an approach to enhance architecture-based self-adaptation through anticipatory prediction of future resource availability. The approach uses a framework that combines various forms of prediction (statistical, bounded, and seasonal) in a practical manner that can be applied to a variety of circumstances. We have argued that self-adaptive systems can take advantage of prediction to improve the choice of adaptations and to reduce disruption to the system. We gave specific consideration to the changes needed to incorporate predictions into one reactive architecture-based self-adaption system, Rainbow. We conducted several experiments that show improvement in the adaptation when predicition is used, and discussed how we addressed some issues that we encountered doing the integration.

In future work, we would like to better quantify the types of resources that can be predicted and would be useful in realistic circumstances. For example, other types of resources to be considered beyond bandwidth are power consumption, memory usage and CPU load. We would like to give more guidance to adaptation writers about when and how to use prediction. We also would like to verify the results discussed in this chapter through additional experimentation and application to real systems.

Acknowledgments. This research was funded in part by the National Science Foundation Grants ITR-0086003, CCR-0205266, CCF-0438929, CNS-0613823, by the Sloan Software Industry Center at Carnegie Mellon, by the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298 and by DARPA grant N66001-99-2-8918. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the US government or any other entity.

References

1. Batista, T.V., Joolia, A., Coulson, G.: Managing dynamic reconfiguration in component-based systems. In: Morrison, R., Oquendo, F. (eds.) EWSA 2005. LNCS, vol. 3527, pp. 1–17. Springer, Heidelberg (2005)
2. Bent, R., van Hentenryck, P.: Regrets only! Online stochastic optimization under time constraints. In: Proc. 19th AAAI (2004)
3. Cheng, S.-W.: Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation, Ph.D. Thesis, TR CMU-ISR-08-113, Carnegie Mellon University School of Computer Science (May 2008)
4. Cheng, S.-W., Garlan, D., Schmerl, B.: Making Self-Adaptation and Engineering Reality. In: Babaoğlu, Ö., Jelasity, M., Montresor, A., Fetzer, C., Leonardi, S., van Moorsel, A., van Steen, M. (eds.) SELF-STAR 2004. LNCS, vol. 3460, pp. 158–173. Springer, Heidelberg (2005)
5. Clements, P., et al.: Documenting Software Architecture: Views and Beyond. Pearson Education, London (2003)

6. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: Towards architecture-based self-healing systems. In: Garlan, et al. [10], pp. 21–26 (2002)
7. Dinda, P., O'Halloran, D.: Host Load Prediction Using Linear Models. Cluster Computing 3, 4 (2000)
8. Frye, C.: Self-healing systems. Appl. Dev. Trends, 29–34 (September 2003)
9. Galtier, V., et al.: Predicting resource demand in heterogeneous active networks. In: Proc. MILCOM (2001)
10. Garlan, D., Kramer, J., Wolf, A. (eds.): Proc. 1st ACM SIGSOFT Workshop on Self-Healing Systems (WOSS 2002), November 18–19. ACM Press, New York (2002)
11. Georgiadis, I., Magee, J., Kramer, J.: Self-organizing software architectures for distributed systems. In: Garlan, et al. [10], pp. 33–38 (2002)
12. Ghosh, D., Sharman, R., Rao, H.R., Upadhyaya, S.: Self-healing systems - survey and synthesis. Decision Support System 42(4), 2164–2185 (2007)
13. Gorlick, M.M., Razouk, R.R.: Using Weaves for software construction and analysis. In: Proc. 13th International Conf. of Software Engineering, pp. 23–34. IEEE Computer Society Press, Los Alamitos (1991)
14. Hentenryck, P., et al.: Online stochastic optimization under time constraints (2008), `http://www.cs.brown.edu/people/pvh/aor5.pdf` (last accessed April 2008)
15. Lu, Y., Abdelzaher, T., Lu, C., Sha, L., Liu, X.: Feedback Control with Queuing-Theoretic Prediction for Relative Delay Guarantees in Web Servers. In: Proc. IEEE Real-Time and Embedded Technology and Applications Symposium (2003)
16. Magee, J., Kramer, J.: Dynamic structure in software architectures. In: SIGSOFT 1996: Proc. of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering, pp. 3–14. ACM, New York (1996)
17. Morrison, R., Balasubramaniam, D., Oquendo, F., Warboys, B., Greenwood, R.M.: An active architecture approach to dynamic systems co-evolution. In: Oquendo, F. (ed.) ECSA 2007. LNCS, vol. 4758, pp. 2–10. Springer, Heidelberg (2007)
18. Mukhija, A., Glinz, M.: A framework for dynamically adaptive applications in a self-organized mobile network environment. In: ICDCSW 2004: Proceedings of the 24th International Conference on Distributed Computing Systems Workshops—W7: EC (ICDCSW 2004), pp. 368–374. IEEE Computer Society, Washington (2004)
19. Oreizy, P., et al.: An architecture-based approach to self-adaptive software. IEEE Intelligent Systems 14(3), 54–62 (1999)
20. Poladian, V., Garlan, D., Shaw, M., Schmerl, B., Sousa, J.P., Satyanarayanan, M.: Leveraging Resource Prediction for Anticipatory Dynamic Configuration. In: Proc. 1st IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2007), July 2007, pp. 214–223 (2007)
21. Poladian, V.: Tailoring Configuration to User's Tasks under Uncertainty, Ph.D. Thesis, TR CMU-CS-08-121, Carnegie Mellon University School of Computer Science (May 2008)
22. Russel, L., Morgan, S., Chron, E.: Clockwork: A new movement in autonomic systems. IBM Systems Journal 42, 1 (2003)
23. Solomon, B., Ionescu, D., Litoiu, M., Mihaescu, M.: A Real-Time Adaptive Control of Autonomic Computing Environments. In: Proc. 4th International Information and Telecommunication Technologies Symposium (U2TS 2006), December 2006, pp. 94–103 (2006)
24. Sousa, J.P.: Scaling Task Management in Space and Time: Reducing User Overhead in Ubiquitous-Computing Environments, Ph.D. Thesis, TR CMU-CS-05-123, Carnegie Mellon University School of Computer Science (2005)
25. Sztajnberg, A., Loques, O.: Describing and deploying self-adaptive applications. In: Proc. 1st Latin American Autonomic Computing Symposium, July 14–20 (2006)

Policy-Based Architectural Adaptation Management: Robotics Domain Case Studies

John C. Georgas[1] and Richard N. Taylor[2]

[1] Department of Computer Science, Northern Arizona University
Flagstaff, AZ 86011 USA
John.Georgas@nau.edu
[2] Institute for Software Research, University of California, Irvine
Irvine, CA 92697 USA
taylor@ics.uci.edu

Abstract. Robotics is a challenging domain that exhibits a clear need for self-adaptive capabilities, as self-adaptation offers the potential for robots to account for their unstable and unpredictable deployment environments. This paper focuses on two case studies in applying a policy- and architecture-based approach to the development of self-adaptive robotic systems. We first describe our domain-independent approach for building self-adaptive systems, discuss two case studies in which we construct self-adaptive ROBOCODE and MINDSTORMS robots, report on our development experiences, and discuss the challenges we encountered. The paper establishes that it is feasible to apply our approach to the robotics domain, contributes specific examples of supporting novel self-adaptive behavior, offers a discussion of the architectural issues we encountered, and further evaluates our general approach.

1 Introduction

One of the major current challenges in software engineering is the development of self-adaptive systems, which are systems that are able to change their behavior in response to changes in their operation or their environment for a variety of goals. In contrast to more specialized terms such as self-healing or self-optimizing, we use the term self-adaptive to inclusively refer to this class of systems, without consideration for their specific adaptive goals.

In addition to self-adaptive software, we are also keenly interested in robotic systems. These systems are amalgams of software and hardware that are highly resource constrained, commonly deployed in environments out of reach of human operators, exhibit a high degree of dependence on events in their environment, and commonly required to perform functions to which there can be little interruption. The domain characteristics naturally motivate the inclusion of self-adaptive capabilities that are currently lacking. The focus of this paper is the intersection of the robotics domain with self-adaptive software, as we see both a driving need for such capabilities in robotic systems as well as a fruitful application domain for self-adaptive technologies.

B.H.C. Cheng et al. (Eds.): Self-Adaptive Systems, LNCS 5525, pp. 89–108, 2009.

The challenge of integrating self-adaptive capabilities into robotic systems is a two-front battle: First, the system itself must be built in such a manner as to be conducive to adaptation by virtue of its very design and construction. Only then can adaptive behavior be integrated into the system. In our research efforts, the fact that the construction of the system must support adaptation implies that modularity is one of the fundamental qualities. This quality is the key enabler that allows adaptation to take place in a fine-grained manner rather than adaptation through wholesale replacement. In addition, due to the fact that adaptive needs are virtually impossible to fully and correctly predict during design, we also posit that adaptive behavior must be built in a way that is flexible and modifiable at runtime.

Much of our previous work has been dedicated to the development of architecture-based self-adaptive systems, and we identify a clear parallel in the capabilities this work provides and the needs of self-adaptive robotic systems. More specifically, we have developed notations and tools that support the design and development of policy- and architecture-based self-adaptive systems that are modular and have the ability to change adaptation policy specifications during system runtime [1]. Our goals with the specific work described in this paper are to:

- establish the feasibility of integrating our policy- and architecture-based self-adaptive system research with robotics domain;
- develop novel self-adaptive capabilities in robotic systems that did not previously exhibit them; and,
- probe into the difficulties and pitfalls of such an integration effort.

This paper is a report of our work to date toward these goals and our experiences in striving to meet them. We describe two case studies we performed in developing self-adaptive robotic systems, beginning with our work in the Robocode system – a robotic combat simulator and development framework – and continuing by discussing the construction of an autonomous Mindstorms NXT robot. For each of these systems, we demonstrate practical self-adaptive solutions to practical domain challenges. Neither of these domains previously considered – much less provided support for – self-adaptive capabilities.

The key contributions of our work in this intersection between self-adaptive architectures and robotic systems are:

- verifying the feasibility of integrating robotics and architecture-based self-adaptive techniques;
- providing examples of novel self-adaptive capabilities in our case-study domains;
- uncovering an important architectural mismatch between architecture-based adaptation and current robotic domain practice; and
- demonstrating that our policy language is adequate for expressing robotic adaptations.

The remainder of the paper offers background information, presents our overall approach to self-adaptive systems, discusses our case studies, and concludes with future work and final remarks.

2 Background and Related Work

This section begins with a discussion of representative robotic architectures, paying particular attention on their support for runtime change. We also discuss related approaches to architecture-based self-adaptive systems.

2.1 Robotic Architectures

One of the first robotic control system architectures to gain wide acceptance was the *sense-plan-act* architecture (SPA) [2]. In SPA, control is accomplished through the *sense* component that gathers information from sensors, the *plan* component that maintains an internal world model used to decide on the robot's actions, and the *act* component that is responsible for executing actions.

SPA architectures, however, scale poorly as robotic systems grow in complexity and scope, and SUBSUMPTION [3] was developed to address these scalability issues. This architecture abandons world models and adopts layered compositions of reactive components. Communication between these components takes place through the *inhibition* and *suppression* of inputs and outputs of lower level components by higher level ones. While the component-based approach of this architecture allows for improved scalability and modularity, the supported modes of communication prove very limiting.

Most current robotic systems are heavily influenced by three-layer (3L) architectures, first described in [4]. These hybrid architectures separate robotic systems into three explicit layers and mix reactive and planning modes of operation: The *reactive* layer captures behaviors that quickly react to sensor information, the *sequencing* layer chains reactive behaviors together and translates high-level directives from the *planning* layer into lower-level actions; the *planning* layer is responsible for deciding on long-term goals.

Despite their differences, these robotic architectures share a commonality in their lack of support for runtime adaptation, discussed in more detail in our paper to a workshop attached to the International Conference on Robotics and Automation (ICRA) [5]. These architectures simply do not consider this concern in their design and therefore do not exhibit the necessary qualities to be amenable to the direct application of self-adaptive techniques – the minimally amenable, perhaps, is the SUBSUMPTION architecture, due to its focus on independent components.

This lack of support for the architectural qualities that promote ease of runtime change is the motivation for the development of the RAS architectural style, also described in [5]. The style combines insights from event-based architectural styles such as C2 [6] and the SUBSUMPTION and 3L robotic architectures, and is aimed at supporting the development of robotic architectures that are modular

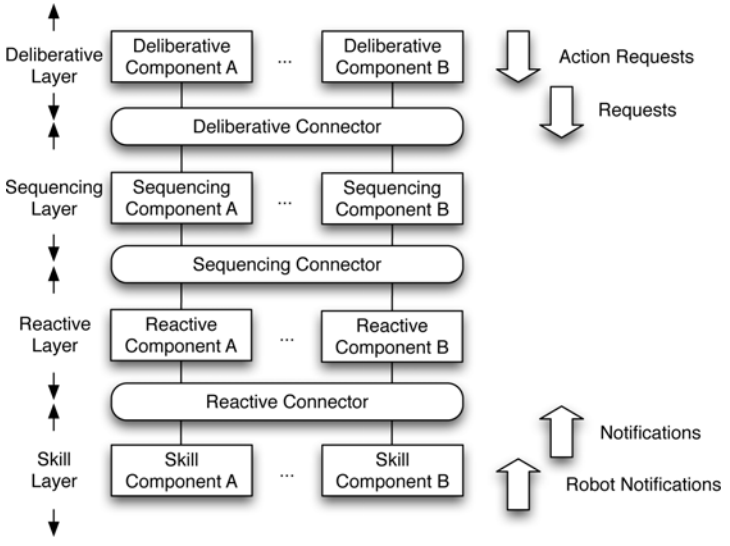


Fig. 1. An illustration of the RAS architectural style, showing the style's layers and event types

and incrementally evolvable while fostering component reuse. Figure 1 outlines the style, which is:

- component-based, with no shared memory between components;
- explicitly-layered into *skill*, *reactive*, *sequencing*, and *deliberative* layers[1] with components belonging to layers based on their complexity and maintenance of state information;
- event-based with communication taking place between components of the architecture through *requests* and *notifications*, sensor information being transmitted by *robot notifications*, and actions being enacted through *action requests*, and;
- connector-based, with independent connectors separating layers and facilitating communication and distribution.

The robotic systems we build in our case studies are built according to the principles of this style, which provides the basis for the construction of robotic systems that foster modularity and, therefore, are more easily modifiable using architecture-based means.

2.2 Self-Adaptive Architectures

Some related work takes a formal approach to the specification of architectures and the artifacts governing adaptation. The work based on COMMUNITY [7], for example, models architectures as abstract graphs while the approach based

[1] While the RAS style uses similar layer names as 3L architectures, there are differences between the two; the reader is referred to [5] for further details.

on the DARWIN [8] architecture description language (ADL) focuses on the self-assembly of systems according to a formally specified set of constraints representing invariant architectural properties. Other approaches are more focused on providing practical tool support for developing self-adaptive architectures. The RAINBOW system [9] adopts a style-based approach and focuses on the specification of styles for specific domains along with style-specific adaptations and constraints tailored for the domain's needs. Our own work is a descendant of such a tool-based approach [10], which conceptualized architecture-based adaptation but left many questions about how to implement adaptive behavior unanswered.

There is also work in the intersection of robotic systems and architecture-based approaches: Applied to sophisticated hardware platforms, the SHAGE framework [11] supports the definition of adaptive strategies managed by a controlling infrastructure, but focuses only adaptations that replace components with alternatives providing similar services. Kramer and Magee have also discussed self-adaptive robotic architectures through the application of a conceptual framework strongly influenced by 3L architectures and focused on self-assembling components using a formal statement of high-level system goals [12]. In contrast, the approach described here embodies a fundamental trade-off away from formal specifications of system behaviors in order to achieve a higher degree of flexibility and support for the runtime change of adaptation policies without necessitating the re-generation of adaptation plans.

3 Approach

Before discussing the case-studies, we present here the high-level approach we use for developing self-adaptive robotic systems. As this paper is focused on the application of our approach to robotic architectures, we keep the discussion minimal; a more extensive discussion of the overall approach appears in [13].

The core of our policy-based approach to architectural adaptation management (PBAAM) appears in Fig. 2. Self-adaptive systems in this approach consist of three fundamental parts: an architectural model specifying the system's structure, a set of adaptation policies capturing how the structure changes, and executable units of code corresponding to each architectural element. These three artifacts are managed at runtime by elements of the PBAAM infrastructure: the *Architecture Model Manager* (AMM), the *Architecture Adaptation Manager* (AAM), and the *Architecture Runtime Manager* (ARM) respectively. Self-adaptive systems are further augmented by a configuration graph model that maintains information about the history of a system's adaptations, as well as a body of architectural constraints that are intended to preserve core system capabilities. These artifacts are managed at runtime by the *Architectural Runtime Configuration Manager* (ARCM) and the *Architecture Constraint Manager* (ACM) respectively.

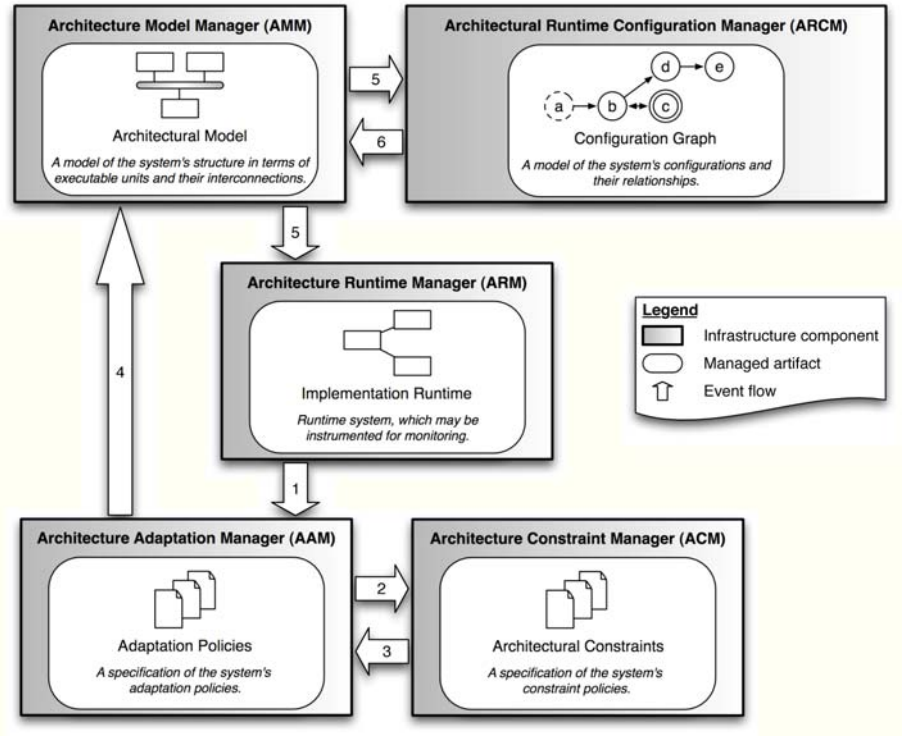


Fig. 2. The tools and activities of the PBAAM approach

3.1 Adaptation Policy Specification

One of the fundamental abstractions in our approach is the adaptation policy: a policy is an encapsulation of the system's reactive *adaptive behavior* and indicates a set of actions that should be taken in response to events indicating the need for these actions. The basic building blocks of adaptation policies are *observations* and *responses*. Observations encode information about a system and responses encode system modifications. Given the architecture-based focus of our approach, responses are limited in the kinds of actions they can perform: they are restricted to operations which change the structure of software architectures and, in essence, reduce to additions, removals, connections, and disconnections of architectural elements.

We specify the structure of adaptation policies using a xADL 2.0 schema [14] that extends the core schemas of the ADL and lays out policy structure. The basic definition of a policy appears below, with XML namespace information removed for the sake of brevity:

```
<xsd:complexType name="AdaptationPolicy">
  <xsd:sequence>
    <xsd:element name="description" type="Description"/>
    <xsd:element name="observationList" type="ObservationList"/>
    <xsd:element name="responseList" type="ResponseList"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="Identifier"/>
</xsd:complexType>
```

Each adaptation policy is characterized by a unique identifier and may contain an optional, human-readable textual description that indicates its purpose to designers. Each policy also contains a list of *observations* and a list of *responses*: when this list of observations is fully satisfied, the entire set of responses is enacted. It is very important to note that this policy schema is highly extensible and can be customized to fit the needs of specific projects. One of the extensions currently under development, for example, extends the notion of policies with support for expressing observations in terms of exhibited behaviors while modeling responses as a set of desired behaviors, where STATECHART models [15] are used to specify behavior.

3.2 Architectural Adaptation Management

The AAM is the element responsible for the runtime management of the specified adaptation policies. When the self-adaptive system is first instantiated, the AAM loads the set of policies and initiates their runtime evaluation: as policies are added and removed from the policy specification, the AAM updates the set of active runtime policies to reflect these changes. The current incarnation of the AAM adopts an expert system for the runtime management of policies. In a very straightforward manner, policies are translated to executable *condition-action* rules and then managed using an expert system shell. More specifically, we adopt the Java Expert System Shell (JESS) [16] for this task, which provides us with a well-tested and efficient platform for the runtime execution of policies.

In coordination with the ARM – an existing element of the ARCHSTUDIO environment that supports runtime evolution and predates our work – the AAM drives architectural change by enacting modifications to the system's architectural model. The ARM's primary responsibility is to ensure that changes enacted to the architectural model are also enacted on the runtime system itself.

Alongside these tools that implement a reactive self-adaptation loop, two additional tools provide capabilities related to constraint and configuration management. Before changes are actually enacted, the ACM ensures that these changes would not violate a body of architectural constraints. These constraints enforce a variety of desired architecture structural properties, ranging from component membership to architectural connectivity. Only those changes that do not violate these constraints are allowed to be enacted by the AMM. The ARCM tool is responsible for monitoring architectural changes as they take place, and recording them in a configuration graph. Nodes in this graph correspond to and

maintain information about architectural configurations, while edges represent adaptations between these configurations. Each edge maintains bi-directional *diff* information, which is used by ARCM to enact explicit user commands to apply rollback or rollforward operations on the architecture (an early description of ARCM appears in [17]).

3.3 Activity Flow

Referring to the activity flows indicated in Fig. 2 by numbered arrows, the adaptation process begins when observations about the running system are collected and transmitted to the AAM (flow labeled 1). These observations are gathered through independent probing elements or through self-reporting components and encapsulate what is known about the system. This information forms the basis for evaluating adaptation policies managed by the AAM.

Any triggered responses are first communicated to the ACM (indicated by the flow labeled 2), which ensures that these suggested responses do not violate the active architectural constraints. Modifications that are deemed allowable are communicated back to the AAM, and then finally enacted by being transmitted to the AMM (event flows 3 and 4, respectively). Both the ARM and ARCM are notified of any changes enacted on the architectural model (event flow 5). The ARM then ensures that these changes are reflected on the executing system, while the ARCM builds the system's configuration graph; ARCM also allows manual modifications triggered by the user (event flow 6).

4 Case Studies

This section offers specific details about two case studies in integrating self-adaptive capabilities with robotic systems. For each of the ArchWall and Archie systems, we will discuss our experiences and call out some of the difficulties we encountered and lessons we learned in providing novel self-adaptive capabilities in domains that did not previously support them.

4.1 Robocode

Our first study was performed using the Robocode[2] system. Initially developed as a Java teaching tool, Robocode is now an open-source system under active development that provides a robotic combat framework and simulator which is used to pit robotic control systems in battle against each other.

Robocode Background. Robocode provides a customizable simulated battlefield into which robots are deployed: the objective of robots is to remain alive while destroying their competitors. Each robot may move, use its radar to detect other robots, and use its gun to fire at opponents. The constraint of primary importance for each robot is the amount of remaining energy. While all robots

[2] http://robocode.sourceforge.net

begin a battle with the same level of energy, energy is depleted by being hit by bullets or colliding with other robots or walls. Energy can also be invested into firing bullets at other robots, but a multiple of this invested energy is recovered by successfully hitting. The goal of each robot, then, is to preserve its own energy by both firing wisely as well as avoiding collisions and enemy fire.

From a software development perspective, the ROBOCODE API provides builders with basic robot control capabilities: movement and steering, control for the robot's scanner and weapon, and support for notifications of battlefield events. How each robot responds to these events is the challenge of ROBOCODE development, and the robots developed by the community vary from the very simple to the very sophisticated. It is important to note that in development for the simulator, a robot is programmed and compiled as a single static unit of code that is then executed by the battle simulator; there is no consideration or support for runtime adaptation.

Architecting ARCHWALL**.** The core architecture of the ARCHWALL robot follows the guidance and constraints of the RAS architectural style (discussed in Section 2.1) and appears in Fig. 3.

In contrast to ROBOCODE development that focuses on robots being monolithically implemented as a single unit of source code, ARCHWALL is comprised of a number of independent components and connectors that are distributed between the JAVA and ROBOCODE environments. These components embody a number of behaviors and are arranged in RAS layers. At the lowest level, the skill layer captures the basic tasks that a ROBOCODE robot can perform: ARCHWALL can scan for other robots, turn its turret, fire at enemies, detect collisions, and control its speed and direction. Using these fundamental facilities, the reactive layer implements a simple collision recovery strategy of stopping and moving

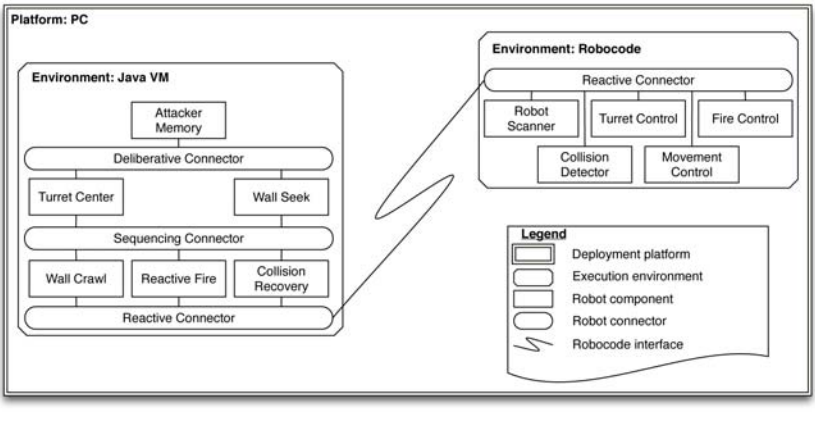


Fig. 3. The architecture of the ARCHWALL robot, showing the layered arrangement of components and connectors implementing robot behaviors, as well as the environments and platforms providing the execution context

away from collisions, firing at any enemy robot detected, and moving along a wall if the robot is near one. The sequencing layer contains components that ensure the turret is always pointed toward the center of the battlefield, and direct the robot to move to the nearest wall if it has not already done so. Finally, the deliberative layer maintains data about enemy robots that attack ArchWall. In accordance with the RAS style, components are arranged in layers according to the timeliness of their responses to events, and their maintenance of state.

The curved connection in Fig. 3 indicates the special-purpose interface we constructed in order to integrate the PBAAM infrastructure with the Robocode simulator (the integration also involved a number of modifications to the simulator framework itself). PBAAM requests for robot actions are translated into the appropriate Robocode API calls, while notifications of simulator events are translated into architectural events and transmitted to the robot's architecture.

With this architecture, ArchWall first seeks a battlefield wall and then follows it for the duration of the battle, embodying the movement behavior of a type of Robocode robots referred to as "wall-crawlers." The turret is always kept pointed toward the center of the battlefield, and ArchWall fires at any robot it detects. This initial set of behaviors is sufficient for the robot to compete in basic battles: in our testing experiences, the robot tends to rank between positions four and six in a battle against ten opponents selected from the set of sample Robocode robots that are distributed with the simulator.

Self-Adaptive ArchWall. The basic behaviors exhibited by ArchWall, while sufficient to be competitive, are certainly not optimal under a wide variety of battlefield conditions. For example, while firing at any enemy robot scanned is perfectly acceptable when the robot's energy is high, it would be helpful to exhibit a more careful firing strategy as the robot's energy is depleted. Similarly, while targeting the center is useful when there are many opponents on the battlefield, it becomes less desirable as the battle progresses and the number of enemies is lessened.

To address these shortcomings, we developed a self-adaptive version of ArchWall by placing the architecture of the robot under the management of the PBAAM toolset. This augmented architecture appears in Fig. 4; the core architecture of the robot appears in unshaded components, while the PBAAM tools are shaded. Along with the system, we deployed a number of adaptation policies addressing the need to change the firing and targeting behaviors of ArchWall as the conditions of the battle change. One policy, for example, states (in an abridged form for brevity):

```
<AdaptationPolicy id="ReplaceFiring">
  <ObservationList>
    <StringObservation>
      <StringObservationContent>
        (energy_report {energy < 60}) </StringObservationContent>
      </StringObservation>
```

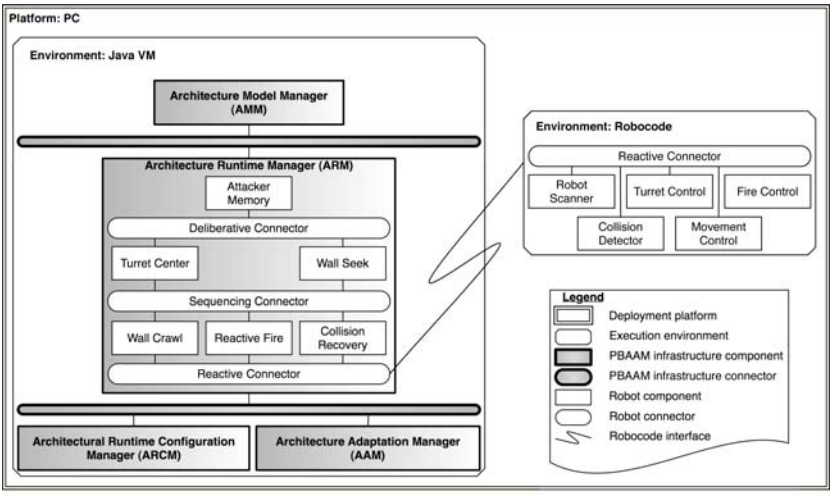


Fig. 4. The PBAAM-managed architecture of the ArchWall robot, showing the robot's architecture as well as the PBAAM tools that enable self-adaptive behavior

```
    <ResponseList>
      <RemoveComponentResponse>
        <RemoveComponent> ReactiveFire </RemoveComponent>
      </RemoveComponentResponse>
      <AddComponentResponse>
        <AddComponentIdentifier>
          DistanceReactiveFire
        </AddComponentIdentifier>
        <AddComponentType>
          DistanceReactiveFire_type
        </AddComponentType>
      ...
    </AddComponentResponse>
  </ResponseList>
</AdaptationPolicy>
```

This policy replaces the firing strategy used by ArchWall when the energy of the robot drops below the indicated threshold by replacing one component with another: *Distance Fire*, which only fires at enemies that are nearby in an attempt to maximize the chances of hitting (and, thereby recovering the invested energy) takes the place of *Reactive Fire*.

Additional adaptation policies also change the way in which the robot moves and scans for opponents as fewer enemy robots remain: in total, the ArchWall robot contains four independent adaptation policies which modify its behavior in different ways. Overall, the addition of the adaptation policies improves the

performance of the robot: the adaptive version of ArchWall tends to rank between positions two and four, while even coming in first on some test runs.

Most importantly, however, is the fact that each adaptation policy is completely independent of the architecture to which it is applied and could be added, removed, or modified during runtime as the robot continues to operate. Furthermore, the architecture and components of the robot are entirely adaptation unaware: None of the components needs to be modified for the transition from the non-adaptive to the adaptive version of ArchWall, and the only changes are those made through architectural means.

Developing the ArchWall robot clearly established the feasibility of integrating architecture- and policy-based self-adaptive software methods in robotic systems by providing novel support for developing self-adaptive Robocode robots. While this framework is admittedly limited to simulation, from a software engineering perspective it exhibits many of the same challenges that developing a self-adaptive system for any other robot does: coherently organizing and relating robot behaviors, for example, and dealing with multiple sources of input in deciding on which actions to perform.

The effort also gave us experience in dealing with an important architectural mismatch between the Robocode framework and the PBAAM infrastructure: Like most robotic system frameworks, robots supported by Robocode are developed synchronously by sequencing behaviors through their explicit ordering in the source code. This way of building systems conflicts with the asynchronous nature of our approach. As this asynchronous and modular nature is a fundamental enabler of runtime change, reconciling this mismatch was necessary and required effort in the design and implementation of each behavior in order to compensate. Each component had to be constructed in a state-based way – that is not necessary in other applications we have applied our approach to – that maintains information about the state of the interactions it is engaged in with components to which it has dependencies. This explicit maintenance of state for inter-component interactions is a key enabler for the integration of our work with robotics; decoupling this interaction modeling from components and isolating it at the architectural level is an area we are interested in pursuing in our future work.

4.2 Mindstorms NXT

We performed the second case study using the LEGO Mindstorms NXT development kit[3]. Released in the summer of 2006, this is another in LEGO's line of kits to support easily accessible and affordable development of robotic systems that has found great traction in academic settings.

Mindstorms NXT Background. Each Mindstorms kit is comprised of Technic pieces which are used to build the structure of robots, servo motors, and a variety of sensors (the commercial kit includes ultrasonic, light, sound, and touch sensors). Computer control for the sensors and motors is provided by an NXT brick: each brick supports enough ports to accommodate a maximum

[3] http://mindstorms.lego.com/Overview/

of three motor and four sensor connections, and also supports a USB port and a Bluetooth wireless connection. These kits are extremely affordable (at the time of this writing, the kits cost about $250 US) but resource constrained: processing is provided by a 32-bit ARM7TDMI microprocessor with 64KB of RAM available to it and 256KB of flash memory for non-volatile program storage.

From a software perspective, the basic platform supports development in two ways: the NXT processing brick firmware can execute user-written programs, and the development and compilation of these programs is supported through the NXT-G visual programming environment. More advanced users may leverage a large variety of third-party libraries and firmware replacements for a variety of programing languages. In the context of our discussion on self-adaptive systems, it is important to once more note that MINDSTORMS robots are developed as single units of code with no pre-existing support for or consideration of runtime change.

Architecting ARCHIE**.** Building on the work described in the previous section, we continued by developing the ARCHIE MINDSTORMS robot; a picture of the robot in our lab can be seen in Fig. 5.

The physical platform of the robot is a modification of a basic three-wheeled MINDSTORMS design. Movement is provided by two motors, each controlling one of the side wheels while the third wheel is unpowered and can freely rotate to any movement direction. A third motor opens and closes the grasping arm of the robot. The robot is equipped with the following sensors: A touch sensor which is mounted in place to detect when an object is within grasping range, a light sensor which detects the light reflectivity of the surface the robot is on, and an

Fig. 5. A picture of the ARCHIE MINDSTORMS robot: a three-wheeled design with a grasping arm and a number of mounted sensors along with the NXT processor brick

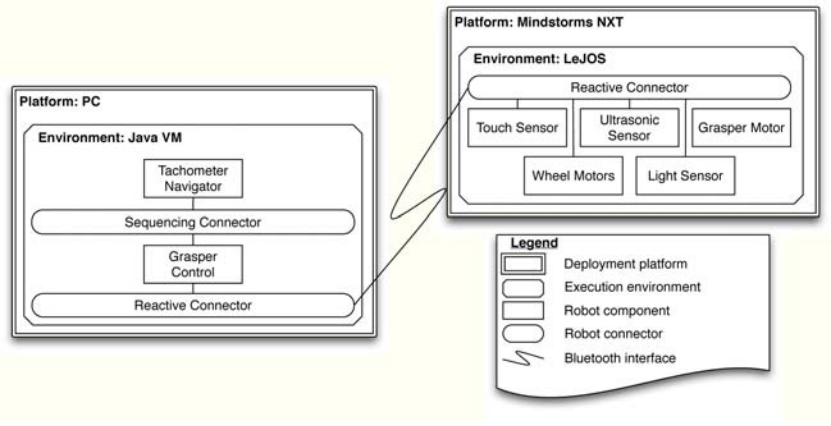


Fig. 6. The architecture ofthe ARCHIE robot, showing the robot's layered architecture distributed among two platforms and environments

ultrasonic sensor providing motion detection as well as range-finding in the front arc of the robot.

The software architecture of ARCHIE is built in the RAS style, and appears in Fig. 6. Once more, we abandon the traditional methods of building robot architectures in this domain, and construct ARCHIE in a component-based manner. The additional complication to integrating information providers and information consumers, in this domain, is the lack of the NXT brick's on-board processing power. As a result, ARCHIE adopts a tele-operation design: the bulk of the processing is performed on a PC running the PBAAM infrastructure, while the MINDSTORMS NXT brick is responsible for executing commands sent to its actuators and transmitting sensor data over the robot's Bluetooth connection. The deployment can be seen in Fig. 6, where the Bluetooth connection is indicated by the curved connection connecting the multi-platform architecture. This divide between the adaptive system and the facilities governing adaptation is not an essential challenge to developing robots using this approach, but was a solution driven by the limited resources of the NXT brick, which prevented us from deploying the PBAAM toolset on-board.

While limiting the range of the robot to that of the Bluetooth connection (roughly 10 meters), the solution was more than adequate for us to demonstrate self-adaptive behavior in our lab. We adopted the LEJOS ICOMMAND API (version 0.6) – a JAVA implementation of an NXT Bluetooth interface – and we replaced the default firmware of the MINDSTORMS platform with the LEJOS NXJ firmware update (version 0.4)[4].

With this architectural configuration, ARCHIE travels to a pre-defined location in our lab and grasps objects (in this case, small balls) if they are there. If it finds the object at the indicated location, it delivers it to its starting location. Navigation is implemented by the *Tachometer Navigator* component in the

[4] http://lejos.sourceforge.net/

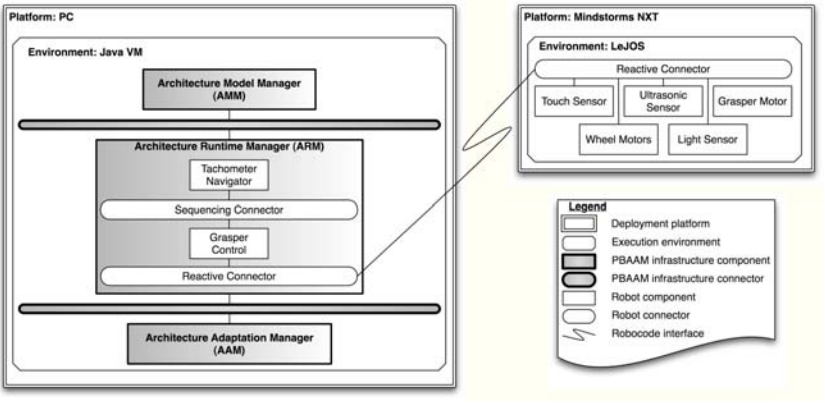


Fig. 7. The PBAAM-managed architecture of the ARCHIE robot, showing how the basic architecture is managed by the PBAAM tools in order to enable self-adaptive behavior

sequencing layer, which keeps track of the robot's location based on tachometer information from its motors. The *Grasper Control* component of the reactive layer provides control for the robot's grasping arm, and basic robot functionality resides in the skill layer.

Self-Adaptive MINDSTORMS**.** The need for adaptation in the ARCHIE robot is naturally motivated by the nature of the architecture itself. Due to driver incompatibilities between the Bluetooth libraries used by the NXT and the PC we used for development (running Mac OS X 10.4), the connection is unreliable and exhibits intermitted data loss and errors. Since the *Tachometer Navigator* component relies on reports from the robot's motors in order to calculate positioning data, corrupted information means that position information can be grossly mistaken, which makes navigation impossible. With this failure (currently self-diagnosed by *Tachometer Navigator* through a determination of whether data conforms to a reasonable envelope), the robot can no longer correctly navigate nor find its way back to the starting position.

One way to address this failure is through architectural self-adaptation. To this end, we developed a self-adaptive version of ARCHIE by managing the architecture with the PBAAM toolset: this augmented architecture appears in Fig. 7. We use PBAAM's adaptation management tools to deploy an adaptation policy that restores navigational support to the robot when *Tachometer Navigator* fails. The policy definition, elided for brevity, follows:

```
<AdaptationPolicy id="ReplaceNavigation">
  <ObservationList>
    <StringObservation>
      (navigation_report {failure ==true})
    </StringObservation>
```

```
  </ObservationList>
  <ResponseList>
    <RemoveComponentResponse>
      <RemoveComponent>
        Tachometer Navigator
      </RemoveComponent>
    </RemoveComponentResponse>
    <AddComponentResponse>
      <AddComponentIdentifier>
        Ultrasonic Navigator
      </AddComponentIdentifier>
      ...
    </AddComponentResponse>
  </ResponseList>
</AdaptationPolicy>
```

The new *Ultrasonic Navigator* component maintains no state information – which is the reason it is inserted into a lower RAS layer – but simply locates a wall using the robot's ultrasonic sensor and continues to follow the lab's walls until it locates the starting location that it detects using the light sensor to measure the reflectivity of the floor (the starting location is wall-adjacent and is more reflective than the remainder of the lab's floor). ARCHIE's components are also designed in the state-based way used for the ROBOCODE platform to account for synchronicity assumptions in the underlying development framework.

As with the ROBOCODE case study, our goal was to establish the feasibility of applying architecture-based self-adaptation techniques to a domain in which they had not previously been demonstrated, namely autonomous mobile robotic systems. So, while the architecture and behavior of ARCHIE are simple, they nevertheless demonstrate a practical self-adaptive solution to a practical problem as well as a successful application of our tools and techniques in this domain. And, despite the simplicity of the platform, we are confident our feasibility claim is valid due to the number of difficulties and challenges our MINDSTORMS robot shares with more complex robotic systems, such as the:

- necessity of integrating data from multiple sensors (sensor fusion) to determine courses of action;
- demands on timely actions in response to sensor information so that the robot's actions are current and actions are not performed too late, and;
- unreliability of sensor information and communication channels that the robot must account for in its control system.

These are just some examples of the kinds of difficulties shared by both real-world robotic systems and the types of MINDSTORMS robots we are developing. These systems are real, mobile, unreliable and resource constrained platforms, which is why we feel justified in the validity of a case-study using this platform.

5 Discussion

This section explores some of the interesting questions and trade-offs we encountered and offers some of our insights into the cross-section of self-adaptive architectures and robotic systems.

5.1 Architectural Mismatch

One of the difficulties we faced in our case studies was the architectural mismatch between the robotic frameworks we were working with and the assumptions of an architecture- and component-based approach. The PBAAM development framework was designed and built to support the development of component-based systems that use asynchronous events for communication and have no assumptions of shared state between them. These high-level design principles are critical in enabling the degree of modularity and decoupling that our work in developing self-adaptive systems relies on.

Robotic systems, on the other hand, tend to be constructed with architectural assumptions about synchronicity and strict temporal ordering of operations in mind. Many robotic libraries, for example, define interfaces to actuators and sensors that operate in a blocking manner: control is not returned until they have completed execution, which is particularly problematic in the case of long-term actions such as movement. This allows systems to be designed with great ease as long as they're constructed in a monolithic manner: it is quite easy to develop behaviors by chaining together a sequence of operations when these operations are invoked sequentially from a single unit of code. It is significantly more challenging, however, to compose these behaviors when fine-grained actions are distributed among many independent and potentially distributed components.

This mismatch between the fundamental design decisions of these domains was challenging to overcome, and required care in component design to account for the lack of synchronicity and the lack of guarantees about event ordering. The benefit, of course, from investing in this effort is the higher degree of modularity and enablement of runtime adaptation that this approach to building systems supports.

The conclusion from this necessity for more effort, of course, is not to dismiss the integration of architecture-based adaptation with robotics, but to recognize when the trade-off becomes worthwhile. Unless there is a driving need for self-adaptive behavior – and therefore the necessity to support a great deal of modularity and runtime evolvability – the effort to build robotic systems in a component-based manner and according to the RAS style while having to bridge this mismatch is not worth the benefits. It is interesting to note that a great deal of effort is currently being invested by the robotics community toward supporting the development of component-based robots and integrating software engineering technologies in their construction, as exemplified by the IEEE RAS TC-SOFT[5].

[5] http://robotics.unibg.it/tcsoft/

5.2 Platform Selection

Our goal of examining the integration of architecture-based adaptation with robotics – rather than the development of industrial-strength robots that is work best left to domain experts – guided our selection of the MINDSTORMS system for experimentation: The platform compares favorably with other commercially available platforms and development tool sets such as the iROBOT or K-TEAM platforms in terms of flexibility as well as cost. While there is a clear loss of sturdiness compared to these pre-manufactured robots, the MINDSTORMS platform provides a degree of flexibility that other packages can't match: a new type of physical structure is only a matter of a few hours of (fun) re-assembly. Most important is the degree of broad availability and appeal of the LEGO kits: the kit supports both Windows and Macintosh platforms in a variety of programming languages, is supported by a large and dedicated community, has sold millions of units throughout the years, and seems to quickly capture the imagination and attention of those exposed to it. As we are interested in both disseminating our research tools and techniques as well as teaching self-adaptive architectures to university students, the MINDSTORMS platform is an ideal and affordable choice for such needs.

6 Future Work

There are a number of directions we plan on pursuing in continuing our work in the intersection of robotic systems and self-adaptive architectures. In the long term, the most important aspects of robotic construction which must be addressed involve examining, understanding, and improving the scalability and reliability of these architectures. This is both an issue of learning more about the implications of using architecture-based techniques in the robotics domain as well as a matter of refining our notations and tools: we expect, for example, to refine our conceptualization of constraints and express them in terms of behavior as opposed to structure.

We also plan on further strengthening our feasibility claims by continuing to build more sophisticated and complex robotic architectures. One such avenue of development involves the construction of mixed deployment, distributed architectures in which the more computationally intensive elements of a robotic architecture remain on a PC platform and communicate through the wireless connection, but where more efficient elements are deployed on the robot platform itself. We hope that these hybrid architectures will ease the difficulty of bridging the architectural mismatch we discussed in the previous section, and significantly improve the performance and stability of our robots.

7 Conclusion

The work presented in this paper is an exploration into the application of architecture-based techniques to the robotics domain in order to support the

development of self-adaptive robotic systems. One of the critical challenges of this effort is supporting the dynamic modification of adaptive behaviors during runtime.

Our previous work in architecture-based self-adaptive systems has been focused on supporting exactly this capability through the use of adaptation policies that are decoupled from the architectures they relate to; these policies are independently managed and the tools we have developed support their runtime modification. We therefore applied this work to the robotics domain by performing two case studies: The first focused on developing a self-adaptive robot for the ROBOCODE robotic combat simulator, while the second involved the development of a self-adaptive MINDSTORMS NXT robot.

We believe that our efforts were successful in multiple ways. Most importantly, our case studies establish the feasibility of applying an architecture-based approach to self-adaptive robotic software by demonstrating practical solutions to practical robotic problems. Furthermore, our work also contributes an initial understanding of the difficulties involved in transitioning our particular approach – and others component-based approaches like it – to the robotics domain due to the architectural mismatch between the assumptions of architecture-based approaches and the actual practice of robotic system development. Finally, we continue to realize that the policy language of our approach is adequate for specifying adaptive behavior in a variety of settings.

The field of robotics is a rich application domain for software engineering research which provides dividends for both communities. Robotic software development greatly benefits from the application of software engineering research to the challenges of the domain, and software engineering researchers gain rigorous test settings and realistic scenarios for their work in a domain that stresses issues such as safety, reliability, and adaptation completeness which are more relaxed in other settings. We plan to continue to examine the intersection of robotic control systems and self-adaptive architectures, particularly with the MINDSTORMS platform as an easily accessible and inexpensive experimental testbed.

Acknowledgments. The authors would like to thank Eric Dashofy for his work on ARCHSTUDIO and André van der Hoek for his contributions to the conceptualization of our approach. This work sponsored in part by NSF Grants CCF-0430066 and IIS-0205724.

References

1. Georgas, J.C., Taylor, R.N.: Towards a Knowledge-Based Approach to Architectural Adaptation Management. In: Proceedings of ACM SIGSOFT Workshop on Self-Managed Systems (WOSS 2004), Newport Beach, CA (October 2004)
2. Nilsson, N.J.: Principles of Artificial Intelligence. Tioga Publishing Company (1980)
3. Brooks, R.A.: A Robust Layered Control System for a Mobile Robot. IEEE Journal of Robotics and Automation 2(1), 14–23 (1986)

4. Firby, R.J.: Adaptive Execution in Complex Dynamic Worlds. PhD thesis, Yale University (1990)
5. Georgas, J.C., Taylor, R.N.: An Architectural Style Perspective on Dynamic Robotic Architectures. In: Proceedings of the IEEE Second International Workshop on Software Development and Integration in Robotics (SDIR 2007), Rome, Italy (April 2007)
6. Taylor, R.N., Medvidovic, N., Anderson, K.M., James, E., Whitehead, J., Robbins, J.E., Nies, K.A., Oreizy, P., Dubrow, D.L.: A Component- and Message-Based Architectural Style for GUI Software. IEEE Transactions on Software Engineering 22(6), 390–406 (1996)
7. Wermelinger, M., Lopes, A., Fiadeiro, J.L.: A Graph Based Architectural (Re)configuration Language. In: ESEC/FSE-9: Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 21–32. ACM Press, New York (2001)
8. Georgiadis, I., Magee, J., Kramer, J.: Self-Organising Software Architectures for Distributed Systems. In: WOSS 2002: Proceedings of the First Workshop on Self-Healing Systems, pp. 33–38. ACM Press, New York (2002)
9. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure. IEEE Computer 37(10) (2004)
10. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An Architecture-based Approach to Self-Adaptive Software. IEEE Intelligent Systems 14(3), 54–62 (1999)
11. Kim, D., Park, S., Jin, Y., Chang, H., Park, Y.S., Ko, I.Y., Lee, K., Lee, J., Park, Y.C., Lee, S.: SHAGE: a Framework for Self-Managed Robot Software. In: SEAMS 2006: Proceedings of the 2006 International Workshop on Self-Adaptation and Self-Managing Systems, pp. 79–85 (2006)
12. Kramer, J., Magee, J.: Self-Managed Systems: An Architectural Challenge. In: Future of Software Engineering (FOSE 2007), pp. 259–268 (2007)
13. Georgas, J.C.: Supporting Architecture- and Policy-Based Self-Adaptive Software Systems. PhD thesis, University of California, Irvine (2008)
14. Dashofy, E.M., Hoek, A.v.d., Taylor, R.N.: A Comprehensive Approach for the Development of Modular Software Architecture Description Languages. ACM Transactions on Software Engineering and Methodology (TOSEM) 14(2), 199–245 (2005)
15. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming 8(3), 231–274 (1987)
16. Hill, E.F.: Jess in Action: Java Rule-Based Systems. Manning Publications Co., Greenwich (2003)
17. Georgas, J.C., van der Hoek, A., Taylor, R.N.: Architectural runtime configuration management in support of dependanble self-adaptive software. In: Proceedings of ACM SIGSOFT Workshop on Architecting Dependable Systems (WADS 2005), St. Louis, MO (May 2005)

A Case Study in Goal-Driven Architectural Adaptation

William Heaven, Daniel Sykes, Jeff Magee, and Jeff Kramer

Department Of Computing, Imperial College London
{william.heaven, daniel.sykes, j.magee, j.kramer}@imperial.ac.uk

Abstract. To operate reliably in environments where interaction with an operator is infrequent or undesirable, an autonomous system should be capable of both determining how to achieve its objectives and adapting to novel circumstances on its own. We have developed an approach to constructing autonomous systems that synthesise tasks from high-level goals and adapt their software architecture to perform these tasks reliably in a changing environment. This paper presents our approach through a detailed case study, highlighting the challenges involved.

1 Introduction

In many defence and civilian organisations in the modern world, people are asked to venture into treacherous environments. Understandably, such organisations are turning to automated means to achieve their goals. Previously, the deployment of autonomous systems in real-world environments was limited by their inability to cope with the complexity and variability in the environment; something which humans deal with naturally. Hence, much recent work [1,2,3,4,5] has been driven by the principle that the (partial) failure of the system *as designed* is to be expected, and that mechanisms to adapt when such failures occur must be provided.

We have developed an approach to constructing self-managing systems that is underpinned by a three-layer model, a standard means of abstraction in robotics [6]. Our three layers—*goal management*, *change management*, and *control*—partition the system in a way that allows different kinds of change in the environment to be handled by different levels of adaptation, from low-level sense-react mechanisms in the control layer to more deliberative and potentially time-consuming reassessment of how the system might continue to meet its high-level goals. For example, if a mobile robot finds its path blocked by a small obstacle then it should be able to detect and avoid this obstacle using mechanisms in its control layer such as sensors, actuators, and basic obstacle avoidance algorithms. However, if the robot finds its navigation system incapacitated, perhaps due to loss of GPS, then it may have to reassess the ways in which it might achieve its goal, perhaps making use of a different set of capabilities or an alternative decomposition of subgoals. Both these cases can be thought of as instances of adaptation. By partitioning a system into three layers, clear distinctions can be

B.H.C. Cheng et al. (Eds.): Self-Adaptive Systems, LNCS 5525, pp. 109–127, 2009.

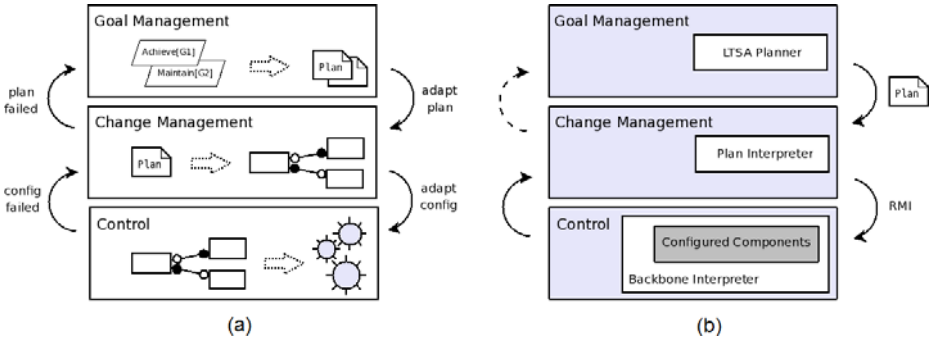


Fig. 1. Overview of conceptual layers (a) and current implementation (b)

made between types of goal-threatening "obstacle" and, accordingly, between means to cope with them.

The three layers are shown in Figure 1(a). The *goal management* layer handles adaptation at the top level, which typically demands relatively sophisticated analysis, such as planning to meet a new set of goals or replanning to continue to meet existing ones. This layer consists of planning mechanisms that synthesise tasks from high-level goals, given a model of the environment of the system.

While the goal management layer concerns adaptation between different tasks, the *change management* layer concerns adaptation within the current task. This layer interprets plans generated by the goal management layer and automatically selects a suitable configuration of software components to execute these plans. There are two types of adaptation that can be handled by this layer. Firstly, the plans used in our approach are *reactive plans* [7,8,9], which means that a single plan can include alternative means of satisfying a goal. When one branch of a plan fails, this layer manages the adaptation necessary to try an alternative. Secondly, it is often the case that more than one component is available to implement a certain capability. When use of an instantiated component is no longer feasible, it may thus be possible to swap in an alternative without interrupting execution of the current plan.

Finally, the *control* layer is the level of execution of the selected component configuration. Low-level adaptive mechanisms such as obstacle avoidance may be implemented here, along with any other behaviour required to carry out the current plan, as prescribed by the layers above.

In addition to the downward communication between the layers—the provision of new plans and new component configurations—there is also feedback communicated upwards from the lower layers. The control layer provides information about the environment to the middle layer, which it uses to decide how to continue executing the current plan. If the environment changes in such a way that this is no longer possible, the middle layer provides diagnostic information about this failure to the goal management layer, which can use this to update its model of the environment and replan.

Our engineering approach is motivated by the general need for self-managing systems to be able to adapt their component configuration automatically. The approach is intended to be applicable to a wide range of such systems, from unmanned vehicles to self-managing service-oriented web applications. However, since we have had the opportunity to experiment extensively with a testbed of mobile robots, the case study presented here draws on our experiences in the former application area. Our experimentation to date has demonstrated the feasability of synthesising software architectures from high-level goals as a means of adaptation and, in particular, the adoption of the three-layer model as a means of abstraction for the engineering of self-managing systems. Finally, it should be noted that our current implementation for the mobile robot testbed uses a centralised architecture. Ongoing work will extend this implementation to address decentralized network-based applications.

An overview of our implementation is shown in Figure 1(b). Domain modelling and planning are performed using an extension to the Labelled Transition System Analyser (LTSA) [10]. The middle layer consists of the plan interpreter and the mechanisms for configuration generation, while the domain-specific components are instantiated using the Backbone interpreter [11] in the bottom layer. Components are implemented in Java, following the model prescribed by the Backbone language [11]. Certain steps in the feedback loop between the middle and upper layers, such as updating the domain model, have not yet been fully automated and will not be demonstrated in this case study.

Most of the technical aspects of our approach have been described before [8,9], but the purpose of this paper is to illustrate its applicability through a detailed case study. Following a summary of related work in Section 2, we present the case study secenario in Section 3. Sections 4 and 5 then describe the mechanisms of the upper two layers necessary to bring about adaptation. In Section 6 we give an overview of the how the system executes in the scenario, describing an adaptation to the changing environment. We conclude in Section 7.

2 Related Work

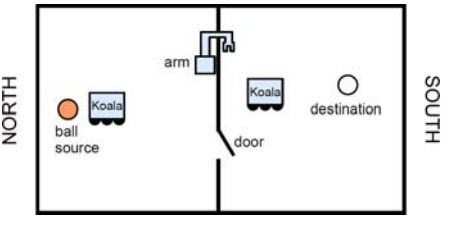
The three-layer model is now widely used in the robotics community [6], where it developed from the early sense-plan-act (SPA) and subsumption [12] architectures. An SPA system consists of a single control loop where sensed data passes to an analysis or deliberation component, which then determines what actions to perform. The main limitation of this approach was that it could not react quickly enough for low-level behaviours. Subsumption was proposed to mitigate this by introducing several layers. The lowest layer was concerned with tight feedback loops such as obstacle avoidance, while the highest layers dealt with abstract goals. However, subsumption suffered from difficulties in specifying the interplay between the layers, and so it was not easy to modify a given program.

As we observed in [8] and [9], many previous authors have described approaches which assume adaptation can be specified and analysed before the system is deployed. Unfortunately, as in our chosen scenario, this is not always the case with autonomous systems.

Garlan and Schmerl [1,13] achieve dynamic change by describing an architectural style [2] for a system and a repair strategy. The repair strategy is a script which modifies the architecture in response to changes in the monitored system properties. The Rainbow approach, by Cheng *et al.* [14,15], extends this idea by calculating for each strategy a utility, in terms of non-functional properties, so that the most appropriate repair can be applied. Dashofy *et al.* [16] use an architectural model and design critics [17] to determine whether a set of changes (an architectural 'delta') is safe to apply to a given configuration. Georgas and Taylor [4] describe a system where change is enacted by architectural policies which are invoked in response to certain events such as component failure. Georgiadis [3] describes a distributed approach to enforcing architectural constraints, though ultimately repairs are specified by the programmer.

The previous work that perhaps bears most resemblance to our combined approach is that by Garlan *et al.* [18], where adaptations are driven by a change of goal, and that by Arshad *et al.* [5,19] where adaptations are found by resorting to replanning. Unfortunately, planning for all reconfigurations comes at some cost, as it corresponds to an SPA architecture in an adaptive context.

3 Scenario



We explain our approach with reference to a humanitarian crisis in which a house has collapsed—perhaps due to an earthquake or military action—and an aid package must be transported to a trapped survivor. Clearly, it is imperative that the package reach its destination, but the task cannot be entrusted to a human being since the environment is dangerous. Hence, an autonomous system is required, which must ensure reliability by adapting to changing circumstances. Three robots are available to perform this task: two mobile Koala robots (`www.k-team.com`) and a fixed Katana arm (`www.neuronics.ch`). The situation is complicated by the presence of a wall between the two locations (which could be regarded as rooms), with a door allowing robots to pass from one side to the other. However, the door cannot be opened by the Koala robots, and may open or close in an unpredictable way. Meanwhile, the Katana arm is able to reach over the wall and hoist the package (represented by a coloured ball) from one robot to the other if necessary. We further complicate the situation by not providing the robots with the position of the door (though the position of the ball source and target are known).

Note that there are at least two ways the goal can be achieved: by carrying the ball through the door after locating it, or by carrying the ball to the arm and having it transferred to the second robot, which is able to transport the ball to the target. We will show how these can be derived as two alternative decompositions of an abstract plan, itself derived from the user's goal. The actions of these

decompositions will represent the concrete operations available to the system such as moving to a known location, searching for an object or location, avoiding obstacles and picking up and putting down balls.

4 Modelling the Environment and Generating Plans

We do not consider adaptation to occur blindly, as it might, for instance, if one were to persue a genetic-algorithms-based approach to self-adaptation. Rather, adaptation is explicitly determined by the requirements of a system, which are determined in turn by the system's goals and its operating environment. For an autonomous system to satisfy its goals over time—which involves determining when, and how, to adapt to keep those goals satisfied—it must maintain a representation of this environment. We call such a representation a *domain model*. The system uses its current domain model and goal specification to generate reactive plans.

As described in [8], a domain model for a non-trivial environment is typically composed of a small hierarchy of sub-models. Models higher in the hierarchy are more abstract representations of the environment than those below them. At the top level, we can abstract away variables such as the number of Koala robots available, or details of their individual capabilities. This allows the abstract domain model to represent only those characteristics of the environment salient to a general expression of the problem domain, which means, in turn, that we are able to generate a correspondingly general top-level plan.

Actions of an abstract domain model are decomposed into actions of a concrete domain model, the latter specifically describing details of available hardware. For example, one particular decomposition of *moveToTarget* includes the sequence of actions *loadBallAtX*, *startGotoTarget*, *unloadedBallAtTarget*, where the first and third actions assume a Koala with loading and unloading capabilities, and the second action assumes a Koala with its own long-range navigation capabilities. In the following exposition, we consider only the abstract plan for the scenario.

4.1 Modelling the Environment

A domain model represents the essential characteristics of a system embedded in an environment and can be thought of as a representation of that environment from the system's point of view. It informs the system of the behaviours available to it and the ways in which these behaviours can both affect and be affected by the environment. The domain model for a system is represented as a finite state machine, capturing the set of logical states in which the system and environment can be (given a predefined set of propositions of interest) and the ways in which the system can move between these states (given a set of actions that the system can control). Formally, a *domain model* is a structure $D = \langle Props, States, Actions, Trans \rangle$ where $Props$ is a finite set of propositions describing (discrete) properties of the system and environment, $States \subseteq 2^{Props}$ is a set of states, $Actions$ is a finite set of actions, and $Trans \subseteq States \times Actions \times States$ is a transition relation.

Domain Properties and Actions. The set of properties *Prop* are either discrete abstractions of (possibly continuous) data that the system is able to sense, such as *BallAtTarget* or *DoorOpen*, or execution states of the system itself, such as *MovingToTarget* or *HoistingSouth*. The boolean values of domain properties can be changed by the occurrence of actions.

The set of actions *Actions* is partitioned by the set $\{Sys, Env_{Dep}, Env_{Ind}\}$, where *Sys* is a set of *system actions*, Env_{Dep} is a set of *causally-**Dep**endent environment actions*, and Env_{Ind} is a set of *causally-**Ind**ependent environment actions*. System actions, as might be expected, are those actions controllable by the system. A system action typically initiates some behaviour of the system. For example, the system action *moveToTarget* initiates the system behaviour of "moving to the target". Environment actions, on the other hand, are not controllable by the system (note that environment actions are indicated by an underscore prefix to the action label, e.g., *_doorOpened*). We further distinguish between causally-dependent and causally-independent environment actions.

A causally-dependent environment action is an action that can only occur once a system action has initiated some particular behaviour. For example, the environment action *_arrivedAtTarget* can only occur when *MovingToTarget* holds. Causally-dependent actions play the role of "notifications", which are monitored by the system to determine whether or not the objective of a particular behaviour has yet been reached. For example, the occurrence of *_arrivedAtTarget* can be thought of as the notification of successful completion of the behaviour initiated by the system action *moveToTarget*. In practice, an action such as *_arrivedAtTarget* coincides with the system's sensing that it has arrived at the target location.

In contrast, a causally-independent environment action is an environment action that is not constrained in any way by the behaviour of the system. For example, the environment action *_doorOpened* can occur whenever the door is closed, irrespective of the behaviour of the system at that point. Causally-independent actions are somewhat akin to *input* actions in I/O automata [20], though the latter are typically fully unconstrained, with all input actions enabled in every state.

Generating Domain Models. Our current implementation uses the Labelled Transition System Analyser (LTSA) [10] to generate an LTS representation of a domain model from a set of actions, a set of *fluent* propositions [21], and a set of LTL constraints. The set of actions for the case-study domain is partitioned along the lines described above as follows:

```
set Sys = {moveToTarget, moveToArmNorth, moveToArmSouth, moveNorth, moveSouth,
       hoistSouth, hoistNorth}
set Env_Dep = {_arrivedTarget, _arrivedArmNorth, _arrivedArmSouth, _arrivedSouth,
       _arrivedNorth, _hoistedNorth, _hoistedSouth}
set Env_Ind = {_doorOpened, _doorClosed}
set Actions = {Sys, Env_Dep, Env_Ind}
```

The only causally-independent environment actions we care about in this environment are *_doorOpened* and *_doorClosed* since the property *DoorOpen* needs to be monitored.

A fluent f represents a property of the environment and is defined by a set of initiating actions I and a set of terminating actions T such that f holds at all states along a path through a given LTS between an occurrence of an action $e \in I$ and an occurrence of an action $e' \in T$. A sample of the set of fluents defined for the case-study domain is shown below (where each fluent definition has the form *fluent* $f = <I,\ T>$):

```
fluent BallAtTarget = <{_arrivedAtTarget}, {moveToArmSouth, moveToArmNorth, moveNorth}>
fluent InNorth = <{_arrivedNorth, _hoistedNorth}, {_arrivedSouth, hoistSouth}>
fluent DoorOpen = <{_doorOpened}, {_doorClosed}> initially true
```

The fluent *BallAtTarget*, for example, becomes true whenever *_arrivedAtTarget* occurs and false whenever any of the actions *moveToArmSouth*, *moveToArmNorth*, or *moveNorth* occur. Note that, by default, a fluent is false in the initial state of the LTS but can explicitly be defined to be true, as is the case for the fluent *DoorOpen*.

While fluents describe the way properties change value, they do not constrain actions. To do this, a set of LTL constraints is defined that describes the conditions under which actions may occur. In the following, `[]` is the "always" operator, `X` is the "next" operator, and `W` is the "weak until" operator.

```
constraint C1 = ([] (!X moveToTarget W !BallAtTarget && InNorth && !MovingToTarget))
constraint C2 = ([] (!X _arrivedAtTarget W MovingToTarget))
...
constraint C11 = ([] (!X _doorOpened W !DoorOpen))
constraint C12 = ([] (!X _doorClosed W DoorOpen))
```

Constraint *C1*, for example, expresses a precondition for the system action *moveToTarget*: this action cannot occur until *InNorth* is true and *BallAtTarget* and *MovingToTarget* are both false. T he domain model does not only constrain system actions. As can be seen, it is necessary to specify the preconditions of environment actions (whether in Env_{Dep} or Env_{Ind}) so that we model the environment as accurately as possible. Causally-dependent actions, such as *_arrivedAtTarget* must be constrained so that they cannot occur until the system is behaving in a way that makes their occurrence appropriate. Here, for example, the precondition for *_arrivedATarget* is that *MovingToTarget* holds. Finally, though causally-independent actions are not constrained by the behaviour of the system, they must follow their own logic: here *_doorOpened* is specified to occur only when the door is closed, i.e., *DoorOpen* is false, and, conversely, *_doorClosed* is specified to occur only when the door is open.

While LTL constraints specify the preconditions of actions, fluent definitions —in addition to defining the properties of the domain—also specify the

postconditions of actions by describing how fluents are affected by the occurrence of actions. For example, it can be seen from the sample fluent definitions above that the postcondition of the causally-dependent environment action *_hoistedNorth* implies $InNorth \wedge \neg HoistingSouth$, since *_hoistedNorth* is in the set of initiating actions for *InNorth* and set of terminating actions for *HoistingSouth*.

A detail of the LTS of the abstract domain model for our scenario is shown in Figure 2(a). As mentioned above, the abstract domain model allows two routes to the target: one involves moving the ball through the door, the other involves hoisting the ball over the wall using the arm. In the LTS shown, the former starts from state 0, in which *DoorOpen* is true, the latter from state 1, in which *DoorOpen* is false.

4.2 Generating Reactive Plans

The key advantage of a reactive plan over traditional linear plans is that a reactive plan includes paths to the goal states from every state in the domain from which those goal states are reachable. In a sense, a reactive plan might be considered a set of linear plans to a common set of goal states. It is this feature of reactive plans that allows for, and drives, adaptation in the change management layer: selecting an alternative path when one fails can allow the system to continue to satisfy its goals, though execution of the alternative path might require system reconfiguration.

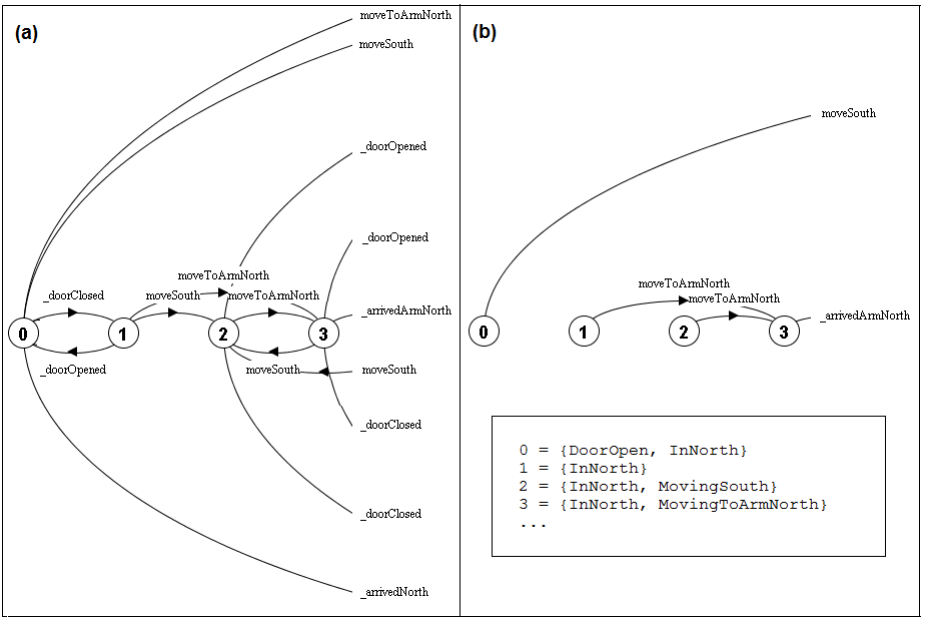


Fig. 2. Detail of domain model (a) and pruned domain model or "plan" for goal Achieve[$BallAtTarget$] (b)

In essence, a reactive plan is a pruned domain model from which all paths that (a) include causally-independent environment actions or (b) are not shortest paths to a goal state are removed. To generate reactive plans from a domain model, we have extended LTSA with an implementation of algorithms adapted from the planning-as-model-checking community [22,7]. The extended LTSA (LTSA Planner) uses the existing analysis features of LTSA to identify goal states in a domain model LTS and then a newly implemented backtracking and pruning facility to construct a reactive plan to attain these goal states. Figure 2(b) shows a detail of the pruned LTS for the abstract domain model in Figure 2(a). Recall that the goal states for the case study are those in which *BallAtTarget* holds. The shortest path from state 0 to a state where this fluent holds requires the *moveSouth* action. However, a precondition of this action is that *DoorOpen* is true. Thus, in state 1, where this is not the case, the shortest path not involving a *_doorOpen* or *_doorClose* action requires the *moveToArmNorth* action.

Formally, a reactive plan is a map $Plan : States \rightarrow Sys \cup Env_{Dep} \cup \{\mathsf{DONE}\}$ where DONE represents a null-action allowing $Plan$ to be applied to goal states. For all $s \in States$ and $a \in Sys \cup Env_{Dep}$, $Plan(s) = a$ only if $Trans(s, a, s')$ for some $s' \in State$. In other words, a state is associated with an action in the generated plan only if that action can be performed in that state (according to the domain model). If defined, $Plan(s)$ thus denotes the action to be taken by the system whenever it is in state s in order to meet the goal for which the plan was generated.

At this point, the importance of the distinction between kinds of action becomes clear. The aim is to generate a reactive plan to be used by the system to achieve and maintain given goals so it might be assumed that only system actions should appear in a generated plan. However, while causally-independent environment actions are indeed excluded, causally-dependent environment actions are required for plan execution. For example, while a causally-independent action such as *_doorOpened* will not appear in a generated plan, causally-dependent actions such as *_arrivedAtTarget* do. This is because a path to a goal is effectively a sequence of system behaviours—and system behaviours must be initiated (by system actions) and terminated (by causally-dependent environment actions). A path to a goal thus consists of a sequence of alternating system and causally-dependent actions.

A reactive plan can equally be thought of as a set of condition-action rules, where each condition represents a logical state in the domain model. Execution of a reactive plan involves determining the current logical state of the system and performing the action prescribed for that state by the corresponding condition-action rule. An extract of the generated plan for the abstract domain is given below. The first two rules shown here correspond to the first two states of the pruned domain model LTS shown in Figure 2(b). (There is, of course, also a state in the pruned domain model LTS to which Rule 14 corresponds, but it is not shown in the above figure.) There are 23 condition-action rules in total, one for each state of the domain model. In the extract below, the domain property conjuncts that are true in each condition have been highlighted for legibility.

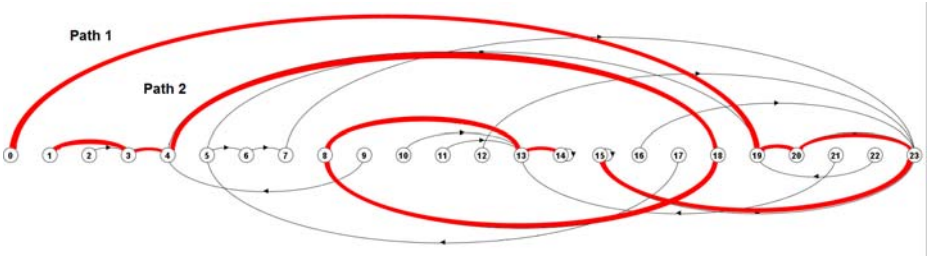


Fig. 3. Plan LTS showing alternative paths to goal states

```
/* Rule 0 */ !MovingSouth && DoorOpen && InSouth && !MovingToArmSouth
    && !BallAtArmSouth && !MovingNorth && !MovingToTarget && !InNorth && !MovingToArmNorth
    && !BallAtArmNorth && !HoistingSouth && !BallAtTarget && !HoistingNorth
-> moveSouth
/* Rule 1 */ !MovingSouth && !DoorOpen && InSouth && !MovingToArmSouth
    && !BallAtArmSouth && !MovingNorth && !MovingToTarget && !InNorth && !MovingToArmNorth
    && !BallAtArmNorth && !HoistingSouth && !BallAtTarget && !HoistingNorth
-> MoveToArmNorth
/* Rule 14 */ !MovingSouth && DoorOpen && !InSouth && !MovingToArmSouth
    && !BallAtArmSouth && !MovingNorth && !MovingToTarget && InNorth && !MovingToArmNorth
    && !BallAtArmNorth && !HoistingSouth && BallAtTarget && !HoistingNorth
-> DONE
```

Execution of a reactive plan can be thought of as a game between system and environment, in which the system makes a move (performs an action, such as *moveToTarget*) and the environment responds (a corresponding action, such as *_arrivedAtTarget* occurs). However, only the system actively seeks to achieve its goal. The environment is indifferent and may or may not respond as expected. When the environment does not respond as expected, execution of the reactive plan will involve jumping between paths in the domain model towards the goal as the system continually makes a move towards its goal and the environment forcing the system to take alternative paths.

The two paths to the goal can perhaps most easily be displayed in an LTS representation of the plan. Figure 3 shows the plan LTS with the alternative paths highlighted. Here, the highlighted paths originate from states 0 and 1 respectively. Recall from Figure 2 that the only difference between these states is that *DoorOpen* is true in state 0 and false in state 1. In other words, the plan involves moving through the door when it is open and hoisting over the wall when the door is closed. As is explained in Section 5, these alternative options correspond to two different modes of component configuration in the system.

To summarise, the abstract domain model expounded above contains properties and actions in terms of the "problem domain" and makes few assumptions about the infrastructure available, other than the existence of at least one mobile agent and a robot arm with fixed location. The set of available concrete actions

(and constraints on their application) are provided in a concrete domain model. The composition and generation of abstract and concrete domain models are the same. The difference lies in the set of actions and the scope. A concrete domain model includes actions specific to the available infrastructure and may have a more narrow scope that focusses on a particular part of the overall environment, such as the functioning of the arm. The decomposition of an abstract action essentially amounts to a small planning step in the concrete domain to get from the precondition of the abstract action to the postcondition. More details of this are described in [8].

In our scenario, the infrastructure consists of two Koalas and the arm. The actions available to these agents include the following:

```
set Sys = {k1.startLoading, k1.startUnloading, k1.startGoToTarget, k2.startLoading,
       k2.startUnloading, k2.startGoToTarget, arm.startPickUp, arm.startPutDown, ...
```

As is described in Section 5, the action *startMoveToTarget*, for example, can be decomposed by the actions *startLoading*, *startGoToTarget*, *startUnloading* for a single Koala.

4.3 Overhead

To give a rough indication of the computation involved in our current implementation of domain modelling and reactive-plan generation, the LTS for the abstract domain model in this case study consists of 25 states and 76 transitions and is generated by LTSA in under 1ms. Plan generation, including some basic preprocessing steps, takes around 50ms. However, since we harness planning-as-model-checking technology, these numbers can be expected to grow exponentially with the size of domain model. This is why planning is positioned in the goal management layer of our conceptual architecture and employed for top-level adaptation only, i.e., when system goals change, or the environment changes significantly enough for the domain model to require updating. In this case study, we focus on adaptation in the change management layer, as detailed in the next section.

5 Generating Software Configurations

In order to put the generated plan into action, the system must derive the software architecture which is able to perform the plan and modify this architecture when it becomes necessary to ensure reliable execution. Each of the robots has a set of software components associated with it which implement and support various behaviours appropriate to each platform. The computational capacity of each robot is limited and so the purpose of the change management layer becomes, in addition to plan execution, that of deciding which components will be instantiated into a configuration.

The selection of components is based primarily on a mapping from the system actions required by the plan to component interfaces. In addition to these

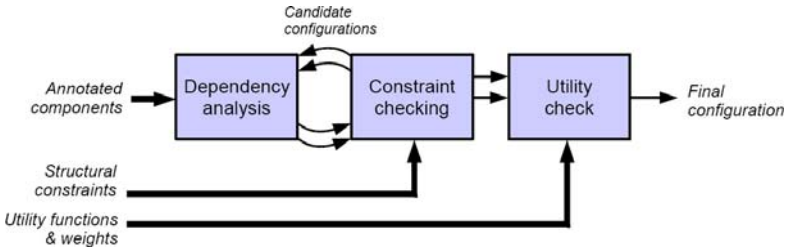


Fig. 4. Configuration generation

purely functional concerns, the system can generate configurations which satisfy arbitrary structural constraints and make choices between alternatives on the basis of non-functional properties of interest to the user, such as reliability or performance. Hence, there are three major steps to the process as shown in Figure 4, which also exhibits the information provided by the user. The first two steps, dependency analysis and constraint checking, are interrelated. Candidate configurations generated by the dependency analysis are checked against the structural constraints, and may be passed back after modification to consider further dependencies. Configurations which satisfy both initial steps are then evaluated according to their non-functional properties to select a final configuration. A configuration is constructed by instantiating components and connecting required ports to the provided ports of another component where the interface type matches. A *complete* configuration contains no component with an unsatisfied requirement.

5.1 Dependency Analysis

In the first step of the algorithm, dependencies between requirements and provisions are used to generate complete candidate configurations which provide sufficient functionality to execute the given plan.

In our scenario, at least three configurations are required: one which enables the Koala to locate and pass through the door, and if that is not possible, one which enables the arm to lift the ball over the wall, plus one which enables the Koalas to transport the ball. Each of these configurations is implied by the actions present in the plan.

For example, the presence of a $startGoToX$ action in the plan indicates that the configuration must include a component which provides a suitable implementation of this action. A particular interface type is associated with each type of action, and thus the system selects components which implement the relevant interfaces. More formally, the set of desired interfaces is derived from the system actions in the plan, $Sys_{Plan} = \text{range}(Plan) \setminus Env_{Dep}$, by way of the binary relation $Implements \subseteq Sys \times Interfaces$. The desired set is the image of this relation under the set of plan actions, $Implements[Sys_{Plan}]$.

Given the set of components required for their functionality, the system can then construct a complete configuration by considering the required interfaces of

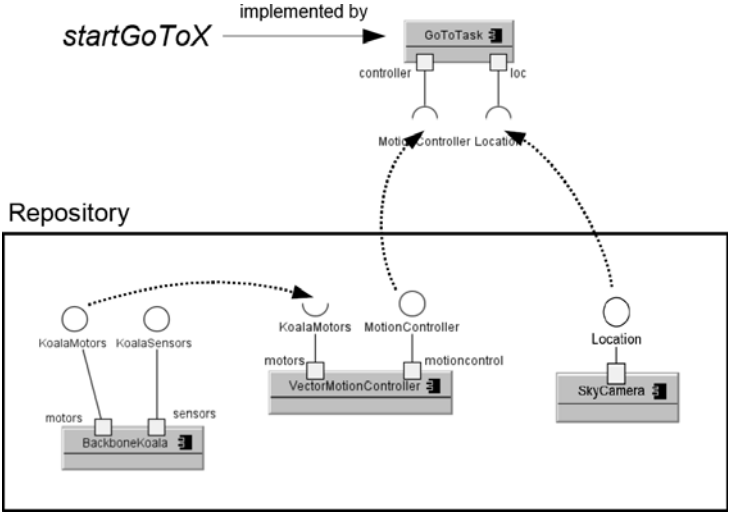


Fig. 5. Components are selected according to the plan actions

those components. Providers of these must also be instantiated and connected to the relevant ports of the action component. The full details of the dependency analysis have been given elsewhere [9].

Figure 5 shows how a configuration would be generated to perform a *startGoToX* action. The system is aware of an interface provided by the GoToTask component that implements this action. Then, to complete the configuration, it considers the requirements of the GoToTask, and their requirements in turn. In this case, the GoToTask requires a MotionController implementation and a LocationServer. The latter is provided by the SkyCamera which connects to external infrastructure. The VectorMotionController has a further requirement of KoalaMotors which are provided by the Koala component, representing the hardware capabilities of the robot. Thus, the configuration for *startGoToX* is c_g ={GoToTask, SkyCamera, VectorMotionController, Koala}.

The other required configurations are generated similarly. For the arm, the required configuration is c_a ={BallGrabber, BallPlacer, KatanaArm, Webcam} where the first two components implement *startPickUp* and *startPutDown* actions. For the door search, there are two possibilities representing different strategies for achieving the behaviour. These are c_{s2} ={VisualDoorLocator, Webcam, VectorMotionController, Koala} and c_{s1} ={IRDoorLocator, VectorMotionController, Koala}. The first component in each implements the behaviour in a different way. VisualDoorLocator requires a Webcam since it uses image recognition to detect the door, while the IRDoorLocator attempts to use the infra-red sensors provided by the Koala to do the same. We assume that in general there are multiple implementations of each interface, and so there are several candidate configurations which provide the same functionality.

5.2 Structural Constraints

Each of the candidates produced by the dependency analysis is checked against any structural constraints that the user has provided. Any candidates which are not valid according to the constraints are essentially vetoed, as is the case in [23] where reconfigurations are checked using Alloy.

Constraints may express an architectural style [2] to which configurations must conform. In the current scenario, one constraint the user has placed on the system is that if the Koalas must move, they must simultaneously avoid (unknown) obstacles. This can be encoded as

$$\text{VectorMotionController} \in arch \longrightarrow \text{ObstacleAvoider} \in arch$$

which states that whenever a configuration includes VectorMotionController (a component for moving the robots), it must also include ObstacleAvoider.

This constraint is an example which could *never* be satisfied by relying on dependency analysis alone, since the ObstacleAvoider is not required by any other component, so there are no candidates in which it will be present. As shown above, for the *startGoToX* action the dependency analysis will produce a solution, c_g, which contains {GoToTask, VectorMotionController, SkyCamera, Koala} but not the ObstacleAvoider. Since the generation of such candidates is entirely ignorant of what might be added to achieve satisfaction, it is necessary to add extra components which are not already included in a given candidate. Components cannot be removed from candidates since this would make them incomplete. Hence, from every candidate vetoed by the constraint check, several new candidates are generated, consisting of the original plus a number of extra components. These candidates are returned to the dependency analysis for "completion", which entails ensuring all the required interfaces of the new components are satisfied, and finally resubmitted to the constraint check.

The configurations derived for the Koala by the dependency analysis do not satisfy the above constraint. Hence, new candidates are generated, completed, and re-checked. For the initial candidate c_g ={GoToTask, VectorMotionController, SkyCamera, Koala}, variants such as $c_g \cup$ {IRDoorLocator} and $c_g \cup$ {ObstacleAvoider} are generated. The completion of the latter does not introduce further components (since everything required by the ObstacleAvoider is already present) and satisfies the constraint, giving c'_g ={GoToTask, VectorMotionController, SkyCamera, Koala, ObstacleAvoider}. The configuration for the arm, c_a, is unchanged, while the ObstacleAvoider is added to the alternatives for the door search giving $c'_{s1} = c_{s1} \cup$ {ObstacleAvoider} and $c'_{s2} = c_{s2} \cup$ {ObstacleAvoider}. These configurations are shown in Figure 6.

5.3 Non-functional Properties

Once the set of complete, valid configurations has been generated, a choice must be made between them on the basis of their non-functional properties, such as reliability or performance. Each component can be annotated with a set of pairs indicating the name of a property, and the value of that property for the component, e.g., ("power-drain", "500mA") and ("reliability", "70%").

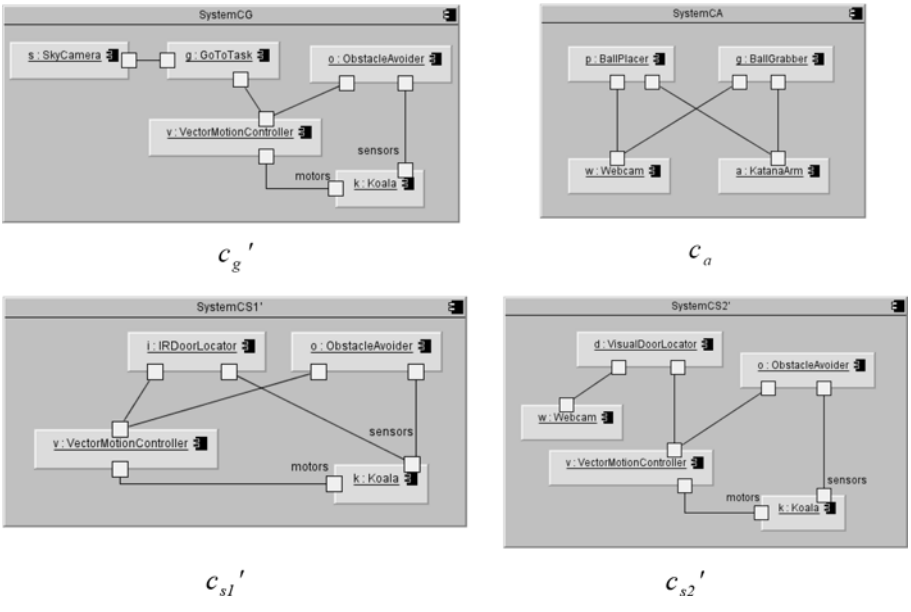


Fig. 6. Candidate component configurations

In addition to these annotations, the user provides utility functions u_{prop}, one for each property, which give the utility of every value in the range of a property. The utility is a real number between 0 and 1 where 1 represents the most useful value. For example, the utility function for power drain may be defined as

$$u_{power}(0mA) = 1 \qquad u_{power}(1000mA) = 0.05$$

The total utility of a component can be calculated from the property values by taking a weighted sum, resulting in a value between 0 and 1, placing functionally equivalent components in a partial order. In other words, the utility $U(c)$ of a component c is

$$U(c) = \sum_{p \in NFProp} w_p \times u_p(c.p)$$

where $NFProp$ is the set of all non-functional properties, $c.p$ indicates the value of p for component c (assumed to be 1 where no annotation is provided), and w_p is the weight for property p. The weights for each property are provided by the user, and represent the user's priorities or preferences among the non-functional properties. Since power drain is particularly important in the current context, w_{power} will be a high value. These weights must sum to 1.

The utility of a configuration is then defined as the average utility amongst the components it contains. The configuration with the maximum utility among the candidates is selected as the final choice. If two configurations have the same utility, the smaller one (in terms of components) is picked.

This method for calculating the utility of a configuration masks a significant assumption which we aim to eliminate in future work. The approach supposes that the component annotations are correct whether the component works in isolation or in a large configuration. It is trivial to conceive of a situation where this is not the case, such as a configuration which involves large numbers of components which claim to be fast, which will no doubt exhibit poor performance.

Nevertheless, the present approach allows us to differentiate between the two solutions for locating the door: c'_{s2}, which uses the Webcam (using a lot of power), and c'_{s1}, which uses the infra-red sensors (which are unreliable). Hence, the properties of interest are the power drain and reliability.

In c'_{s2}, Webcam is annotated with ("power-drain", "500ma") and VisualDoorLocator is annotated with ("reliability", "high"). In c'_{s1}, IRDoorLocator is annotated with ("reliability", "low"). There is no annotation for power drain, and so this is assumed to be the best case (its utility is 1). The utility of the two configurations can then be computed by providing weights for reliability and power drain. In this case the user is concerned that a high power drain will prevent the ball from being delivered. Hence, $w_{power} = 0.8$ and $w_{reliability} = 0.2$. The computed utility values are given below.

$$U(\text{Webcam}) = 0.8 \times u_{power}(800mA) + 0.2 \times 1 = 0.36$$

$$U(\text{VisualDoorLocator}) = 0.8 \times 1 + 0.2 \times u_{reliability}(high) = 0.98$$

$$U(c'_{s2}) = \frac{U(\text{Webcam}) + U(\text{VisualDoorLocator}) + 3}{5} = 0.868$$

$$U(\text{IRDoorLocator}) = 0.8 \times 1 + 0.2 \times u_{reliability}(low) = 0.82$$

$$U(c'_{s1}) = \frac{U(\text{IRDoorLocator}) + 3}{4} = 0.995$$

Here, c'_{s1} has the higher utility and is selected. This means that the final choice of configurations is

$$c'_{s1} = \{\text{IRDoorLocator}, \text{ObstacleAvoider}, \text{VectorMotionController}, \text{Koala}\}$$

for the first mode of operation (applicable when the door is open), and

$$c_a = \{\text{BallGrabber}, \text{BallPlacer}, \text{Webcam}, \text{KatanaArm}\}$$

$$c'_g = \{\text{GoToTask}, \text{VectorMotionController}, \text{ObstacleAvoider}, \text{Koala}\}$$

for the second mode of operation (applicable when the door is closed).

5.4 Overhead

The whole configuration generation process takes between 200 and 6000ms (about 3100ms on average), while a fully scripted configuration (where the result is written directly by the programmer) takes less than 16ms to be instantiated. This compares favourably against the performance of Arshad's Planit [19] where planning for configuration takes at best 4.92 seconds, and up to hundreds of seconds if plan execution time is included. Using planning to generate configurations as well as behaviour would be significantly more expensive.

6 Reconfiguration

Having seen how the plan and initial configurations are generated, we now consider the behaviour of the system as it is running, and how it responds to problems in the environment. Once the ball has been collected, one Koala proceeds to search for the door using the IRDoorLocator. Suppose that the annotation marking this component as unreliable proves to be true, due to noise in the infrared sensors. The Koala fails to find the door after scanning the entire area of the wall and so the IRDoorLocator reports this failure by throwing an exception. This triggers a reconfiguration to find a replacement for the failed component. As shown above, there is such an alternative, c'_{s2}, which uses the more reliable VisualDoorLocator. Although the system is not aware of the meaning of the "reliability" property, it switches to the more reliable alternative by simply excluding the failed components, disregarding the property weights provided by the user. Reconfiguration happens in parallel with normal execution such that the change is almost imperceptible to an observer.

When the door is found to be closed the system must use another strategy to achieve the goal. In this case the alternative is to use the Katana arm. This requires the configuration used on the Koala to switch to c'_g, and c_a to be instantiated on the arm.

7 Conclusions

This case study demonstrates how our three-layer abstraction—previously applied in the field of robotics—can be successfully applied to self-adaptive software systems, including their self-assembly. In our case, the model separates the concerns of such systems into high-level task planning and replanning, architectural configuration and reconfiguration, and component-based control. Although we have not yet fully elaborated the replanning feature, which involves updating the domain model at runtime to reflect arbitrary changes in the environment, we believe the advantages of the general approach are apparent. In particular, each layer of the model operates at a level of abstraction appropriate to it, simultaneously simplifying the problem and generalising the solutions, such that several opportunities for adaptation are uncovered. The arrangement of the layers aims to ensure that the most costly adaptations are peformed the least frequently. Of course, the general mechanisms of the approach are not tied to the particular implementation outlined here.

Adaptability is provided by four mechanisms: (i) control loops within domain components, (ii) architectural reconfiguration, (iii) the nature of reactive plans, and (iv) dynamic replanning. Replanning will likely necessitate modification of the domain description with information gathered at runtime, which in the minimal case is that a certain action fails (either because of changes in the environment or a software fault) in some contexts. The planner can then be invoked to generate a new plan which avoids the failing action. Replanning will also be required when the system's goals change.

Interesting future work includes further extension of ideas in [8] to address scalability of the goal management layer by generating plans from sub-goals and verifying that these sub-goals are consistent with the overall goal. Also, the approach as currently implemented relies on a central plan interpreter in the change management layer to co-ordinate component execution on several hosts, with only indirect communication between each host. Aside from being a single point of failure, this scheme assumes the entire world state is (correctly) observable on this host. This can be made more scalable and robust by providing a means to distribute plan interpretation and configuration generation. Naturally such concurrent plan execution leads to synchronisation and co-ordination issues which must be addressed [24,25].

Finally, we also propose enriching the domain model to capture uncertainty and partial knowledge about the environment, possibly by exploiting existing work on modal transition systems [26], which allow "maybe" transitions and facilitate the incorporation of new domain knowledge by model-merging.

Acknowledgements. The work reported in this paper was funded by the Systems Engineering for Autonomous Systems (SEAS) Defence Technology Centre established by the UK Ministry of Defence.

References

1. Garlan, D., Schmerl, B.: Model-Based Adaptation for Self-Healing Systems. In: 1st Workshop on Self-Healing Systems (2002)
2. Garlan, D., Allen, R., Ockerbloom, J.: Exploiting Style in Architectural Design Environments. In: Preneel, B. (ed.) FSE 1994. LNCS, vol. 1008. Springer, Heidelberg (1995)
3. Georgiadis, I., Magee, J., Kramer, J.: Self-Organising Software Architectures for Distributed Systems. In: 1st Workshop on Self-Managed Systems (2004)
4. Georgas, J.C., Taylor, R.N.: Towards a Knowledge-Based Approach to Architectural Adaptation Management. In: 1st Workshop on Self-Managed Systems (2004)
5. Arshad, N., Heimbigner, D., Wolf, A.: A Planning Based Approach to Failure Recovery in Distributed Systems. In: 1st Workshop on Self-Managed Systems (2004)
6. Gat, E.: Three-Layer Architectures. In: Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems (1998)
7. Ghallib, M., Nau, D., Traverso, P.: Automated Planning: Theory and Practice. Morgan Kaufmann, San Francisco (2005)
8. Sykes, D., Heaven, W., Magee, J., Kramer, J.: Plan-Directed Architectural Change For Autonomous Systems. In: SAVCBS (2007)
9. Sykes, D., Heaven, W., Magee, J., Kramer, J.: From Goals to Components: A Combined Approach to Self-Management. In: SEAMS (2008)
10. Magee, J., Kramer, J.: Concurrency: State Models & Java Programming. Wiley, Chichester (2000)
11. McVeigh, A., Kramer, J., Magee, J.: Using Resemblance to Support Component Reuse and Evolution. In: SAVCBS (2006)
12. Brooks, R.: A Robust Layered Control System for a Mobile Robot. Robotics and Automation 2(1), 14–23 (1986)

13. Cheng, S.W., Garlan, D., Schmerl, B., Sousa, J., Spitnagel, B., Steenkiste, P.: Using Architectural Style as a Basis for System Self-Repair. In: 3rd Working IEEE/IFIP Conference on Software Architecture (2002)
14. Cheng, S., Garlan, D., Schmerl, B.: Architecture-Based Self-Adaptation in the Presence of Multiple Objectives. In: SEAMS (2006)
15. Cheng, S., Huang, A., Garlan, D., Schmerl, B., Steenkiste, P.: An Architecture for Coordinating Multiple Self-Management Systems. In: 4th Working IEEE/IFIP Conference on Software Architecture, pp. 243–252 (2004)
16. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: Towards Architecture-Based Self-Healing Systems. In: 1st Workshop on Self-Healing Systems (2002)
17. Robbins, J.E., Hilbert, D.M., Redmiles, D.F.: Using Critics to Analyze Evolving Architectures. ISAW-2 / Viewpoints (1996)
18. Garlan, D., Poladian, V., Schmerl, B., Sousa, J.P.: Task-Based Self-Adaptation. In: 1st Workshop on Self-Managed Systems (2004)
19. Arshad, N., Heimbigner, D., Wolf, A.: Deployment and Dynamic Reconfiguration Planning for Distributed Software Systems. Software Quality Journal 15(3) (2007)
20. Lynch, N., Tuttle, M.: An Introduction to Input/Output Automata. CWI-Quarterly 2(3) (1989)
21. Giannakopoulou, D., Magee, J.: Fluent Model Checking for Event-Based Systems. In: ESEC / FSE (2003)
22. Giunchiglia, F., Traverso, P.: Planning as Model Checking. In: European Conference on Planning (1999)
23. Warren, I., Sun, J., Krishnamohan, S., Weerasinghe, T.: An Automated Formal Approach to Managing Dynamic Reconfiguration. In: ASE (2006)
24. Lomuscio, A., Sergot, M.: Deontic Interpreted Systems. Studia Logica (Special Issue on The Dynamics of Knowledge) 75 (2003)
25. Inverardi, P., Mostarda, L., Tivoli, M., Autili, M.: Synthesis of Correct and Distributed Adaptors for Component-Based Systems: an Automatic Approach. In: ASE, pp. 405–409 (2005)
26. Sibay, G., Uchitel, S., Braberman, V.: Existential Live Sequence Charts Revisited. In: ICSE (2008)

Model-Centric, Context-Aware Software Adaptation

Oscar Nierstrasz, Marcus Denker, and Lukas Renggli

Software Composition Group, University of Bern, Switzerland
http://scg.unibe.ch

Abstract. Software must be constantly adapted to changing requirements. The time scale, abstraction level and granularity of adaptations may vary from short-term, fine-grained adaptation to long-term, coarse-grained evolution. Fine-grained, dynamic and context-dependent adaptations can be particularly difficult to realize in long-lived, large-scale software systems. We argue that, in order to effectively and efficiently deploy such changes, adaptive applications must be built on an infrastructure that is not just model-driven, but is both *model-centric* and *context-aware*. Specifically, this means that high-level, causally-connected models of the application *and* the software infrastructure itself should be available at run-time, and that changes may need to be scoped to the run-time execution context.

We first review the dimensions of software adaptation and evolution, and then we show how model-centric design can address the adaptation needs of a variety of applications that span these dimensions. We demonstrate through concrete examples how model-centric and context-aware designs work at the level of application interface, programming language and runtime. We then propose a research agenda for a model-centric development environment that supports dynamic software adaptation and evolution.

1 Introduction

It is well-known that real software systems must change to maintain their value [26]. It is therefore curious to observe that the technology we use to develop software systems tends to hinder and inhibit change rather than to enable and support it [31]. Statically typed languages, for example, are based on the assumption that first-class values have fixed types that will not change, especially at run-time. Few mechanisms are available to developers to deal with the fact that interfaces do change over time, and real software systems may need to cope with different versions of the same libraries, possibly depending on the run-time context. Design patterns offer further evidence of ungainly workarounds that developers need to regain flexibility at run-time, for example to change the apparent behaviour of objects as a consequence of a change in state [17].

Long-lived, software intensive systems [50] cannot always be modified in a static way. Furthermore, although certain kinds of anticipated adaptations can be

B.H.C. Cheng et al. (Eds.): Self-Adaptive Systems, LNCS 5525, pp. 128–145, 2009.

built in by design as run-time configuration parameters, there are many kinds of dynamic adaptation that cannot be anticipated so easily. One canonical example of such an adaptation is run-time instrumentation: certain kinds of anomalies only manifest themselves with deployed software systems. As it is not possible to anticipate for all cases what and where to trace to observe the problematic behaviour, it may be necessary to dynamically adapt the running system. Other examples exist (such as adding new features to an always-running system), but the key characteristics remain the same — the software may need to be adapted dynamically, in a fine-grained way, while taking care not to disturb existing behaviour.

There are many important dimensions of software change. Let us just consider three of these that pose challenges for software development:

Timescale — Software is changed not only at the coarse scale of versions and releases, but also at a medium scale (*e.g.*, start-up configuration) and at a fine scale (run-time adaptation and instrumentation). Particularly at the dynamic end, little support is available to developers aside from certain design patterns and relatively low-level reflective mechanisms.

Granularity — Here too we see that software is changed not only at the coarse granularity of subsystems and packages, or the medium granularity of classes and methods, but also at a finer granularity within methods and procedures. Fined-grained, run-time adaptation of software must typically be anticipated by design, and necessitates the use of boilerplate code (*e.g.*, case-based reasoning over anticipated scenarios) or design patterns (*e.g.*, State or Strategy patterns). Unanticipated run-time adaptation will typically entail low-level techniques such as bytecode transformation.

Scope — Changes may be globally visible, they may be localized to individual users, or they may depend on an even finer context. The same software entities may need to behave differently as the run-time context changes. Mobile applications, for example, may need to switch to a fall-back behaviour as services become unavailable. Run-time instrumentation of software entities, as another example, may need to be dynamically adapted if the same entities are used by the instrumentation layer itself (*i.e.*, to avoid endless instrumentation loops) [14].

Although model-driven and round-trip engineering techniques have proved to be effective in maintaining the connection between high-level and low-level views of software systems, they do not especially address the problem of dynamic adaptation. We argue that it is necessary to go a step further from model-driven towards *model-centric* software, in which high-level, causally connected views of software and their application domain are available at run-time. In this paper we show several examples of run-time adaptation at the level of source code, so the "high-level models" appropriate to these applications take the form of ASTs that reflect the structure of software to be adapted.

Furthermore, such systems must be *context-aware* in order to control the scope of adaptations and changes. In our examples we show how context can

play an important role in software adaptation to control the scope of change. We argue that current programming technology offers only very weak support for developing context-aware applications, and that new research is urgently needed into novel *context-oriented programming* mechanisms [21].

In this paper we make our case for model-centric, context-aware software adaptation by presenting two examples of platforms that adopt this approach. We show how the presence of sufficiently high-level models at run-time can enable very dynamic forms of context-dependent software adaptation.

In Section 2 we present *Reflectivity*, a relatively mature platform for dynamic, model-centric software adaptation. We have used Reflectivity extensively in various projects to support different forms of adaptation, such as run-time instrumentation, dynamic aspects, and software transactional memory. Next, in Section 3, we present ongoing work on *Diesel*, a lightweight language workbench which can be used to adapt the programming environment to support the expression of high-level application concepts by introducing numerous, lightweight domain-specific languages. We discuss further applications of these ideas and our vision for a research agenda in Section 4 and provide an overview of related work in Section 5. We conclude in Section 6 with some remarks on future work.

2 Reflectivity — A Platform for Model-Centric Software Adaptation

In this section we present Reflectivity, a platform that supports dynamic adaptation of software by means of causally connected, high-level models of the source code [9]. The purpose of this section is (i) to motivate the need for dynamic software adaptation for various applications such as runtime instrumentation, dynamic aspects, and software transactional memory, (ii) to motivate the need for better mechanisms to support context-dependent adaptation, and (iii) to demonstrate that sufficiently high-level models available at run-time (in this case ASTs causally connected to bytecode) facilitate run-time adaptation.

Reflectivity is built on top of Smalltalk, since it already provides extensive support for run-time reflection, albeit at a relatively low-level of abstraction [9]. Furthermore, Smalltalk provides full access to the implementation of its infrastructure, making it ideal for extensive experimentation. Any other language that supports run-time structural and behavioural reflection and access to the infrastructure would also be suitable.

2.1 A Model for Dynamic Software Adaptation

The particular challenge we are focusing on is support for dynamic, fine-grained and possibly context-dependent software adaption. Let us consider the canonical example of run-time instrumentation:

- We may need to install the instrumentation code *dynamically* in the running system because the phenomena we wish to study only occur in the deployed system (say, a web service).
- The adaptation is *fine-grained* because we wish to monitor only part of a given method (say, conditional access to an authorization service).
- The adaptation is *context-dependent* because we are only interested in monitoring calls made from a specific application, not others.

Other plausible scenarios, such as adding features to a running system, would serve as well for establishing our requirements.

In order to dynamically adapt software, we need a *model* to reason about it. In our run-time instrumentation scenario the following properties are important:

Abstraction Level: This model should be high-level, reflecting the language concepts we wish to instrument, rather than, say, the generated bytecode.

Completeness: The model should represent the complete software, from coarse-grained structures like classes, methods down to sub-method structures such as variable accesses and method calls.

Although these properties may seem obvious, in most cases the representations used for software adaptation today do not satisfy them. The representation used is often plain (source) text. Modern development environments do better: here the code is represented with dedicated data-structures that better support code presentation (*e.g.* pretty printing) or code change (*e.g.* refactoring). But these data structures are those of the development environment, not of the language itself. They are not available to support run-time adaptation.

Runtime representations are often tailored solely towards execution, such as bytecode representations for Java or Smalltalk. Representations based on bytecode are low-level, and therefore suffer from a semantic mismatch with the core language concepts.

The reflective representation of the structure of software available in many modern object-oriented languages provides a high-level model for packages, classes and methods, but it lacks any representation of sub-method structure.

As we are especially interested in adaptation *at runtime* and *by the system itself*, we conclude that the model needs to have the following properties:

Self Representation: The model of the software needs to be available from within the running system itself.

Causal Connection: When we change this model (either from the outside or from within the system), the behavior of the program needs to change. Conversely, when the system changes, the representation needs to change, too. The program needs to stay in sync with the model at all times.

Meta-annotations: We need to be able to extend the representation to use it in many contexts and annotate it with meta-data. For example, different tools that deal with the structure of the system need slightly different information.

> Annotations allow the programmer to associate meta-data with any node, making the existing AST-based representation extensible.

Models that have the properties of *self representation* and *causal connection* are called *reflective*. There is a long history of reflection in programming languages in general and in object-oriented languages in particular [45]. *Sub-method reflection* [10,9] provides a model of the software that exhibits all the properties discussed above. Software is represented down to the statement level by a causally connected, annotated *Abstract Syntax Tree* (AST). Changes to the AST will be (lazily) propagated to the generated bytecode using runtime just-in-time compilation. Semantics are given to annotations by an open compiler infrastructure: annotations are interpreted by dedicated compiler plug-ins.

We will now illustrate this approach by a series of examples.

2.2 Run-Time Instrumentation

With a causally connected model of the whole software, we can provide a framework to support *instrumentation at runtime*. The model chosen is that of *partial behavioral reflection* [47,12,40].

The central notion is the *Link*. The link is set as an annotation to one or more nodes of our AST. The link points to a meta-object and can be parameterized to indicate which information is passed to the meta-object. In addition, we can set a condition to specify when a link is active and to specify if the meta-object is called before, after or instead of the original instruction. Figure 1 shows the interaction of the AST, the link and the meta-object.

Links are specified as annotations on the AST. A compiler plugin transforms the AST before execution to take the links into account. A link thus results in code to be inserted in the program at the nodes where it is installed.

This model provides some interesting characteristics: it is completely dynamic due to just-in-time compilation at runtime. We can create links at runtime, configure them and install them in the system. Links can even be installed by other links, or they can remove or install themselves.

A side-effect of using the higher-level representation provided by the AST is improved performance. It is actually easier to generate more efficient code using the AST [10]. Furthermore, we only have to generate bytecode for those parts of the system that take part in the execution. The actual set of classes

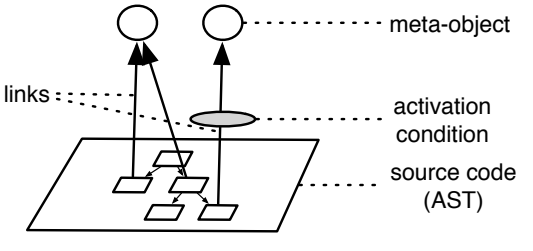


Fig. 1. The link-meta-object model

used by an application is generally smaller than the overall code base by an order of magnitude. As a result, dynamic code generation provides an additional performance benefit [9].

2.3 Localization: Annotating Structure

We have seen both a structural model of our system, and a framework for dynamic behavioral reflection based on annotating the structural model with *links*. Now we will see how to manipulate this structure using behavioural reflection.

To make this possible, we need to be able to reference the structural model from the behavioral world. The simplest way to do this is to allow the nodes of the structural model to be meta-objects, as shown in Figure 2.

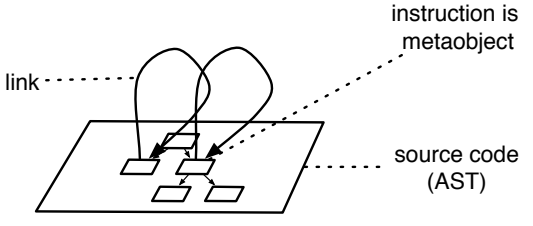


Fig. 2. Bridging structural and behavioral model

This allows a node to be annotated before it is executed. This way one can easily realize tracing or feature analysis [13]. For example, a simple code-coverage tool can be realized by installing a link on each AST node of interest. We just provide a method `markExecuted` to mark AST nodes as having been executed. The links are activated when the AST nodes are executed, and simply invoke this method to record the fact.

```
link := GPLink new metaObject: #node;
            selector: #markExecuted.
```

Listing 1.1. Code-coverage analyzer realized with Reflectivity

When we install the link on the node representing methods, we obtain method level coverage. But with our sub-method model, we can go a level deeper and even install the link on all assignments.

To improve performance, it is even possible to remove the tagging-link at runtime just after tagging the node. In this way, the method would, at the next execution, be recompiled to only call `markExecuted` on those nodes that have not yet been executed before. We can also take advantage of activation conditions, for example, to only tag nodes that are executed in the context of a unit-test.

The possibility of both installing and removing links at runtime allows for just-in-time annotation: links are installed on-demand on all methods that are to be executed next. This way an annotation can spread itself through the system, driven by the flow of execution itself. Examples like these make Reflectivity especially suitable for building self-monitoring and self-evolving systems.

2.4 Scoping the Effect of Changes

An interesting problem arises when instrumenting basic system classes. The instrumented code itself is used at runtime by the meta-object, leading to endless loops. This makes any use of reflection on basic system classes like *Number* or *Array* impractical. This problem can be seen with all reflective systems — a well-known example is CLOS [6].

To solve this problem, we provide the possibility to scope the activation of links towards meta-level execution [14]. Links are parameterized with the level for which they are activated. This way we can restrict the introduced change towards, for example, base-level program execution. Note that the same mechanisms can be used to reason about execution of meta-level programs. For example, it may be the case that a profiler realized as a meta-object needs to be analyzed to improve performance. By restricting the link that activated the profiler to the meta-level, we can use the profiler on itself without the danger of endless loops.

2.5 Implementing Higher-Level Dynamic Language Features

With partial behavioral reflection, we can easily adapt the programming language to support new language features. Very deep changes can be realized that normally require changes at the level of language implementation (*i.e.* the virtual machine).

Dynamic Aspects. Partial behavioral reflection can serve as an efficient technique for implementing *Aspect-Oriented Programming* [39]. We have used the dynamic features of Reflectivity to implement *dynamic aspects*, which are not woven into the code at compile time, but instead can be introduced and retracted at runtime. Compared with traditional runtime AOP implementation techniques, we can see some improvements. We can generate better code than typical bytecode transformation based approaches as we can leverage the higher-level AST representation [9]. We can leverage the link-conditions to efficiently control aspect activation at runtime.

Transactional Memory. We have realized software transactional memory for a dynamic language [37]. It is notable that this realization was done without any changes to the underlying virtual machine. With the help of Reflectivity we introduced a transactional execution context and were able to reify all state access to be handled by the transactional model implemented in the host language.

3 Diesel — An Engine for Bringing Models Closer to Code

We have seen that it is important to have higher level models of software available at run-time in order to enable dynamic software adaptations. But what about

the application logic itself? Models of the software structures can be far removed from the application domain. What we need to enable dynamic adaptation of application logic is to make application models more explicit in the code. One way of doing this is to raise the level of programming by means of specialized domain specific languages (DSLs).

This raises new, extrinsic problems, as a new DSL will not be able to automatically benefit from existing development tools available for the general-purpose host language. We need to be able to adapt the host language and environment to support new DSLs. As before, the adaptions must be *fine grained* because DSL code may be interspersed with regular source code, *dynamic* because we want to be able to change host compilers and tools on the fly, and *context-dependent* because DSL code may be restricted to certain parts of an application.

Diesel is a lightweight language workbench that closely integrates with the host language [16]. This enables developers to incrementally bend the syntax and semantics of the host language to suit their exact needs for a particular problem domain. As such, Diesel is an environment for developing domain specific languages (DSLs), with the aim of giving application developers more suitable abstractions than the host language provides. Contrary to other approaches, Diesel reuses the traditional compiler toolchain and closely integrates with the existing tools of the programming environment, such as editors, debuggers, inspectors, *etc.* A close integration is crucial to keep the abstraction gained through new language features. While language developers might want to toggle between a view on the original source and the transformed result, domain developers would like to stay at the abstraction level of their code at all times.

The host language of our implementation is Smalltalk, which provides us with a uniform development environment. The abstract code representation of the compiler is reused in all parts of the system and can thus take advantage of our extensions. For example, if we change the syntax in a specified part of the system, syntax highlighting and debugger continue to work. Application developers do not have to learn new tools, but continue to use the existing ones even if they mix multiple languages.

3.1 Example: Modelling Relationships

A common challenge in transforming UML models to code is how to implement relationships between objects. The problem has long been solved in relational databases [2,32], but none of today's mainstream languages provide first-class relationships [29]. If done by hand, it is easy to introduce subtle bugs that might be very hard to detect. With code generation, huge chunks of boilerplate code may appear that are hard to understand and impossible to change. In either case, debugging and maintaining the code is cumbersome, because developers not only have to think in terms of the high-level DSL code that they specify, but also in terms of the code that is generated.

We present an approach to solve this problem with Diesel. The example only handles 1:1 relationships, however it could easily be extended to support arbitrary n:m relationships.

To implement the write accessor `next:` of a double linked list, a developer or a code generator would write something like this:

```
1 Link>>next: aLink
2    next isNil ifFalse: [ next instVarNamed: 'prev' put: nil ].
3    next := aLink.
4    next isNil ifFalse: [ next instVarNamed: 'prev' put: self ]
```

At run-time lines 2 and 4 are need to ensure that the inverse relations are properly updated. The actual assignment only occurs on line 3. Most parts of the code are not interesting to developers. It is an unnecessarily complex code fragment specifying an implementation detail of 1:1 relationships that is probably used at several places throughout the application.

We now replace the complex write accessor from above with a plain write accessor that does not update the opposite relationships:

```
Link>>next: aLink
    next := aLink
```

However we put high-level annotations next to the instance variable declaration of the `Link` class, to tell Diesel that all write access to these variables requires their respective inverse relationships to be updated:

```
Link instanceVariables: #(
    next <opposite: prev>
    prev <opposite: next>
)
```

When compiling the method, Diesel will automatically generate the necessary boilerplate code around it. This generated code is never visible, not even in the debugger, to ensure that the developer can concentrate on the high-level model. The magic behind the transformation comes from a rule that has been added to the Diesel engine. Whenever source-code is parsed, translated and annotated these rules are processed to enable interaction with the compiler:

```
1  TreePattern
2     match: '`variable := `expression'
3     do: [ :context |
4        variable := context at: '`variable'.
5        opposite := variable annotationNamed: 'opposite'.
6        opposite notNil ifTrue: [
7           context addNodeBefore: ``(`,variable isNil
8              ifFalse: [ `,variable instVarNamed: `,opposite put: nil ]).
9           context addNodeAfter: ``(`,variable isNil
10             ifFalse: [ `,variable instVarNamed: `,opposite put: self ]) ]
```

The rule given above matches all parse tree nodes that assign an expression to an instance variable (line 2). The remaining code checks if the instance variable is annotated with an opposite annotation (lines 5–6) and then defines the transformation programmatically. This is done using a quasi-quoting mechanism [3] to build and inject the AST nodes that update the opposite relationship into the tree. Subsequently the tree is transformed to bytecodes, by the standard Smalltalk compiler. The handwritten and the transformed code result in identical bytecodes, therefore both approaches perform equally at runtime. In practice the minimal increase in compilation time due to the additional transformations can be neglected.

3.2 Scoping the Effect of Changes

In the above example the scope of the transformation is given at the level of parse tree nodes. Often such transformation rules only apply to a carefully chosen part of the system however. For example, the above transformation should be used in model code only, but not in the UI implementation. Diesel supports a wide variety of additional constraints that can be composed and added to transformation rules: packages, namespaces, classes, class hierarchies or even specific methods.

Furthermore arbitrary conditions can be added to the transformed parse-tree nodes, so that the transformation is only in effect if a certain runtime condition is met. The generated code can resort to the reflective capabilities of the system [38] and select the appropriate behaviour depending on the runtime context [7].

4 Towards a Research Agenda

Full reflection (*i.e.*, with run-time intercession) has been widely available in dynamic programming languages for many years, notably in Smalltalk [38] and CLOS [22]. Nevertheless, reflection has been commonly considered to be either too dangerous or too difficult to use for common programming tasks. Static languages such as Java and C++ offer a weaker form of reflection that only supports introspection at run-time (*i.e.*, the possibility to examine but not to affect the model elements).

There is increasing pressure to adapt software systems at run-time. If the host programming language does not offer reflective features, programmers can be forced to adopt workarounds, such as reifying and interpreting model elements within their programs. This obviously places a heavy burden on developers who must build up this infrastructure themselves, possibly in *ad hoc* ways.

The Reflectivity framework simplifies the development of reflective applications by offering ASTs as relatively high-level, causally connected, run-time models of program elements. Reflection at the sub-method level is enabled since the deep structure of programs is captured, unlike in other approaches which stop at the method level. We have used the Reflectivity framework extensively for

various purposes, including the analysis of software features [13], dynamic monitoring of software entities from the IDE [41], and even for the implementation of pluggable types, where type expressions are encoded as source code annotations and interpreted by compiler plug-ins [19].

Despite these successes, Reflectivity is focused on reifying model elements of the host programming language, not those of the application domain. (One could also say that the application domain of Reflectivity *is* the host programming language.) What we are missing is Reflectivity for domain models. We envision a system where end users can change their domain models on the fly, without having to touch the host programming language [36].

Traditionally domain concepts are translated and encoded in the source code in such a way that makes it difficult to reason about these concepts once the software is deployed. It is often difficult, for example, to find the software components responsible for a given end-user feature in the source code. In a model-driven approach, one would express domain concepts at the level of meta-models and models, and then generate code from these descriptions. In a straightforward approach, this still has the consequence that the domain models are no longer directly expressed in the code. If features must be dynamically adapted, there is no easy way to manipulate these model elements from the running system. Instead of generating code from models, we feel that it is necessary to *bring models closer to code*. This means that domain concepts should be expressed directly in the source code, rather than being encoded using concepts of the solution domain. In essence, rather than applications being model-driven, with models merely being used to generate code, we believe that they should be *model-centric*, with models being first-class entities that can be manipulated at run-time.

One way of bringing models closer to code is to provide higher-level, domain specific languages for model concepts more directly. One downside of this approach is the potential for the proliferation of DSLs, each with their own obscure syntax. DSLs should therefore be simple and lightweight. Another important downside is that the existing development environment will need to be adapted to work with each new DSL. Diesel addresses these problems by offering a lightweight framework for specifying simple DSLs that are transformed into the ASTs used by Reflectivity. The transformations are used to keep the development tools in sync with each DSL, so that editors and debuggers, for example, can present developers with the original DSL code rather than the generated host language code. In a sense, the model *is* the code.

Returning to the theme of software adaptation, we note that any adaptation manifests itself as a kind of software change, which is possibly intrusive and possibly context-dependent. As an example, consider software that should adapt its policies for ensuring changing requirements (*e.g.*, related to concurrency control, or security, or transactional behaviour) dynamically according to (i) the run-time context (*e.g.*, the presence of competing applications), and (ii) availability of related services (*e.g.*, for optimistic or pessimistic transaction support). Such an application has clearly defined requirements but can only partially anticipate its run-time context and the nature of services available to meet those requirements.

Depending on the context, the same software will need to behave differently, and may need to be adapted in unanticipated ways.

We summarize the research directions that we see as essential to support dynamic software adaptation as follows:

Bring Models Closer to Code. In order to enable dynamic software adaptation, models should be first-class, high-level artifacts available at run-time for both introspection and intercession. Structured source code and run-time annotations offer one light-weight technique to embed domain knowledge in source code [28]. Lightweight DSLs are another promising technique.

Model-Centric Development. Rather than seeking ways to embed models in source code, perhaps we should replace the source code as an artifact and directly program with models. After all, third generation languages were once seen as a way to "generate (machine) code" from high-level specifications. Nowadays we consider programs in these languages to be the source code. We need to take the next step and jettison our third generation object-oriented languages in favour of models as source code. Environments to support model-centric development would concentrate on directly manipulating first class models and their meta-models. Models would be available at run-time to support the same kinds of adaptations available to the developer at development time.

Context-Oriented Programming. COP refers to programming language mechanisms and techniques to support dynamic adaptation to context [21]. Although many present-day applications need to be context-aware, context-dependent behaviour is generally programmed in *ad hoc* ways, due to the lack of support in modern programming languages. Some COP languages have been proposed [7,18], and both Reflectivity and Diesel are examples of a frameworks that provide some degree of COP support, but research is very young, and there is no consensus how best to support COP in programming languages.

5 Related Work

There is a long history of reflective programming languages, ranging from dynamic languages such as Lisp, Smalltalk, Scheme and CLOS, to static languages like C++ and Java, which provide a more limited form of run-time introspection rather than full intercession. All of these approaches are limited to models of the code base, and do not take models of requirements, design decisions or architecture into consideration.

Over the years various approaches have attempted to keep high-level knowledge about software in sync with the software itself. The earliest examples of these is probably Literate Programming [24], in which documentation and source code are freely interspersed and maintained together.

In some cases Architectural Description Language (ADL) specifications are considered to be part of the running software system, rather than simply a higher-level description of it, as it is the case with Darwin [27]. Although ADLs

provide a high-level interface for specifying and configuring components at an architectural level, there is not really any explicit representation of a model that is developed in tandem with the rest of the software.

Generative programming approaches [8] produce software from higher-level descriptions using such mechanisms as generic classes, templates, aspects and components. A general shortcoming of these approaches is that the transformation is uni-directional — there is no way to go from the code back to higher-level descriptions. Round-trip engineering refers to approaches in which transformations are bi-directional [1]. Models and code are still considered to be separate artifacts, so models are not available at run-time for making adaptive decisions.

Case tools and 4GLs represent an attempt to simplify the generation and adaption of an application. However the main focus of these approaches was to generate code and not to consider models as executable artifacts of software development.

Model-driven engineering (MDE) [4,42] refers to a more recent trend in which application development is driven by the development of models at various levels of abstraction. Platform-independent models are transformed to platform-specific models, and eventually to code which runs on a specific platform. Generally these transformations are performed off-line, so models are not necessarily available to the run-time system, though some approaches support this [20].

The Eclipse Modeling Framework (EMF) [5] provides facilities for manipulating models and generating Java source code from these models. Here too the focus is on models of source code, rather than on other views of a software system. The model and the code are still separate entities.

Naked objects [35] is an approach to software development in which domain objects and software entities are unified. Business logic is encapsulated in the domain objects and the user interface is completely generated from these domain objects. In this approach the domain model and the executing runtime are tightly coupled. Although naked objects address the earlier complaint against approaches which separate domain models from the source code or the running system, they do not offer any help in integrating other views of the software as it is being developed (*i.e.* requirements models, architectural views, and so on).

Aspect-Oriented Programming (AOP) [23] provides a general model for modularising cross cutting concerns. *Join points* define points in the execution of a program that trigger the execution of additional cross-cutting code called *advice*. Join points can be defined on the runtime model (*i.e.*, dependent on control flow). Although AOP works at a sub-method level, it does not provide a structural model of the system or any other reflective capabilities. The goal of AOP is to modularize crosscutting concerns, not to provide a model for dynamic software adaptation.

Reflex [47] pioneered partial behavioral reflection in the context of Java. Here links are associated with so called *hooksets*, abstractions of operations at the bytecode level. Therefore, the structural model of Reflex is that of bytecode, not the higher-level AST. In addition, Reflex does not support meta-annotations on bytecode.

Context-Oriented Programming (COP) [21] refers to programming language support for developing applications whose behaviour depends on the run-time context. Present prototypes of COP languages focus on mechanisms for adapting behaviour to context, but provide little support for reasoning about context at the model level.

Changeboxes [11] provide a mechanism to control the scope of change in a running system. Deployment and development versions of a running software system can co-exist without interfering. Mechanisms for merging differences and resolving conflicts, however, must be handled in an *ad hoc* fashion, as no fully general approach exists for all usage scenarios. Changeboxes currently operate at the level of source code changes. There is no notion of changes to higher-level models.

Unlike general-purpose programming languages, domain specific languages (DSLs) tend to be compact languages that provide appropriate notations and abstractions for a particular problem domain. It was shown that DSLs increase productivity and maintainability for specialized tasks [15]. DSLs are often categorized as being either homogenous (internal), where the host-language and the DSL are one and the same, or heterogeneous (external), where the two languages are distinct [44]. Techniques have been proposed to define language and semantics for new DSLs [25]. The idea of designing languages that embrace the addition of new DSLs has been a focus of research in the past [33,49,48]. However integrating new languages into existing tools has been largely neglected.

In the 1990s there was considerable interest in the development of architectural description languages (ADLs) [43] to capture and express architectural knowledge of a software system. ADLs can be viewed as DSLs for describing the architecture of complex software systems. Many DSLs formalize architecture in terms of components, connectors, and the rules governing their composition [43]. This idea is also implicitly contained in the notion of *scripting languages*, which can be seen as DSLs for composing applications from components written in another, usually lower-level programming language [34]. Despite this, the interplay between conventional object-oriented languages, ADLs, scripting languages and DSLs has not yet been thoroughly studied nor has it been exploited in practice.

6 Concluding Remarks

Software systems are under increasing pressure to support run-time adaptation for localization, mobile platforms, dynamic service availability, and countless other context-dependent applications. Unfortunately mainstream programming languages and development environments focus on limiting and restricting change rather than enabling it. Specifically, dynamic, fine-grained and context-dependent software adaptation is not well-supported by modern development technology.

In this paper we have presented two ongoing research projects that illustrate the principle of model-centric, context-aware software adaptation, and we have outlined a number of promising research directions for further exploration.

Reflectivity is a mature, model-centric framework for dynamic software adaptation. Software structures are represented at run-time by means of abstract

syntax trees, which enables both dynamic and fine-grained adaptation. Furthermore, by means of partial behavioural reflection, adaptations can be scoped to the dynamic context. As a typical example, run-time instrumentation can be scoped to the context of a given feature, so it is possible to identify which software components support a given feature.

Diesel is a new framework for bringing models closer to code by supporting the definition of lightweight DSLs that are transformed to the host programming language using the same high-level source code models as provided by Reflectivity. The development tools, such as editors and debuggers, are aware of the transformations, so developers can continue to work with the high-level models rather than the transformed code, even while debugging. Here too, run-time models are used to support fine-grained adaptations that can be scoped to specific parts of the application requiring these DSLs.

Reflectivity and Diesel both build on top of the same rich programming environment and therefore could profit from each other in the future. On one hand, the causal connection of model and code would help Diesel to automatically propagate language changes to all its users. Furthermore Diesel could apply language transformations that normally happen at compile-time from the dynamic world. On the other hand, Reflectivity could profit from a richer language infrastructure and the compiler plugins could reuse the Diesel transformations.

These are only two examples of promising research directions to support dynamic change and software adaptation. We argue that more research is needed to close the gap between models and code. In general, software systems need to be *change-enabled* — instead of limiting and restricting change, they should actively enable change by treating change as a first-class entity [30,31]. Ultimately we want model-centric and context-oriented programming environments where we can directly manipulate models both during development time and run-time, and software can be dynamically adapted by context.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 - Sept. 2010).

We thank Tudor Gîrba and Jorge Ressia for their careful reviews of various drafts. We also thank the anonymous reviewers for their numerous suggestions on how to improve the presentation of this paper.

References

1. Antkiewicz, M., Czarnecki, K.: Framework-specific modeling languages with round-trip engineering. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 692–706. Springer, Heidelberg (2006)
2. Atkinson, M.P., Buneman, O.P.: Types and persistence in database programming languages. ACM Computing Surveys 19(2), 105–170 (1987)
3. Bawden, A.: Quasiquotation in Lisp. In: Partial Evaluation and Semantic-Based Program Manipulation, pp. 4–12 (1999)

4. Bézivin, J., Gerbé, O.: Towards a precise definition of the OMG/MDA framework. In: Proceedings of Automated Software Engineering (ASE 2001), pp. 273–282. IEEE Computer Society, Los Alamitos (2001)
5. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.: Eclipse Modeling Framework. Addison Wesley Professional, Reading (2003)
6. Chiba, S., Kiczales, G., Lamping, J.: Avoiding confusion in metacircularity: The meta-helix. In: Futatsugi, K., Matsuoka, S. (eds.) ISOTAS 1996. LNCS, vol. 1049, pp. 157–172. Springer, Heidelberg (1996)
7. Costanza, P., Hirschfeld, R.: Language constructs for context-oriented programming: An overview of ContextL. In: Proceedings of the Dynamic Languages Symposium (DLS) 2005, co-organized with OOPSLA 2005, pp. 1–10. ACM, New York (2005)
8. Czarnecki, K., Eisenecker, U.W.: Generative programming: methods, tools, and applications. ACM Press/Addison-Wesley Publishing Co., New York (2000)
9. Denker, M.: Sub-method Structural and Behavioral Reflection. PhD thesis, University of Bern (May 2008)
10. Denker, M., Ducasse, S., Lienhard, A., Marschall, P.: Sub-method reflection. Journal of Object Technology, Special Issue (2007); Proceedings of TOOLS Europe 2007, vol. 6/9, pp. 231–251 (2007)
11. Denker, M., Gîrba, T., Lienhard, A., Nierstrasz, O., Renggli, L., Zumkehr, P.: Encapsulating and exploiting change with Changeboxes. In: Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007), pp. 25–49. ACM Digital Library, New York (2007)
12. Denker, M., Greevy, O., Lanza, M.: Higher abstractions for dynamic analysis. In: 2nd International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006), pp. 32–38 (2006)
13. Denker, M., Greevy, O., Nierstrasz, O.: Supporting feature analysis with runtime annotations. In: Proceedings of the 3rd International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2007), pp. 29–33. Technische Universiteit Delft (2007)
14. Denker, M., Suen, M., Ducasse, S.: The meta in meta-object architectures. In: Proceedings of TOOLS EUROPE 2008. LNBIP, vol. 11, pp. 218–237 (2008)
15. van Deursen, A., Klint, P.: Little languages: Little maintenance? In: Kamin, S. (ed.) First ACM-SIGPLAN Workshop on Domain-Specific Languages, DSL 1997, January 1997, pp. 109–127 (1997)
16. Fowler, M.: Language workbenches: The killer-app for domain-specific languages (June 2005), `http://www.martinfowler.com/articles/languageWorkbench.html`
17. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Reading (1995)
18. González, S., Mens, K., Heymans, P.: Highly dynamic behaviour adaptability through prototypes with subjective multimethods. In: DLS 2007: Proceedings of the 2007 symposium on Dynamic languages, pp. 77–88. ACM, New York (2007)
19. Haldimann, N., Denker, M., Nierstrasz, O.: Practical, pluggable types for a dynamic language. Journal of Computer Languages, Systems and Structures 35(1), 48–64 (2009)
20. Haustein, S., Pleumann, J.: A model-driven runtime environment for web applications. Software and System Modeling 4(4), 443–458 (2005)
21. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology 7(3) (March 2008)
22. Kiczales, G., des Rivières, J., Bobrow, D.G.: The Art of the Metaobject Protocol. MIT Press, Cambridge (1991)

23. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
24. Knuth, D.E.: Literate Programming. Center for the Study of Language and Information, Stanford (1992)
25. Krahn, H., Rumpe, B., Völkel, S.: Integrated definition of abstract and concrete syntax for textual languages. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 286–300. Springer, Heidelberg (2007)
26. Lehman, M., Belady, L.: Program Evolution: Processes of Software Change. Academic Press, London (1985)
27. Magee, J., Dulay, N., Eisenbach, S., Kramer, J.: Specifying distributed software architectures. In: Botella, P., Schäfer, W. (eds.) ESEC 1995. LNCS, vol. 989, pp. 137–153. Springer, Heidelberg (1995)
28. Marschall, P.: Persephone: Taking Smalltalk reflection to the sub-method level. Master's thesis, University of Bern (December 2006)
29. Nelson, S., Pearce, D.J., Noble, J.: First class relationships for OO languages. In: Proceedings of the 6th International Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL 2008) (2008)
30. Nierstrasz, O., Denker, M., Gîrba, T., Lienhard, A.: Analyzing, capturing and taming software change. In: Proceedings of the Workshop on Revival of Dynamic Languages (co-located with ECOOP 2006) (July 2006)
31. Nierstrasz, O., Denker, M., Gîrba, T., Lienhard, A., Röthlisberger, D.: Change-enabled software systems. In: Wirsing, M., Banâtre, J.-P., Hölzl, M. (eds.) Challenges for Software-Intensive Systems and New Computing Paradigms. LNCS, vol. 5380, pp. 64–79. Springer, Heidelberg (2008)
32. Nixon, B., Chung, L., Mylopoulos, J., Lauzon, D., Borgida, A., Stanley, M.: Implementation of a compiler for a semantic data model: Experiences with taxis. In: SIGMOD 1987: Proceedings of the 1987 ACM SIGMOD international conference on Management of data, pp. 118–131. ACM, New York (1987)
33. Odersky, M.: Scala language secification v. 2.4. Technical report, École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland (March 2007)
34. Ousterhout, J.K.: Scripting: Higher level programming for the 21st century. IEEE Computer 31(3), 23–30 (1998)
35. Pawson, R.: Naked Objects. Ph.D. thesis, Trinity College, Dublin (2004)
36. Renggli, L., Ducasse, S., Kuhn, A.: Magritte — a meta-driven approach to empower developers and end users. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 106–120. Springer, Heidelberg (2007)
37. Renggli, L., Nierstrasz, O.: Transactional memory in a dynamic language. Journal of Computer Languages, Systems and Structures 35(1), 21–30 (2009)
38. Rivard, F.: Smalltalk: a reflective language. In: Proceedings of REFLECTION 1996, April 1996, pp. 21–38 (1996)
39. Rodríguez, L., Tanter, É., Noyé, J.: Supporting dynamic crosscutting with partial behavioral reflection: a case study. In: Proceedings of the XXIV International Conference of the Chilean Computer Science Society (SCCC 2004), Arica, Chile. IEEE, Los Alamitos (2004)
40. Röthlisberger, D., Denker, M., Tanter, É.: Unanticipated partial behavioral reflection: Adapting applications at runtime. Journal of Computer Languages, Systems and Structures 34(2-3), 46–65 (2008)
41. Röthlisberger, D., Greevy, O., Nierstrasz, O.: Exploiting runtime information in the IDE. In: Proceedings of the 16th International Conference on Program Comprehension (ICPC 2008), pp. 63–72. IEEE Computer Society, Los Alamitos (2008)

42. Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. Computer 39(2), 25–31 (2006)
43. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, Englewood Cliffs (1996)
44. Sheard, T.: Accomplishments and research challenges in meta-programming. In: Taha, W. (ed.) SAIG 2001. LNCS, vol. 2196, pp. 2–44. Springer, Heidelberg (2001)
45. Tanter, E.: Reflection and open implementations. Technical report, University of Chile (2004)
46. Tanter, É., Gybels, K., Denker, M., Bergel, A.: Context-aware aspects. In: Löwe, W., Südholt, M. (eds.) SC 2006. LNCS, vol. 4089, pp. 227–242. Springer, Heidelberg (2006)
47. Tanter, É., Noyé, J., Caromel, D., Cointe, P.: Partial behavioral reflection: Spatial and temporal selection of reification. In: Proceedings of OOPSLA 2003, ACM SIGPLAN Notices, November 2003, pp. 27–46 (2003)
48. Tratt, L.: Domain specific language implementation via compile-time meta-programming. ACM TOPLAS 30(6), 1–40 (2008)
49. Warth, A., Piumarta, I.: OMeta: an object-oriented language for pattern matching. In: DLS 2007: Proceedings of the 2007 symposium on Dynamic languages, pp. 11–19. ACM, New York (2007)
50. Wirsing, M., Hölzl, M. (eds.): Report of the Beyond the Horizon thematic group 6 on Software Intensive Systems (2006)

Modeling of Context-Aware Self-Adaptive Applications in Ubiquitous and Service-Oriented Environments

Kurt Geihs, Roland Reichle, Michael Wagner, and Mohammad Ullah Khan

Universität Kassel, Wilhelmshöher Allee 73,
34121 Kassel, Germany
{geihs,reichle,wagner,khan}@vs.uni-kassel.de

Abstract. Mobile computing in ubiquitous environments has to cope with both predictable and unpredictable changes in the execution context, which introduces the need for context-aware adaptive applications. Such environments are also characterized by dynamically discoverable services that can be utilized by applications to improve their functionality and quality of service (QoS). Thus, application adaptation decisions not only depend on context properties, but also on service availability and QoS-properties. In this chapter we present a novel comprehensive modeling approach that facilitates the model-driven development of such applications. Our focus is on modeling concepts which align the description of services and their QoS-properties with the context modeling approach. We provide a harmonized view on context and service properties, bridging the syntactical and semantic differences through an ontology. We also consider related aspects like semantic service discovery and service level agreements.

Keywords: context awareness, compositional adaptation, ontology, service-oriented architecture, ubiquitous computing, utility function.

1 Introduction

Mobile computing in ubiquitous environments triggers a need for dynamically adaptable applications that can be reconfigured during run-time, in order to remain useful under the changing context situation. For example, an application may need to adapt when connectivity changes, battery is running low, user preferences change, some specific sensor value exceeds a predefined threshold, or services or devices appear or disappear.

Context-aware adaptive applications based on component frameworks have been a subject of intensive research for several years. Adaptation usually involves a reconfiguration of the component composition to suit the changed context. In the EU-IST project MUSIC [1] we explore an advanced kind of compositional adaptation by considering dynamically discovered services as possible replacements for application components. Clearly, this opens up new possibilities as well as challenges for the application adaptation. In the companion chapter "MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments" [20] in this volume a new planning framework has been introduced that supports self-adaptation of mobile applications with regard to changes in the service landscape. By the term

B.H.C. Cheng et al. (Eds.): Self-Adaptive Systems, LNCS 5525, pp. 146–163, 2009.

'planning' we refer to the computation and evaluation of alternative application configurations in response to context changes and the selection of a feasible configuration, best-fitted for the current context situation.

Our adaptation approach extends compositional adaptation by adding service-based adaptation. Services can be dynamically discovered and bound by applications. A newly discovered service may replace a component (or another service) in a component-based application if it increases the overall application utility. Our adaptation planning framework takes service properties and service level agreements into account when computing an adaptation decision in reaction to a context change.

A major goal of our research is to facilitate the development of self-adaptive applications. We adopt a model-driven development approach that builds on available modeling languages for service-oriented computing with regard to functionality, interfaces, QoS-properties and semantic properties. The main contribution of this chapter is a new modeling framework that explicitly addresses heterogeneity aspects that arise from the independent development of services, context sensors and reasoners, as well as the application itself. It provides a coherent set of concepts and a modeling methodology that integrate seamlessly into an adaptation planning framework for context-aware, component-based adaptive applications. The modeling concepts have been designed such that the description of services and their QoS-properties align with the context model. Since both QoS-properties and the context model are based on a coherent set of ontologies, our approach achieves the bridging of syntactical and semantic differences. Thus, it provides a harmonized, integrative view on context and service properties. Furthermore, it also covers aspects like semantic service discovery and service level agreements.

The chapter is organized as follows. We start with a motivating scenario in section 2. In section 3 we identify the requirements of context-aware adaptive applications in ubiquitous computing environments that serve as a guideline for our work. The main concepts of the new application variability model and the adaptation reasoning approach are explained in section 4. The integration of service-based adaptation is presented in section 5. Section 6 explains the approach using a modeling example based on the scenario of section 2. Related work is discussed in section 7. Section 8 concludes the chapter and presents perspectives for future work.

2 Motivating Scenario

The following scenario shall motivate the need for service-based adaptations. A sales agent visits customers and uses a context-aware Customer Relationship Management (CRM) system, which offers functions like accessing and sharing customer- and business-related information, route planning, calculation of travelling delays and notifying people, who are affected by delays.

Scene 1: The sales agent meets a customer and uses the CRM system on his smart-phone to record agreements. During the meeting he is notified about a new appointment at another customer site and decides to prepare for it. He uses the CRM application on his smart-phone to find the best route and to estimate the travel time. For this, the application uses a navigation service provided over the WLAN at the customer site and a traffic information service available through the Internet. There

are several providers for both of these services, and the CRM application selects the one that offers a quick and precise solution to the agent.

Scene 2: After the meeting, the agent walks out of the customer's building and gets into the car to head for the next meeting. As he moves away from the customer site, the smart-phone loses connection to the customer WLAN, and the Internet connection is switched seamlessly to GPRS by the mobile IP software installed on the smart-phone. The connection to the navigation service, which the CRM application needs to monitor the progress towards the destination, is lost. The GPRS provider also offers a navigation service, but with a rather low accuracy. The car has a navigation system based on GPS, which provides not only a more accurate navigation service via a Bluetooth connection but also a better display. The CRM application automatically reconfigures itself to use these services. It also saves battery life on the smart-phone by offloading local processing to the external services.

Scene 3: Halfway to the meeting, the agent runs into a traffic jam. The CRM application detects this situation, alerts the agent that he will be late, estimates the delay using data obtained from the traffic information service and offers the agent to notify affected customers. The agent validates this proposal and the CRM application sends messages to the customers.

Our goal is to provide generic support for such kind of automatic service-based adaptations. Therefore, we have developed appropriate modeling concepts and integrated them into a planning framework for compositional adaptation.

3 Requirements

In order to facilitate the development and operation of context-aware self-adaptive applications in ubiquitous environments, we provide a model-driven development approach, an adaptation middleware, and suitable development tools. The middleware separates context sensing, adaptation reasoning and application reconfiguration from the pure application logic. It provides a generic and reusable infrastructure for a wide range of adaptive applications. See [20] for details of the middleware and the adaptation approach. The middleware is complemented by a model-driven development approach that allows the specification of application adaptation capabilities at a high and platform-independent level and facilitates automatic generation of source-code tailored for the middleware.

In order to realize context-aware adaptive applications in ubiquitous environments, the adaptation approach and the adaptation models have to cope with the following requirements:

- **Application variability:** Dynamic reconfigurations and reasoning about different configuration alternatives require the definition of an application variability model by the application developer. Since adaptations are intended to maximize the QoS perceived by the user, the variability model has to be enriched with QoS-metadata of the involved components.
- **Dynamic service discovery:** Ubiquitous computing environments are assumed to offer services that can be discovered and accessed dynamically at runtime. This requires specification means for the integration of external services into the

application variability model. Their QoS-properties must be made available for the adaptation reasoning process. Therefore, support for semantic service discovery, semantic description of QoS-properties and service level negotiations is required.

- **Heterogeneity:** Services in ubiquitous computing environments may have been developed independently by different parties. This implies that QoS-properties of services may have different names and representations. Likewise, heterogeneity may be found with the context management components, in particular if third party context sensors and reasoners are integrated. Thus, QoS-properties and context information that describe the properties of the execution context have to be semantically enriched in order to enable interoperability and integration.
- **Integration of service and context properties into the planning:** The adaptation planning is responsible for evaluating the usefulness of alternative application configurations under a given context situation. Therefore, QoS-properties of available components and services have to be related to the properties of the execution context.

In particular with respect to the heterogeneity aspect, we argue that a comprehensive modeling notation is required to provide a common vocabulary that enables developers of different components to share a harmonized semantic view on the QoS-properties of involved components/services and the properties of the execution context. Our ontology-based modeling approach, as discussed in the following sections, provides such a vocabulary and thus eases the development task. Without such kind of support, bridging syntactical and semantic differences of QoS-properties and context information and their mediation is hard to achieve. Furthermore, the modeling concepts have to be aligned to the underlying adaptation planning framework and have to include provisions for semantic service discovery and description of service levels and service level agreements.

In a previous publication [16], we already presented an ontological framework for context modeling. Here we present a very substantial enhancement and generalization of that approach by considering the additional aspects and challenges arising from a service-oriented ubiquitous computing environment.

4 Application Variability and Adaptation Reasoning

The following explanations require a basic understanding of the underlying adaptation approach. Therefore, we shortly describe its most important concepts. For a more detailed description the reader is referred to our companion chapter [20] that presents the corresponding run-time support for planning and execution of adaptations.

Applications are based on a component framework model that supports dynamic reconfiguration at run-time. When there is a significant context change the middleware evaluates and compares all available application variants based on different QoS-metadata associated to the involved component realizations. Thus we consider applications that are developed with a QoS-oriented component model, which defines all reasoning dimensions used by the planning-based middleware to select and deploy the component implementation that provides the best utility.

The utility of a component configuration is computed using a developer-defined utility function. Those parts of the application that are evaluated during planning are

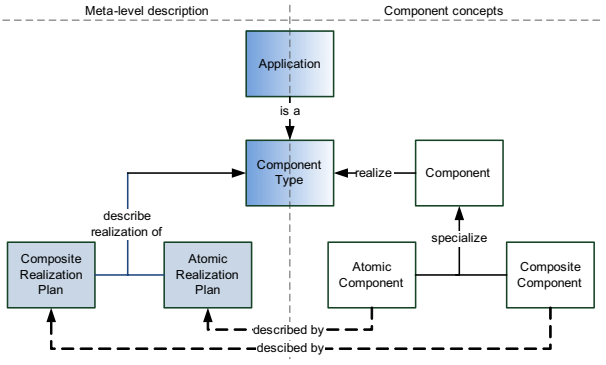


Fig. 1. Creating application variants

called variation points. Each variation point identifies a functionality of the application that can have different realizations. In our case the variation points are component types corresponding to a certain part-functionality of the application. Each component implementation suitable for a variation point is reified as a plan, which mainly consists of a meta-structure that reflects the properties of the component implementation. A plan exhibits both requested properties (e.g., memory consumption, network bandwidth, monetary cost) and offered properties (e.g., throughput, response time, result accuracy) referring to the QoS model of the application. Property predictors help estimating the offered properties of the component associated with a plan. They predict the values of non-functional offered properties of a component (or component composition) as a function of required properties depending on a given execution context. These predicted property values are input arguments to a normalized utility function to obtain the expected utility of the component in the given context. The planning middleware compares the expected utility of all alternate component configurations, and finally selects the one that provides the highest utility value.

Figure 1 illustrates that an application is viewed as a *Component Type* that can have different realizations. The details and the QoS-properties of a certain realization are described using *Plans*. Corresponding to the atomic and composite component types, there are two types of Plans: *Atomic Realization* and *Composite Realization*. An Atomic Realization Plan describes an atomic component and contains just a reference to the class or the data structure that realizes the component. The Composite Realization Plan describes the internal structure of a composite component by specifying the involved Component Types and the connections between them.

Variation is obtained by describing a set of possible realizations of a Component Type using Plans. In order to create a possible variant, one of the Plans of a Component Type is selected. If the Plan is a Composite Realization Plan, it describes a collaboration structure of further Component Types, which in turn are described by Plans again. Now we proceed by recursively selecting one realizing Plan for every involved Component Type. The recursion stops if an Atomic Realization Plan is chosen. Therefore, by resolving the variation points we create application variants that correspond to a certain composition of components depending on the plans that are chosen for each of the Component Types.

With service-based adaptation a part-functionality may be provided through a dynamically discoverable and accessible service. Thus, compositional adaptation is extended by taking a service as a possible realization of a Component Type. To do so, the QoS-properties, interfaces and binding information have to be included in a corresponding plan.

Service discovery protocols integrated in the middleware advertise any newly discovered services to a plan broker. Plans for these services and from known service repositories are generated from service level descriptions so that they are available when the planner initiates an adaptation at a later time. Of course, plans are discarded when services become unavailable to the middleware and an adaptation process is triggered if a service described by the discarded plan is currently in use.

A service might offer a predefined set of service levels. Then, for each service level a separate plan is generated by the plan broker. Thus, the planning framework is able to take service levels into account when planning the adaptation.

Two kinds of property predictors, representing service levels, can be associated with plans. If the service level is fixed, then the property predictors report the values defined in the SLA contracts associated with the plan. In case of a choice of service levels, the property predictors iterate over the service level property values to calculate the utility of the application variant.

5 Modeling of Service-Based Adaptation

Our modeling notation provides means for the creation of the application variability model together with QoS-metadata, property predictors and utility functions. It also facilitates the semantic description of the execution environment, in terms of context and resource models, and includes specification means for external services, service levels and service level agreements. In the following we focus on the specific modeling concepts for the integration of external services into compositional adaptation.

5.1 Semantic Annotation of Variation Points

The variability model consists of application type, component types and realizing plans. We have developed a new UML Profile [12] to support such modeling tasks. In order to consider services as a possible replacement for certain components, the corresponding component types have to be annotated with a set of descriptions specifying the expected functionality, the required interfaces and the expected set of QoS-properties. Such annotations are only required for component types that may be the target of dynamic service discovery. Clearly, core components that are crucial for the general functionality of the application will most likely not be candidates for dynamic substitution by externally provided services.

The semantic web community has addressed very similar challenges with regard to semantic service discovery and matching. We intend to reuse their approaches and modeling support as much as possible and thus, we associate OWL-S descriptions to the corresponding component types. As we only want to state our requirements for an external service, only the service profile and process model are needed, while the service grounding is omitted. However, we avoid, as far as possible, that a developer

has to bother with the OWL-S descriptions and the ontology modeling. Therefore, we generate the most relevant parts of the OWL-S descriptions from the UML application variability model where the component types are explicitly modeled through their provided and required ports and the corresponding interfaces. The interfaces also indicate the input and output parameters and their data-types (which in OWL-S terminology are called messages). We provide an UML-to-OWL-S transformation, which is similar to WSDL2OWL-S [15, 23] and to the one provided through the OWL-S Editor of the University of Malta [24]. However, they generate OWL-S descriptions from WSDL specifications, while we start with UML models.

The resulting OWL-S description associated with the component type is used for service discovery and matching. We support two levels of matching: a high-level matching mechanism considering the service category included in the service profile and a low-level approach considering also ports and interfaces. For the high-level approach we support both existing taxonomies of service categories like NAICS [7] as well as the definition of own taxonomies in the form of simple OWL ontologies. The high-level matching approach uses only taxonomical information assuming that the service provides the expected ports and interfaces. This is particularly useful for simple and fast service matching on mobile devices with limited computation capabilities.

In order to enable QoS-driven adaptation reasoning, the services are expected to provide information about the offered QoS-properties that are evaluated in property predictors and/or the utility function. Therefore, we must be able to specify the QoS dimensions for which the discovered services are expected to provide information.

5.2 MUSIC Ontology Concepts

In section 3 we highlighted the need for a harmonized view on QoS-properties of services and properties of the user and execution context. This is achieved by the MUSIC Ontology. It establishes a common vocabulary for QoS-properties and context information and serves as the baseline for information management and representation in MUSIC. We also define an extension to the OWL-S profile ontology that facilitates semantic service discovery, description of service levels and service level agreements.

5.2.1 Two-Level Hierarchy

A critical issue with processing ontologies, in particular on resource constraint mobile devices, is the number of defined classes, properties and relationships. Therefore, the MUSIC Ontology is divided into a two-level hierarchy. We distinguish between domain-specific and general entities, in the same way as proposed in [25, 26]. The top-level ontology captures general knowledge to provide a semantic vocabulary, which is applicable for most foreseen applications. It also consists of extensions that we provide for the integration of external discoverable services into the adaptation planning process.

The top-level ontology is complemented by domain-specific sub-ontologies that can be plugged-in for applications of a particular application domain. Whereas the top-level ontology is the stable part of the ontology, the domain-specific sub-ontologies are regarded as the extensible parts.

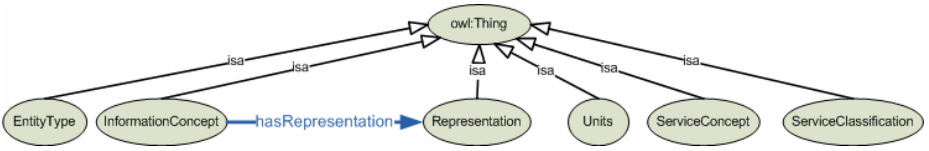


Fig. 2. Top-level concepts of the ontology

5.2.2 Top-Level Concepts of the MUSIC Ontology

The top-level concepts of the ontology consist of *EntityType*, *InformationConcept*, *Representation*, *Unit*, *ServiceClassification* and *ServiceConcept* as shown in Figure 2. *EntityTypes* categorize the entities of the world that are characterized through context information. Examples of entity types are *Person*, *Device*, and *ResourceEntity*. *InformationConcept* encapsulates *Scopes* and *Metadata*, which are used to represent the actual information that is provided for an entity of a certain type. *Scope* represents the type of information, e.g. *Location*, and *Metadata* concepts can be associated to context or resource information, in order to provide additional meta-information, e.g. a *TimeStamp* that specifies the time of sensing the corresponding data. QoS-properties are modeled as a specialization of *Scope*. They are considered as a special information type that expresses the characteristics of a service with regard to its quality.

Each *InformationConcept* is associated to a *Representation* concept, which defines how the information is internally structured with regard to *Scopes*, *Metadata*, *DataType* properties and *Units*. Here, we allow more than one representation for an *InformationConcept*, in order to explicitly deal with the heterogeneous nature of the ubiquitous computing environment. *Units* concept is used to define measurement units in a way that allows their automatic conversion. We distinguish between *BaseUnits*, e.g. *Meter* or *Byte*, and *DerivedUnits* that can be derived from one or more *BaseUnits*. The *ServiceClassification* concept helps referring to the functionality of a service through the definition of a taxonomy of service functionalities.

Further modeling concepts required for the integration of external services in the adaptation reasoning process are provided as extensions to the OWL-S Service profile ontology. These concepts are generalized by the *ServiceConcept* class of the MUSIC Ontology and are described in more detail in section 5.2.4.

5.2.3 Information Concept and Representation

In order to explain further the terms *InformationConcept* and *Representation*, a simple example is shown in Figure 3. It shows that the class hierarchy of representations reflects corresponding to the class hierarchy of the information concepts. For instance, as *Required_Bandwidth* is a *QoSProperty* which is in turn an *InformationConcept*, the *RequiredBandwidthRepresentation* is a specialization of *QoSPropertyRepresentation* which is in turn a specialization of *Representation*. As the *QoSProperty Required_Bandwidth* is associated with *RequiredBandwidthRepresentation* which provides different specializations, namely *ReqBandwidth_DefaultRep_KBps, ReqBandwidth_DefaultRep_MBps*, etc., the *Required_Bandwidth QoSProperty* can be represented in different ways. Support for different representations is one way of coping with the heterogeneity that one must expect if context sensors and/or services are developed independently.

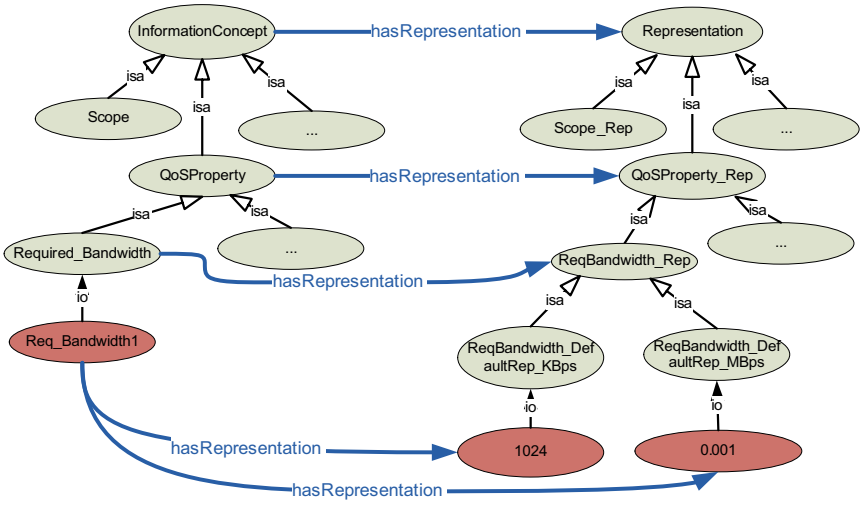


Fig. 3. Example for information concepts and representations

We also include support for the definition of Inter-Representation-Operations (IRO), as developed in the ASC project [19]. For example, it allows a consumer to ask for data of a certain *Scope*, characterizing a certain *Entity* and having a certain *Representation*. If this does not match the representation provided by the context sensor, an appropriate representation can be computed with the corresponding IRO.

5.2.4 Extensions to the OWL-S Profile Ontology

Semantic service discovery and matching are facilitated through the association of component types with a semantic description of the corresponding part-functionalities along with a description of the port types and interfaces. In order to enable QoS-driven adaptation planning, the services are expected to provide information about their offered QoS-properties that are evaluated in property predictors and/or the utility function. Therefore, we must be able to specify the QoS-dimensions that we expect from discovered services.

In Service Level Management, QoS-properties of services are defined in Service Level Agreements and established by a service level negotiation process. As we want to evaluate the discovered services with regard to their QoS-properties, the QoS-properties should already be subject to the service discovery process. At best, a service description used for service discovery would include specifications of the service functionality, the QoS-dimensions that the service should provide information on and the required QoS-properties to improve the specification of the utility function.

The OWL-S description associated to a component type contains the corresponding specifications and is used for service discovery. Therefore, we have extended the OWL-S Service Profile ontology to provide machine understandable information on QoS dimensions and also on service level requirements to enable a quality-aware service discovery. The ontology includes the specification of Service Level Dimensions (SLDs) and Service Levels (SLs) on these dimensions. The SLDs correspond to

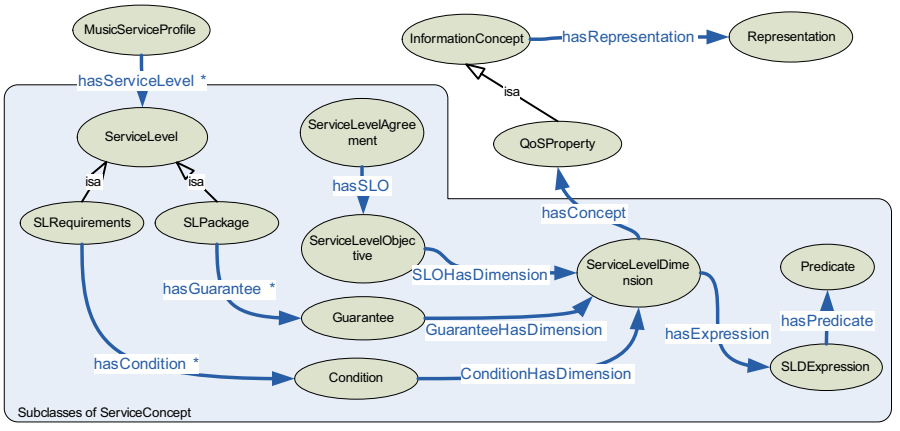


Fig. 4. MUSIC ontology used for quality-aware service discovery

the QoS-dimensions used in the property predictors and/or the utility function, and the specification of an SL allows service providers and service consumers to specify their interest in QoS with regard to the SLDs. We distinguish between two types of SLs, namely, Service Level Requirements (SLR) and Service Level Package (SLP). The service providers can offer different service level packages for one service, which means that they offer different quality levels at different conditions. A quality level consists of several obligations. Each obligation offers a guarantee on an SLD. Service providers and consumers negotiate a certain quality level by agreeing on obligations on quality dimensions and the result is an SLA with several Service Level Objectives (SLOs).

The specification of the QoS-properties could be included either into the service profile ontology or into the ontology for the service's process model. Including it into the process model would allow for more flexibility as QoS-properties could be associated to single processes. However, in the current adaptation planning process it is only possible to consider the QoS-properties of the whole service. Therefore, we include the QoS-property specification into the service profile ontology.

Figure 4 shows the main classes and their relationships defined in the ontology used for quality-aware service discovery: *ServiceLevel*, *ServiceLevelDimension* (SLD), *QoS-Property* and *Representation*. Each SLD refers to a *QoSProperty*, which is a specialization of the class *InformationConcept* and, thus, refers to a representation through the property *hasRepresentation*. By providing several representations for one concept and appropriate Inter-Representation-Operations we enable the automatic conversion of an individual of one concept with a special representation into an individual of this concept with another representation. In particular, this applies to the class *Units* which is a specialization of the class *Representation*. For example, a bandwidth quantity "KByte/s" can be transformed into "MBit/s". By providing different representations for one concept, we cope with different terms and metrics used by different parties to describe the same QoS-dimensions.

A Service Level Package and a Service Level Requirement are specializations of a Service Level. As aforementioned, a Service Level consists of several conditions

and/or obligations. Obligations are expressed by guarantees on quality dimensions (SLDs). If the service provider and the service consumer agree on a guarantee, it becomes an obligated guarantee in the resulting Service Level Objective in the resulting SLA. Each condition and each guarantee consist of a predicate (e.g. lessthan), a SLD (e.g. *ResponseTime*) with its associated representation/unit (e.g. second), a variable and a value, e.g. *lessthan(ResponseTime, x, 200, milliseconds)*.

Usually, the obligated guarantees are created by a negotiation process between the service provider and the service consumer. A matching algorithm can be applied to identify service candidates along with service level requirements for further consideration in the planning framework. The SLP's guarantees are compared to the SLR's conditions and vice versa. If all conditions are fulfilled, the SLP becomes a candidate for the Planning framework. This does not need to be the SLP which fits best to the SLR. The SLR only acts as a filter to find a set of alternative services. For the service level negotiation, currently only a few standards are available. The most mature protocol seems to be the XML-based WS-Agreement specification [27]. However, WS-Agreement is not designed to be semantically interpreted, i.e. it does not foresee references to an ontology. Therefore, we will enhance the standard with semantic annotations in a similar way as it was done with WSDL-S for WSDL and we will establish a mapping from our OWL-S descriptions to specifications that can be used in WS-Agreement service level negotiations.

5.3 Characteristics of QoS-Properties

In reality we cannot expect that all the discovered services are able to provide QoS information on exactly the set of QoS dimensions expected in the utility function. Therefore, we have to live with a mismatch between the expected and provided QoS dimensions.

QoS-properties of external services are provided to the planning framework by property evaluators and property predictors. These can be arbitrarily complex functions ranging from just returning a constant value to predicting the QoS-properties based on context information and other QoS-properties. Besides, property evaluators also mediate the QoS-properties of the external service with regard to performing inter-representation-operations. This means, the utility function requests information on a QoS dimension in a certain representation (metric) and the property evaluator is responsible to provide the information in the corresponding representation, e.g. by performing inter-representation-operations. For the modeling of property evaluators and predictors we had already provided a basic support in the MADAM project [2, 3]. However, with regard to service-based adaptation we have to extend that support in order to deal with QoS-properties that are not provided by the discovered service; e.g., we can assign a default/derived value in such cases. If this is not reasonable or possible at all, the service cannot be considered in the adaptation reasoning process.

6 Example

In this section, we show how the introduced modeling framework can be utilized to realize the CRM Application in section 2. Please note that our particular focus is on

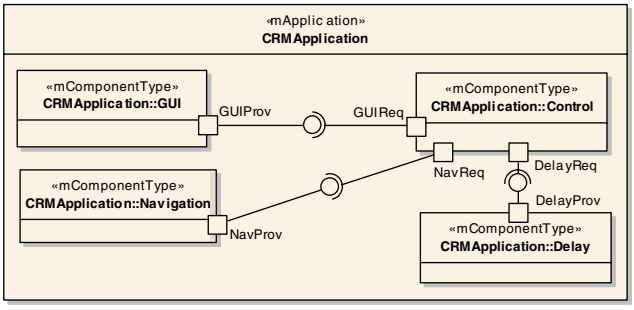


Fig. 5. UML composite structure for the CRM application

the integration of services into the application, and the related modeling aspects. Hence, we do not show how the models are utilized by the run-time environment. The corresponding middleware that provides the run-time environment is presented in detail in our companion chapter [20]. In general, thanks to our model-driven development approach, a developer can specify the adaptation capabilities and context-awareness of an application at a high level and is not confronted with implementation details (see [3] for more details on the model-driven development approach of the MUSIC project).

The variability model of the example application is based on the UML composite structure diagram, as shown in Figure 5.

The application is composed of four component types: GUI (user interface), Control (main application logic), Delay (traffic information), and Navigation (route planning and navigation). By allowing different realizations for each component type a number of application variants can be derived from this simple specification that are created at run-time by the *Adaptation Manager* of the adaptation middleware [20]. Each of the involved component types is further specified in an additional UML class diagram defining the corresponding port types and interfaces.

In order to mark the *Navigation* Component Type as a component that can be realized through external services and to facilitate semantic service discovery and matching, the *NavProv* port type of the *Navigation* component type is associated (UML dependency) to a *PortDescription*.

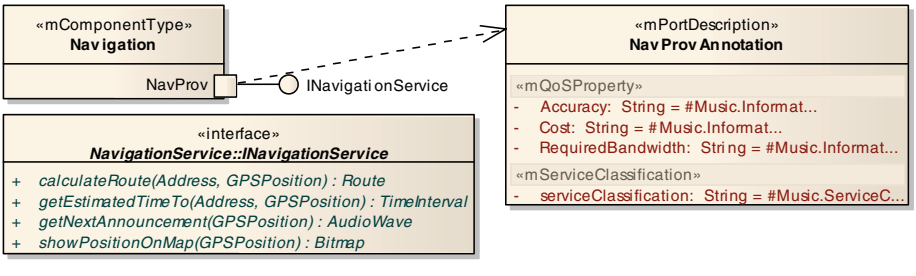


Fig. 6. Semantic annotation of the Navigation component type

```
<music-extension:MusicServiceProfile rdf:ID="MusicServiceProfile_NavigationService">
   <profile:serviceClassification rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
   http://www.ist-music.eu/MUSICServiceTaxonomy.owl#NavigationService</profile:serviceClassification>
   <profile:textDescription rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
   provides a basic navigation service to MUSIC applications</profile:textDescription>
   <profile:serviceName rdf:datatype="...#string">NavigationService</profile:serviceName>
      <music-extension:hasServiceLevel>
        <music-extension:ServiceLevelRequirements rdf:ID="ServiceLevelRequirements_Navigation">
          <music-extension:hasCondition>
            <music-extension:Condition rdf:ID="Condition_Bandwidth">
              <music-extension:conditionHasDimension>
                <music-extension:ServiceLevelDimension rdf:ID="Dimension_Bandwidth">
                  <music-extension:hasConcept>
                    <music:RequiredBandwidth rdf:ID="RequiredBandwidth">
                      <music:hasRepresentation rdf:resource="...#ReqBandwidth_DefaultRep_KBytePerS"/>
                    </music:RequiredBandwidth>
                  </music-extension:hasConcept>
                </music-extension:ServiceLevelDimension>
              </music-extension:conditionHasDimension>
            </music-extension:Condition>
            <music-extension:Condition rdf:ID="Condition_Cost">
              ...
            </music-extension:Condition>
            <music-extension:Condition rdf:ID="Condition_Accuracy">
              ...
            </music-extension:Condition>
          </music-extension:hasCondition>
        </music-extension:ServiceLevelRequirements>
      </music-extension:hasServiceLevel>
</music-extension:MusicServiceProfile>
```

Fig. 7. Example of a generated OWL-S profile

The port description includes attributes *service classification* to categorize the expected functionality and QoS-properties expected from the discovered service. The default value of such attributes has the structure *<QoSProperty>:<Representation>* and points to the corresponding semantic concept in the ontology and the expected representation. This UML specification of the Navigation Component Type serves as basis for a UML-to-WSDL and WSDL-to-OWL-S transformation. An extract of the resulting OWL-S Profile specification is shown in Figure 7.

The OWL-S Profile description includes a *MUSICServiceProfile* for the *NavigationService*. Apart from the service classification, it mainly consists of a *ServiceLevelRequirements* specification, which defines the corresponding QoS-properties in *ServiceLevelDimensions* of conditions. The rationale for this is that we also foresee a QoS-aware service discovery. Thus we allow the specification of expressions that are associated to a QoS-Dimension in a condition. However, in this example we just want to define the QoS-properties that the discovered service should support. Therefore, expressions can be omitted here.

An example of a corresponding OWL-S profile description as expected from a discovered service is depicted in Figure 8. The profile description defines a *ServiceLevelPackage* that expresses a guarantee that the cost for invoking the service will be less than 1.25 Dollar. The conversion to Euro as the expected currency (see previous profile description) is automatically handled by the corresponding Inter-Representation-Operation.

```
<music-extension:ServiceLevelPackage rdf:ID="ServiceLevelPackage_Navigation_LowCost">
  <music-extension:hasGuarantee>
    <music-extension:Guarantee rdf:ID="Guarantee_Cost">
      <music-extension:guaranteeHasDimension>
        <music-extension:ServiceLevelDimension rdf:ID="Dimension_Cost">
          <music-extension:hasConcept>
            <music:Cost rdf:ID="Cost">
              <music:hasRepresentation rdf:resource="...#Cost_DefaultRep_Dollar"/>
            </music:Cost>
          </music-extension:hasConcept>
          <music-extension:hasExpression>
            <music-extension:SLDExpression rdf:ID="SLDExpression_1">
              <music-extension:hasPredicate rdf:resource="...#lessthan"/>
              <music-extension:Value rdf:datatype="...#float">1.25</music-extension:Value>
              <music-extension:Variable rdf:datatype="...#string">Cost</music-extension:Variable>
            </music-extension:SLDExpression>
          </music-extension:hasExpression>
        </music-extension:ServiceLevelDimension>
      </music-extension:guaranteeHasDimension>
    </music-extension:Guarantee>
  </music-extension:hasGuarantee>
</music-extension:ServiceLevelPackage>
```

Fig. 8. Example of OWL-S profile description of a discovered service

The necessity and importance of basing the properties of the execution context and the QoS-properties of an external discoverable service on a common semantic vocabulary defined in an ontology becomes obvious, when looking at the following utility function of the application:

$$utility = \frac{1}{1+e^{-10.0\cdot(AvBandwidth-ReqBandwidth)}} \cdot \left(user_{cost} \cdot \left(\frac{1}{1+0.5\cdot Cost} \right) + user_{acc} \cdot \left(\frac{Accuracy}{\max Accuracy} \right)^{0.8} \right)$$

The utility function takes into account user preferences for the influence of the cost and the accuracy of the utilized service[1]. The corresponding terms result in a lower utility for high costs and a higher utility for a higher accuracy. Besides, the utility function compares the *AvailableBandwidth* provided by the execution environment and the *RequiredBandwidth* of the discovered service (or more precisely, of the application variant utilizing the service) and adjusts the utility through a sigmoid function. If the corresponding context sensor estimating the currently available bandwidth and the external service are developed independently, a common vocabulary as established by the MUSIC Ontology is indispensable.

7 Related Work

Many projects have already addressed the challenge of realizing context-aware adaptive applications. One of the most famous works in this area is the Context Toolkit by

[1] The user can control the adaptation decision by adjusting these preferences. We argue that the user should be able to influence the adaptation decision but should not be confronted with too many details of the adaptation approach. In our opinion, adjustable preferences for influencing factors are a good compromise.

Salber and Dey [28]. Poladian et al. developed a utility-based framework for service selection and adaptation based on the awareness of resource demands and QoS capabilities of services [29], similar to our variation point and plan concepts. Sousa et al. presents an integrated framework for adaptation of context-aware applications that enables end-users to assemble their own collections of services at run time and to tune QoS policies of services to task-specific goals [30, 31]. At first glance these projects address very similar challenges as we do. However, in contrast to these projects our work mainly focuses on the bridging of syntactical and semantic differences of QoS-properties and context properties, which is required for the dynamic composition of independently developed components/services and context sensors at run-time.

Adaptive Service Grids (ASG) is an open initiative that enables the dynamic binding of services in adaptive service environments [11]. A basic concept of ASG is the semantic service request. It contains a description of the requested functionality, but it does not specify the concrete service that should be invoked. During the planning the platform tries to find a service that perfectly matches the semantic service request or to find a service (combination) that fits as much as possible. When an agreement with a particular service is achieved, a digital contract is set up and signed by both parties. If some services do not support negotiation mechanisms, the platform simply selects services based on their static properties. The approach is similar to ours, as it uses a semantic description of the desired functionality utilizing a domain ontology to discover services. However, in contrast to our approach the planning is not really QoS-driven. Therefore, support for QoS specification and the mediation of QoS-properties only play a secondary role in ASG.

Our approach has many similarities and shares a lot of concepts with the work done by Bleul et al. [6]. In particular, we follow nearly the same approach to the modeling of QoS dimensions, Service Level Requirements and Service Level Packages and to the integration of the resulting specifications into the OWL-S description of a service. Like ours, their work focuses on quality-aware service descriptions. In contrast to their approach, our main target is the integration of dynamically discovered services in a QoS-driven planning framework for adaptive applications on mobile devices and to align the modeling of QoS-properties with the modeling of context information and context properties.

WSML [18] and WSLA [9] are specification languages for SLA. Both languages allow specification of quality dimensions, metrics and guarantees. However, both approaches lack the usage of semantics. Therefore, an important facility to bridge the gap between different terms and semantic related metrics is missing.

Pure OWL-S [10] already provides support for specifying QoS. In OWL-S, QoS is specified as service parameters; but it does not really deal with the specification of QoS representations and metrics and lacks the specification of guarantees. Therefore it is not applicable to semantic and quality-aware service discovery. WSMO [17] also provides support for QoS specifications. However, it provides only indirect support for QoS as non-functional properties can be utilized for specifying quality dimensions and functional properties for specifying the relations between them.

DAML-QoS [22] is an ontology for QoS specification. It differentiates between QoS offers and QoS requirements but does not support the specification of service packages. It uses object-oriented identifiers to bridge among different terminologies and metrics in quality dimensions. These identifiers are predefined metrics. In our

approach basic transformation between different representations and metrics are defined in the underlying ontology in a very compact manner.

SWAPS [15] is a semantic approach to matching WS-Agreement descriptions for automatic partner selection. It uses semantic matchmaking but lacks the ability to transform metrics. Also it uses rules to describe a matching process. The authors of [4] describe another project, based on WS-A, which presents an extension of WS-A to specify negotiation terms. These terms are evaluated at runtime if certain guarantees are not satisfied. In contrast to these works, in our approach we do not try to exactly match Service Level Requirements and Service Level Offers. We make the QoS-properties of the service available to the planning framework, and the best suited service is determined by the utility function. The Service Level Requirements only act as a means for pre-filtering the discovered services.

8 Conclusions

We have presented a novel comprehensive modeling approach for the integration of service-based adaptation in a planning framework for compositional adaptation of context-aware applications. We have shown how semantic descriptions can be associated to variation points in the component framework. This enables the inclusion of dynamically discovered services into the adaptation planning process. A unique feature of our approach is the fact that it bridges between the adaptation decision parameters needed by the planning framework and the properties of the discovered services in terms of their service levels and execution context. Thus, our modeling framework provides a harmonized view on QoS-properties of external discoverable services and conventional context properties of component-based applications.

Our service ontology extends the OWL-S service descriptions with QoS-descriptions needed for the service discovery and the planning framework. Besides, we use another ontology as a base for the context model. Both ontologies are part of an overall ontology that enables a coherent modeling of QoS-properties and context properties. Our approach also covers requirements that arise from the mismatch of expected and provided information about QoS-properties. Based on these modeling concepts and the UML descriptions provided for the ports and interfaces of the component types, we can automatically generate service proxies for discovered services using transformation tools. A proxy acts as local representative of a service and thus, realizes the binding and the integration of a service into the component configuration. They also act as mediators that manage the QoS-properties between local components and the integrated services.

Basic tool support integrated into the Eclipse software development framework is available already for our new modeling approach. It has been used to generate prototype implementations of adaptive applications that demonstrate the viability of our approach. Nevertheless, more research has to be done to improve and further integrate the tool support and to help the developers with learning and using the new model-driven development approach. Furthermore, we need many more practical experiments in order to understand better the implications of our modeling and adaptation approach for the performance and scalability of the whole self-adaptive system. Last but not the least we need to study the ergonomics of self-adaptive systems in order to

design such systems in a way that meets the user's expectations in ubiquitous computing applications.

Acknowledgements

The contributions of the MUSIC project partners are gratefully acknowledged.

References

1. EU IST FP6 project MUSIC (Self-adapting Applications for Mobile Users in Ubiquitous Computing Environments), http://www.ist-music.eu
2. EU IST FP6 project MADAM (Mobility and Adaptation Enabling Middleware), http://www.ist-madam.org
3. Geihs, K., et al.: A Comprehensive Solution for Application-Level Adaptation. Software Practice & Experience. Wiley, Chichester (2008), http://dx.doi.org/10.1002/spe.900
4. Aiello, M., Frankova, G., Malfatti, D.: What's in an Agreement? An Analysis and an Extension of WS Agreement. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 424–436. Springer, Heidelberg (2005)
5. Bleul, S., Geihs, K.: Automatic Quality-Aware Service Discovery and Matching. In: 13th Annual Workshop of HPOpenView University Association (HP-OVUA), May 2006, pp. 109–118. Infonomics Consulting, Stuttgart (2006)
6. Bleul, S., Weise, T.: An Ontology for Quality-Aware Service Discovery. In: First International Workshop on Engineering Service Compositions (WESC 2005), IBM Report RC23821, December 2005, pp. 35–42 (2005)
7. Economic Classification Policy Committee. North American Industry Classification System (NAICS), http://www.census.gov/epcd/www/naics.html
8. DARPA. Profile-based Class Hierarchies, http://www.daml.org/services/owl-s/1.1/ProfileHierarchy.html
9. Dan, A., et al.: Web Service Level Agreement (WSLA) Language Specification, http://www.research.ibm.com/wsla/WSLASpecV1-20030128.pdf
10. Martin, D., et al.: OWL-S, OWL-based Web Service Ontology (2004)
11. EU IST FP6 project ASG (Adaptive Services Grid), http://asgplatform.org
12. Geihs, K., et al.: Modeling of Component-Based Self-Adapting Context-Aware Applications for Mobile Devices. In: IFIP Working Conf. on Software Engineering Techniques, Warsaw, Poland (2006)
13. University of Malta. OWL-S Editor to Semantically Annotate Web-Services, http://staff.um.edu.mt/cabe2/supervising/undergraduate/owlseditFYP/OwlSEdit.html
14. Oldham, N., et al.: Semantic WS-agreement partner selection. In: WWW 2006: Proceedings of 15th Intern. World Wide Web Conference, pp. 697–706. ACM, New York (2006)
15. Paolucci, M., et al.: Towards a Semantic Choreography of Web Services: From WSDL to DAML-S. In: Zhang, L.-J. (ed.) ICWS, pp. 22–26. CSREA Press (2003)
16. Reichle, R., et al.: A Comprehensive Context Modeling Framework for Pervasive Computing Systems. In: Meier, R., Terzis, S. (eds.) DAIS 2008. LNCS, vol. 5053, pp. 281–295. Springer, Heidelberg (2008)

17. Roman, D., et al.: WSMO - Web Service Modeling Ontology. In: DERI Working Draft 14. Digital Enterprise Research Institute (DERI) (2004)
18. Sahai, A., et al.: Towards Automated SLA Management for Web Services, HP Laboratories Palo Alto, HPL-2001-310 (R.1) (2001)
19. Frank, K., et al.: CoOL - A Context Ontology Language to enable Contextual Interoperability. In: Stefani, J.-B., Demeure, I., Hagimont, D. (eds.) DAIS 2003. LNCS, vol. 2893. Springer, Heidelberg (2003)
20. Rouvoy, R., Barone, P., Ding, Y., Eliassen, F., Hallsteinsen, S., Lorenzo, J., Mamelli, A., Scholz, U.: MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments. In: Cheng, B.H.C., et al. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525. Springer, Heidelberg (2009)
21. Carnegie-Mellon University, WSDL-to-OWLS, http://www.daml.ri.cmu.edu/wsdl2owls/
22. Zhou, C., Chia, L.-T., Lee, B.-S.: Semantics in Service Discovery and QoS Measurement. IT Professional 7(2), 29–34 (2005)
23. WSDL2OWL-S, http://www.daml.ri.cmu.edu/wsdl2owls/
24. OWL-S Editor to Semantically Annotate Web-Services, http://staff.um.edu.mt/cabe2/supervising/undergraduate/owlseditFYP/OwlSEdit.html
25. Wang, X.H., et al.: Ontology Based Context Modeling and Reasoning using OWL. In: Proceedings of Workshop on Context Modeling and Reasoning (CoMoRea 2004), Orlando, Florida USA (March 2004)
26. Gu, T., et al.: A Middleware for Building Context-Aware Mobile Services. In: Proc. of the IEEE 59th Vehicular Technology Conference (VTC 2004 spring), Milan, Italy (May 2004)
27. WebServices Agreement Specification (WS-Agreement), http://www.ogf.org/documents/GFD.107.pdf
28. Salber, D., Dey, A.K., Abowd, G.D.: The context toolkit: aiding the development of context-enabled applications. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: the CHI Is the Limit, Pittsburgh, Pennsylvania, United States. ACM, New York (1999)
29. Poladian, V., Sousa, J.P., Garlan, D., Shaw, M.: Dynamic Configuration of Resource-Aware Services. In: 26th International Conference on Software Engineering, pp. 604–613. IEEE Computer Society, Edinburgh (2004)
30. Sousa, J.P., Schmerl, B., Steenkiste, P., Garlan, D.: Activity-oriented Computing. In: Mostéfaoui, S., Maamar, Z., Giaglis, G. (eds.) Advances in Ubiquitous Computing: Future Paradigms and Directions, pp. 280–315. IGI Publishing, PA (2008)
31. Sousa, J.P., Balan, R.K., Poladian, V., Garlan, D., Satyanarayanan, M.: User Guidance of Resource-Adaptive Systems. In: Intl. Conf. on Software and Data Technologies, pp. 36–44. INSTICC Press, Porto (2008)

MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments

Romain Rouvoy[1], Paolo Barone[2], Yun Ding[3], Frank Eliassen[1], Svein Hallsteinsen[4], Jorge Lorenzo[5], Alessandro Mamelli[2], and Ulrich Scholz[3]

[1] University of Oslo, 0316 Oslo, Norway
rouvoy@ifi.uio.no, frank@ifi.uio.no
[2] HP Italy, 20063 Cernusco sul Naviglio, Italy
paolo.barone@hp.com, alessandro.mamelli@hp.com
[3] European Media Laboratory GmbH, 69118 Heidelberg, Germany
yun.ding@eml-d.villa-bosch.de,
ulrich.scholz@eml-d.villa-bosch.de
[4] SINTEF ICT, 7024 Trondheim, Norway
svein.hallsteinsen@sintef.no
[5] Telefónica I+D, 47151 Valladolid, Spain
jorgelg@tid.es

Abstract. Self-adaptive component-based architectures facilitate the building of systems capable of dynamically adapting to varying execution context. Such a dynamic adaptation is particularly relevant in the domain of ubiquitous computing, where numerous and unexpected changes of the execution context prevail. In this paper, we introduce an extension of the MUSIC component-based planning framework that optimizes the overall utility of applications when such changes occur. In particular, we focus on changes in the service provider landscape in order to plug in interchangeably components and services providing the functionalities defined by the component framework. The dynamic adaptations are operated automatically for optimizing the application utility in a given execution context. Our resulting planning framework is described and validated on a motivating scenario of the MUSIC project.

Keywords: Adaptation planning, component-based architectures, self-adaptation, service-oriented architectures.

1 Introduction

With the emergence of ubiquitous computing, common future scenarios will consist in people moving around carrying mobile devices, which they use extensively to assist both leisure and business related tasks. This will not only involve interactions with services provided through the Internet, but also with services directly provided by devices available in the surrounding environment.

For developers of mobile applications this is a very challenging scenario. Users' movements in ubiquitous computing environments cause frequent and unexpected changes in the execution context of their applications. For example, a mobile device is frequently roaming, and its applications have to be dynamically adapted to remain

B.H.C. Cheng et al. (Eds.): Self-Adaptive Systems, LNCS 5525, pp. 164–182, 2009.

useful under new network conditions. Such an adaptation requires the detection of context changes, but also the selection of an application configuration that maintains a satisfactory *Quality of Service* (QoS) in the new context. Furthermore, when services become a part of the ubiquitous environment, both the availability and the quality of the services on which the applications depend becomes a concern of the application developer. There is therefore a need to dynamically discover services both when they become available and when they disappear. Also, such applications need to embed logic enabling them to reason about how and when to use a service available in the surrounding, to select among service alternatives when there are more than one available, and to adapt when a service disappears. Such a self-adaptation process is generally complex and costly to implement. To achieve self-adaptation, developers can use programming language features, such as conditional expressions, parameterization, and exceptions. However, these approaches introduce complexity by intertwining adaptation and application logic. Also, they make software evolution difficult. Conversely, approaches that use application independent middleware approaches for adaptation relieve the applications from adaptation concerns [1].

In the MUSIC project, we follow the latter approach by seeking to separate the self-adaptation concern from the business logic concern and delegate as much as possible of the added complexity related to self-adaptation to generic middleware. The adaptation process relies on the architecture model of the application, which specifies its adaptation capabilities and its dependencies to context available at runtime. In MUSIC, an application is modeled as a component framework, which defines the functionalities that can be dynamically configured with conforming component implementations. Thus, the purpose of an adaptation-planning framework is to evaluate the utility of alternative configurations in response to context changes, to select a feasible one (*e.g.*, the one with highest utility) for the current context and to adapt the application accordingly.

In this chapter, we propose a comprehensive extension of the MUSIC platform and planning framework we initially sketched in [2]. Currently, MUSIC only supports the adaptation of component-based architectures. The proposed extension enables the self-adaptation of mobile and ubiquitous applications in the presence of *Service-Oriented Architectures* (SOA). The planning middleware evaluates discovered remote services as alternative configurations for the functionalities required by an application. This means that the extended planning framework can support seamless configuration of component frameworks based on both local and remote components as well as services. In particular, components and services can be plugged in interchangeably to provide the functionalities defined by the component framework. In case of services, the planning framework deals directly with *Service Level Agreement* (SLA) protocols supported by the service providers. In addition to that, we introduce in this chapter a support for advertising services and associated service levels, in order to satisfy dynamically incoming service requests. Hence, MUSIC applications can use the MUSIC platform to share services with the environment.

In the remainder of this chapter, we first describe in section 2 the MUSIC approach to planning-based adaptation for component-based applications. In section 3, we introduce a motivating scenario for the support of SOA for self-adaptive applications in a ubiquitous environment, as well as derive a set of requirements. Section 4 exposes the MUSIC support for consuming and providing services in ubiquitous environments. Section 5 describes the integration of SOA into the MUSIC platform from an

implementation perspective, while section 6 provides a preliminary validation of our approach by discussing how the requirements derived in section 3 are met by the proposed design. In section 7, we discuss related work before concluding and pointing out further work in section 8.

2 The MUSIC Approach to Self-Adaptation

Planning-based adaptation of a component-based application refers to the capability of a system to adapt to changing user needs and operating conditions by exploiting knowledge about its composition and *Quality of Service* (QoS) characteristics of its constituting components [2,3,4,5,6]. In MUSIC this knowledge is provided in the form of a *QoS-aware model* (cf. Figure 1), which describes the abstract composition, the relevant QoS dimensions and how they are affected when varying the actual component configuration. This model is exploited by the adaptation middleware to select, connect, and deploy a configuration of Component Realizations providing the *best utility*. The utility measures the degree of fulfillment of user preferences while optimizing device resource utilization [1,3]. The model describes the abstract composition as a set of Roles collaborating through Ports, which represent either functionality provided to or required from collaborating components. Properties and property predictor functions associated with the ports define how the QoS properties and resource needs of components are influenced by the QoS properties of the components they depend on. A port has a Type defining the functionality represented by the port in terms of interfaces and protocol. Component realizations implement ports and a component realization can be used in a role if the ports match (same type). Component realizations are Atomic or Composite. A Composite Realization is itself an abstract composition and allows for recursive decomposition. Constraints are predicates over the properties of the constituting components of a composition, which restrict the possible combinations of component realizations (*e.g.*, configuration consistencies) [3,7].

The model is represented at runtime as *plans* within the middleware. A plan reflects a component realization and describes its ports and associated property

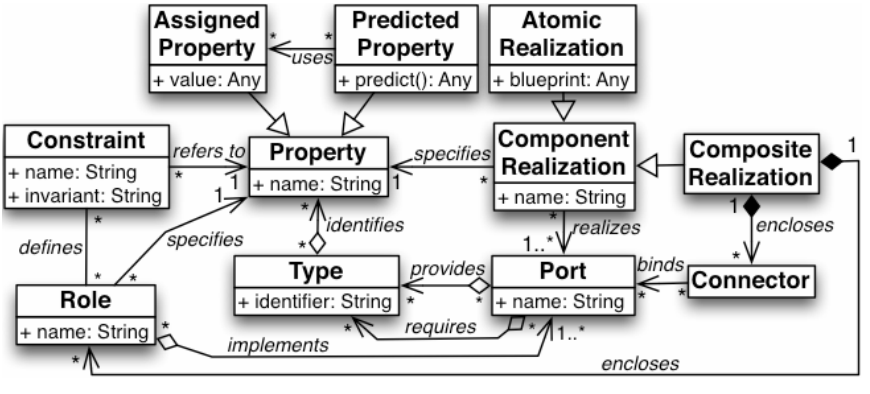


Fig. 1. Description of the MUSIC meta-model

predictors as well as *implicit dependencies* on the hosting platform (*e.g.*, platform type and version). In the case of an atomic component realization, it also contains a reference to the class, which realizes the component. In the case of a composite realization, the plan describes the internal structure in terms of roles and ports and the connections between them. Variation is obtained by describing a set of possible alternative realizations of the roles.

Then, *planning* refers to the process of selecting the components that make up an application configuration providing the best possible utility to the end-user. This process will be triggered at start-up of the application and at run-time when the execution context suddenly changes. When such an adaptation process is triggered for a particular type, the planning middleware iterates over the plans associated to the roles. For each plan, it resolves the plan dependencies and evaluates the configuration suitability to the current execution context by computing the Predicted Properties. The predicted properties are input to the normalized utility function that computes the expected utility of the evaluated application configuration [1,2,3,4,5]. The utility function of an application is provided by the developer and is typically expressed as a weighted sum of dimensional utility functions where the weights express user preferences (*i.e.*, relative importance of a dimension to the user). A dimensional utility function measures user satisfaction in one property dimension.

An example model for an application assisting traveling on public transportation is shown in figure 2. It is described as a collaboration of five roles. GUI presents a graphical user interface on the device. Main embeds the application logic and binds the different functionalities together. Main interacts with Route to find the shortest route and the estimated travel time. It also uses Map to get localized maps and Location to get the current location. The QoS properties used in the model are specified in table 1. Property predictors for the application, specified as functions of the properties of the components it consists of, are associated with the composition in figure 2. The utility function assumes that the user always prefers high accuracy and low battery consumption, while the relative weighting (w_acc, w_bat) will be extracted from the user profile by the middleware.

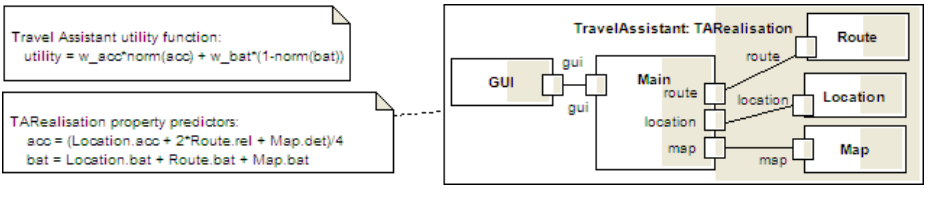


Fig. 2. Example model for a TravelAssistant application

Table 1. Relevant QoS properties for the TravelAssistant application

Property	Description	Value range
acc	Accuracy	1-10
det	Level of detail of map	1-10
rel	Reliability of estimated travel time	1-10
bat	Power consumption of a component or link	0-∞

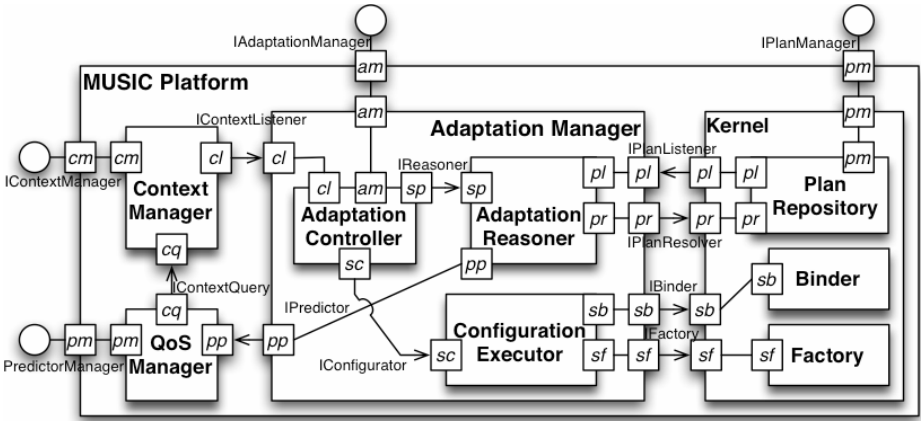


Fig. 3. Architecture of the MUSIC platform

The middleware manages a collection of active applications and seeks to maximize the overall utility, which is computed as a weighted sum of individual application utilities. The weights in this case express application priorities of the user.

Figure 3 depicts the component-based architecture of the MUSIC platform. The planning is typically triggered by context changes detected by the Context Manager. The Adaptation Controller coordinates the adaptation process. The Adaptation Reasoner supports the execution of the planning heuristics, which is driven by metadata included in the plans [4]. The Plan Repository provides an interface *IPlanResolver* to the adaptation reasoner allowing for the recursive retrieval of plans associated to a given port. Any additional metadata on the required types will help the plan repository to exclude plans and thus drastically reduce the exploration space [4,6]. The adaptation reasoner builds a valid application configuration and discards those whose dependencies are unresolved. Then, the heuristics ranks the application configurations by evaluating their utility based on the computation of the predicted properties, whose values are retrieved from the QoS Manager.

The *reconfiguration process* is handled by the Configuration Executor, which uses the set of plans selected by the planner to reconfigure the application. This requires the collaboration of the components, which must implement a reconfiguration interface allowing the middleware to bring them to a state where they can be safely replaced and transfer their state to an alternative component.

3 Challenges of Ubiquitous and Service-Oriented Environments

The term *service* is perhaps one of the most over-used and confusing terms in the software industry as analyzed in [9]. Typically, services are defined as *functionalities* or capabilities provided by a software system to other software systems or to a human user [10]. In the context of SOA, services are provided by independent service providers, which instantiate the providing software on their computers and advertise the services they provide using standardized mechanisms, such that they can be discovered and bound dynamically by consumers which need them. A fundamental concept

of service-orientation is the standardized *service contract* [11], which is used to express the service semantics and capabilities. Service QoS properties are normally negotiated between the service provider and the *service consumer*, and are described as part of the service contract as a *Service Level Agreement* (SLA). A *service level* is used to describe the expected performance (*e.g.*, response time and availability) and properties such as billing, termination terms, and penalties in case of a violation of the SLA [12]. A SLA can either be created after selecting a fixed service level offer among several pre-defined offers or, in more complex cases, after a customization via a negotiation process. An SLA may be valid for a limited period or may be terminated explicitly. During *SLA provisioning*, the provider monitors the service QoS and adapts its resources to avoid *SLA violations*. The consumer may also perform monitoring to avoid trusting the provider blindly.

The platform presented in the previous section focuses on component based self-adapting systems. When mobile devices move around in ubiquitous computing environments they experience a dynamic service landscape and additional requirements to self-adaptation arise which require extensions of the platform. To investigate these issues, we consider the following scenario of Paul who is on his way to meet a friend, assisted by applications on his mobile device. First, we introduce several situations that Paul encounters and explain how he and his device react. Then, we explain the requirements that enable such flexibility.

3.1 Example Scenario: Paul on His Way to Meet a Friend

Paul has been at a concert in Paris. Now, he is taking the subway to a friend to see her new home and to tell her about the show. His MUSIC-enabled mobile device is WiFi-, UPnP-, and GPS-enabled. It provides several applications, among them a service-based version of the *TravelAssistant* from the example and a media-sharing application.

The *TravelAssistant* assists Paul with route planning, ticket vending, detects traveling delays, and notifies Paul if he is affected by such delays. The media-sharing platform, called *InstantSocial* [8], appears as a web site. However, instead of relying on a central Internet server, it is served by a composition of services scattered across nearby devices. As more users participate, this platform becomes more robust, the number of shared content items increases and it may become more attractive for the users. As soon as a critical mass of users leaves, it stops operating.

Scene 1. The scenario starts with Paul entering the Paris subway. He wants to plan the journey to his friend, which requires a route service for the subway as well as a location and a map service for the remaining trip. RATP, the operating company, offers a route service for public transportation and a map service of Paris at two QoS levels: *basic* and *premium* quality. Via UMTS, there is also access to a commercial service of high quality, though for a higher monetary cost. Paul requests services of high quality and his device chooses the cheap premium service of RATP.

Scene 2. With his *TravelAssistant*, Paul devises his journey and buys a ticket. As regular traveler, he has an electronic pass. Upon approaching a validation post, his device detects it, Paul's pass is checked, and the entrance gate opens automatically.

Scene 3. Inside the train, Paul thinks of searching for further pictures of the concert. He starts the *InstantSocial* application, which configures itself according to the other *InstantSocial* instances in the vicinity. His device notifies Paul about the presence of a matching media-sharing group. He joins and a moment later his display shows a selection of pictures, each representing a collection of shots of interests. He browses through the content, selects the ones he likes, and begins to download.

Scene 4. During the trip, there is an incident in the metro, blocking the planned itinerary. The travel assistant notifies Paul and proposes an alternative metro route with a different final station. Unfortunately, planning the remaining trip is not as smooth as desired: RATP reserves a large share of its bandwidth to guide the emergency personnel and declines to offer the high-quality map service. Furthermore, he cannot use GPS because the system's satellites are out of sight. As best solution, Paul's device chooses the external high-quality services, despite the higher cost.

Scene 5. Now, Paul is in a train with fewer visitors of his concert. Due to the decreased robustness, *InstantSocial* adapts it focus from *sharing* to *collecting* pictures. The other instances tend to do the same such that the combined media platform weakens. Finally, Paul is notified about the poor quality and he terminates *InstantSocial*.

Scene 6. After leaving the subway, the GPS module starts working and his device guides him through the streets. Some minutes later, he arrives at his friend's home in time with a device full of impressions to share.

3.2 Requirements for Planning-Based Adaptation

During Paul's journey, the applications on his device make flexible use of a variety of services and protocols nonetheless remains operational through various context changes. In particular, *TravelAssistant* and *InstantSocial* depend on external services that are dynamically chosen and used. Each *InstantSocial* instance also offers services to other instances. All these examples of flexibility require middleware support, which is provided by the design presented in this chapter.

Scene 1 shows a service selection process depending on QoS. The use of an UPnP-based service in scene 2 demonstrates the need for alternative connection protocols and services. Scene 3 demonstrates the degree of flexibility required: an *InstantSocial* instance is a combination of local and external services; it is able to offer and may use services at different QoS levels. The actual composition of the instance at a specific time has to be decided at runtime. Scene 4 features a willful reduction of a QoS level by the provider of an external service. It results in an adaptation to an alternative service provider, although the original provider is still offering the service, too. In scene 5, the device has to cope with an unplanned service termination by the sudden disappearance of *InstantSocial* instances. Furthermore, it demonstrates the deliberate termination of services by the user. Thus, to support scenarios of the kind presented above, we need to extend the platform to deal with the following SOA requirements:

- Dynamic discovery of services,
- Dynamic binding and change of binding to service providers,
- Negotiation of service level agreements and detection of violations,
- Hosting and publishing of services.

4 Supporting Service-Oriented Architectures within MUSIC

The interpretation of the term *service* presented in the previous section relates naturally to the port concept in the conceptual model presented in section 2. Thus we can accommodate services in the conceptual model simply by considering that ports represent services provided by or required by components, that services are described by types, and that service levels are described by properties. However, the middleware must be extended in several ways to cope with the challenges derived above. The remaining of this section introduces the *consumer-* and the *provider-side* support offered by the MUSIC platform in order to enable the seamless integration of services made available in a ubiquitous computing environment.

4.1 Consuming Services within MUSIC

In SOA-based computing environments, an application typically uses one or more services, which possibly depend on further services and so on. Thus, a large number of computers owned and administrated by different organizations may potentially be involved. This problem is aggravated when we deal with several applications running concurrently. Thus, optimizing utility over the entire set of involved computers is likely to be intractable both from a technical and administrative point of view. Therefore, we have to delineate the scope of an adaptation to be more tractable. To this end, we introduce the notion of *adaptation domain* and the distinction between *internal* and *external* services.

An *adaptation domain* is a collection of MUSIC platform instances controlled by one *adaptation manager*. It includes one distinguished node (*e.g.*, a handheld device), which represents a permanent binding to a user. This node acts as the *nucleus* around which the adaptation domain forms dynamically as *auxiliary* nodes come and go. The movement of nucleus nodes or changes in connectivity due other phenomena causes the dynamic evolution of an adaptation domain. Adaptation domains may overlap in the sense that auxiliary nodes may be members of multiple adaptation domains. This adds to the dynamics and increases the complexity because the amount of resources the auxiliary nodes are willing to provide to a particular domain may vary depending on the needs of other served domains. The user of a nucleus node may start and stop applications or shared components, and the set of running components is adapted by the adaptation manager according to these user actions and context changes, taking into account the resource constraints.

Clearly, it makes a difference whether a role is bound by instantiating a component implementation running in the adaptation domain where a system is built (*private instance*), by using a service provided by a component instance already running there (*internal service*), or by connecting to a service provided by a third party (*external service*). In the first two cases, the adaptation manager building the system must provision the resources and has control of the provided service level. In the latter case, the service level is outside the control of the adaptation manager, and it is necessary to negotiate an SLA with the service providers in order to compare the suitability of services by different providers and weight against deploying an internal service. External services may be provided by other adaptation domains or by third party providers (also referred to as *external non-MUSIC services*).

Discovery of Services and Service Levels. Providers make their services accessible to third parties according to specific *discovery protocols*. The MUSIC platform supports an extensible set of discovery protocols allowing the detection of services available in the service landscape. The discovery of a service triggers the retrieval of its *service description*, which includes information on the service capabilities, semantics, and possibly the offered service level(s) or QoS properties in form of an *agreement template*. The service description and, if available, the related agreement template are then converted to service plans, each one reflecting an alternative realization for the service level.

Negotiation of Service Level Agreements. The planning phase involves the evaluation of the available plans, for selecting the composition optimizing the utility of the applications running on the device. The utility depends on the QoS properties predicted by the services, whose value can be *static* or *dynamic*. Static properties consist of fixed values that do not change over the time. Dynamic property values can change according to the current status of the service. Evaluating the actual QoS values for such properties requires a process of negotiation with the service provider. The current MUSIC negotiation protocol is inspired by the WS-Agreement specification [13] (for both the definition and the creation/monitoring of SLAs), where the provider enriches the service description with an agreement template and the consumer fills in the template to create and submit an *agreement offer*. The offer creation is driven by *Service Level Objectives* (SLO), which are conditions defined at application or configuration level and act as pre-defined criteria for negotiating an SLA contract. Once the provider has accepted the offer, the agreed property values are reflected in the plan.

Provisioning of Service Level Agreements. Whenever a service available in the landscape is selected for use as a result of the adaptation reasoning, the MUSIC platform instantiates *service proxies.* These Proxies act as local representatives of the remote services and encapsulate the communication protocol necessary to access them in a location-transparent way. They are created by a *binding framework*, which provides dedicated proxy factories. Each factory supports a particular communication protocol to export or import a service. During the binding phase, the SLA contract associated with the selected plan is provisioned and enforced by the involved parties, which includes the reservation of computing resources and the deployment of SLA monitoring facilities [11,15,16].

Monitoring of Service Level Agreements. For the purpose of SLA monitoring, the service proxy is instrumented with appropriate monitoring mechanisms according to the content of the SLA contract (*e.g.*, response delay, result quality). Both parties are responsible for checking the status of the *agreement* and for taking proper actions in case of violation of the agreement. Thus, after the creation of an *agreement,* the MUSIC middleware, at any given time, must be able to check the current state of the agreement itself. When an agreement is not fulfilled anymore, the MUSIC middleware must terminate it and trigger a new adaptation process in order to detect a new set of available services and to select among them the best candidate to replace the one breaking the contract. SLA-enabled service providers handle the state model of an agreement and of its constituting terms, and make them accessible to consumers in form of readable properties of the agreement.

On the consumer side, the MUSIC middleware architecture is responsible for checking the state of an agreement according to pre-defined policies (*e.g.*, at given intervals or when detecting that the expected performance of a service is degrading). By querying the service provider for the agreement state, it is possible to detect whether the agreement has been violated or not. In case of violation, the consumer terminates explicitly the agreement by invoking a *terminate* operation on the *provider side* (since there might be costs associated to the usage of the service), and discards the related service plan, hence triggering a new adaptation process.

4.2 Providing Services within MUSIC

Hosting both applications and components providing services to the outside world in an adaptation domain complicates the adaptation reasoning. In addition to the user owning the device, there are also external service consumers, which may have conflicting needs (expressed in the SLA). Fortunately, the utility function approach lends itself quite naturally to cope with such situations. Our solution is to treat shared components providing services to external clients in the same way as applications and equip them with their own utility function, computing the degree of fulfillment of active SLAs. Using the weights, the overall utility function balances the utility to the owner of the device against the utility to service clients. This information about user preferences is included in user profiles.

Another difficulty is related to property prediction. For shared services, the resources needed by the component to guarantee a certain QoS often depend on the number of consumers. Hence, property predictor functions for shared services must take this into account.

Publishing of Services and Service Levels. By publishing its description using the discovery protocols supported by the MUSIC platform, a service running on a node can be made available to other nodes within the adaptation domain. Each service description encloses the service type as well as an *agreement template* describing the static QoS properties that are provided by this service. QoS dimensions referring to dynamic properties of the application are unbound in order to be fixed at a later time depending on the capabilities and the processing load of the hosting node.

Negotiation of Service Level Agreements. The MUSIC platform supports the negotiation of agreements by playing the role of a service provider. Whenever a service consumer selects one of the published services, the MUSIC platform receives an *agreement offer* for consuming this service. The MUSIC platform applies the negotiation heuristics to decide whether to accept or reject this offer by taking the current resource availability into account. This heuristics predicts the impact of accepting the offer with regards to agreements that have been already accepted. If the resulting impact does not trigger any violation of previous agreements, the MUSIC platform creates an *agreement*, which keeps track of the negotiation process.

Provisioning of Service Level Agreements. When a service consumer requests an internal service, the MUSIC platform checks that the requested service refers to an accepted agreement. Then, the *binding framework* instantiates a service skeleton—*i.e.*, a local representative of the service consumer—which reflects the ongoing agreement and implements one of the supported communication protocols

(*e.g.*, SOAP or RMI). Invocations received via the service skeleton are delegated to the service instance locally deployed on the node.

Monitoring of Service Level Agreements. Depending on the negotiated properties agreed in the agreement, the service skeleton is instrumented with context sensors, which are responsible for monitoring the agreement. The MUSIC platform provides a library of sensors for observable properties (*e.g.*, invocation latency) as part of its context middleware. If one of the sensors detects a violation in one of the dimensions of the agreement, it notifies the MUSIC platform about this violation, which results in the notification of the service consumer and the termination of the agreement.

5 Realizing the Support for Service-Oriented Architectures

This section describes the extension of the MUSIC platform in order to support the SOA principles as well as the realization of the MUSIC reference implementation.

5.1 Architecture of the Service-Oriented MUSIC Platform

To support the above-mentioned SOA principles [11], we have integrated new components into the MUSIC Platform (cf. Figure 4, the composite component SOA Support). As MUSIC is independent of a particular technology, various implementations of these components can be developed (*e.g.*, Web Service, CORBA, RMI, or UPnP).

More specifically, the Service Discovery is responsible for publishing and discovering services using different discovery protocols. The Remoting Service is responsible for the exporting of services at the service provider side, and for the binding to these services at the service consumer side. Whenever a service is exported, it is enabled to accept requests from (remote) service consumers. Each *service description* defining the provided functionalities and containing the necessary information for the consumer to access the service[1] can be published by the service discovery. If the service provider offers additional guarantees for the published services, *agreement templates* are published in addition to the *service description*.

The service discovery supports the dynamic registration of *discovery listeners*. A *discovery listener* can have interest for particular services and can enforce customized policies to handle them. For example, the *Remote Platform Discovery Listener* is particularly interested in finding remote instances of the MUSIC platform in order to provide information about the MUSIC platforms connected to the applications. The *SLA Discovery Listener* is interested in finding services accompanied with an SLA support. Upon the discovery of services, the service discovery notifies the registered *discovery listeners* by passing them the service descriptions. Since plans are the base for the Adaptation Manager to perform planning-based adaptation, the *discovery listeners* create service plans based on the *service descriptions* and the *agreements* negotiated by the SLA Negotiation. Plans for remote services are generated whenever

[1] For example, the service consumer needs the remote service URL in order to access it. In case of a RMI-based binding, this URL would be `rmi://localhost:8080/EchoService`. While, in case of a Web Service, the URL is the location of the WSDL document, *e.g.,* `http://localhost:8090/axis2/services/EchoService?wsdl` for the *Echo Service*.

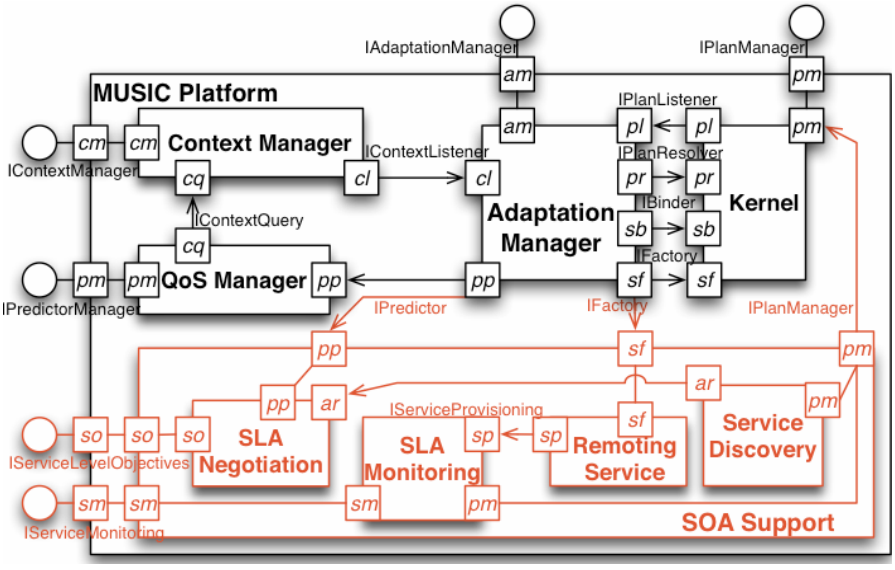


Fig. 4. SOA configuration of the MUSIC platform

services are discovered; hence plans are available when the adaptation manager triggers an adaptation at a later time. Plans are automatically discarded and removed from the Plan Repository whenever remote services disappear or for some reason become unavailable to the middleware.

The distributed instances of the MUSIC platform form a federation such that the service discovery on different platforms can interact with each other. Hence, MUSIC platform A can be aware of a service, which is published using a protocol supported by MUSIC platform B and not supported by A. If the remoting service on platform A supports the appropriate communication protocol, A is able to bind to that service which it would not able to discover alone.

Agreement templates can be either static or allow for dynamic negotiation [12]. Furthermore, a service may be offered at a pre-defined set of service levels. When the service discovery detects such a service, it first generates an abstract service plan enclosing structural and behavioral metadata related to the service. Then, in order to reflect the alternative service levels the service discovery publishes an extended version of the service plan for each service level into the plan repository. Such a service level plan inherits the metadata of the service from the abstract service plan and extends it with the additional QoS properties described by the particular service level (*e.g.*, service accuracy and cost).

The adaptation manager is then able to compare each available service level when applying the reasoning heuristics. Since service negotiation is a time critical factor for an efficient planning process, it should be resolved as soon as possible. In MUSIC, the negotiation is generally performed during service discovery for static QoS properties (*e.g.*, service cost) described by the service levels. The resulting static QoS property values are included into the service plan such that the predicted properties can

automatically report them at a later time. However, in presence of a flexible service level [11,13], the negotiation becomes dynamic, meaning that the SLA is negotiated during the planning process. Dynamic negotiation is particularly required when the adaptation manager needs to reason about up-to-date QoS properties (*e.g.*, current service accuracy). In this case, the predicted properties, when evaluated by the reasoning heuristics, delegate the negotiation of the requested property to the SLA negotiation. The negotiation protocol is driven by SLOs, which are pre-defined criteria for negotiating SLA [15].

The Configuration Executor generally iterates over the plans composing the new configuration in order to reconfigure the application. As described in section 2, the configuration executor distinguishes between plans which refer to available services and plans which refer to services that are not available yet. In order to benefit from remote services, the configuration executor now faces a third case: If the plan refers to a remote service available in the environment, the configuration executor uses the Remoting Service to generate a specific component that will act as a *service proxy*. A service proxy is a local representative of the remote service. In particular, it implements the service type described by the application components and encapsulates the communication necessary to access the remote service. By invoking the service proxy, a service consumer interacts with the remote service in a location-transparent way—*i.e.*, as if the remote service is a local one.

The remoting service supports the dynamic integration of binding frameworks. During the binding phase, the SLA associated with the selected plan is provisioned and enforced by the involved parties. For the purpose of monitoring, the service proxy is instrumented with appropriate monitoring mechanisms by the component SLA Monitoring according to the content of the SLA (*e.g.*, response delay, result quality). The SLA monitoring is responsible for checking the status of the agreement for taking proper actions in case of its violation.

As an example of performing SLA monitoring in ubiquitous environments, the service proxy implements a disconnection detection algorithm. This disconnection support is inspired by the principles of *ambient programming* [17]. When loosing the connection to a remote service, the proxy stores the incoming service requests in a queue and returns a non-blocking *future object* to the application. The future object includes actions that are triggered whenever the connection is resolved to process the result of the request. If the connection is lost for a long period, the service proxy terminates the agreement via the component SLA negotiation. Subsequently, the SLA monitoring removes the associated service level plan from the plan repository to trigger an adaptation of the application. During the reconfiguration process, the request queue is transferred to the new component (or service proxy) that will be selected and deployed by the middleware.

5.2 Implementation of the Service-Oriented MUSIC Platform

The reference implementation of the MUSIC platform is based on the architecture described in section 5.1. The selection of the framework, which the reference implementation of the MUSIC architecture is built upon, has been made to meet the most

relevant requirements for the MUSIC platform. They are, in particular, open source framework, multi-platform support, suitability for resource-constrained devices, and SOA support. Therefore, we selected OSGi to leverage the MUSIC platform.

OSGi (`http://www.osgi.org`) defined itself as the dynamic module system for Java, is a service-oriented component-based framework. The success of OSGi may be attributed to its relative simplicity, efficiency, openness, and portability. Multiple open source implementations of OSGi are available. Since its initial design, OSGi targets resource-constrained devices. Some existing implementations, such as *Concierge* [18], exhibit a reasonable memory footprint for resource-constraint devices (80 kB). Furthermore, some initiatives, such as the *OSGi Mobile Specification (JSR-232)* [19], the *Eclipse eRCP project* [20] or the *Sprint Titan platform* [21], propose OSGi for hosting applications in mobile devices. OSGi offers a class-loading mechanism to dynamically load/unload modules (bundles in the OSGi terminology). This feature is particularly interesting to support the plug-ability of the MUSIC architecture. Plug-ability is required to tackle the heterogeneity in communication and service discovery technologies. It also allows the integration of an extensible set of customized context sensors and adaptation algorithms.

SOA is the cornerstone of both OSGi and MUSIC. The SOA implementation in OSGi is simple and efficient, based on fast Java method invocations and a service registry, which provides mechanisms to react on the appearance and disappearance of services (essential in mobile environments). However, OSGi lacks of distribution support because OSGi services only communicate within one Java VM. The Service Discovery and the Remoting Service jointly extend OSGi with transparent distribution support and provide an abstraction to dynamically incorporate realizations of different discovery and communication protocols.

The Service Discovery delegates requests for publishing and discovery to protocol-specific implementations of service discovery, which are plugged into the platform as OSGi services implementing the interface *IServiceDiscoveryFactory*. Currently, the *Service Location Protocol* (SLP) protocol based on *jSLP* [22] and the *Universal Plug and Play* (UPnP) protocol based on the *UPnP bundle of DomoWare* [23] are supported by the MUSIC platform.

The Remoting Service supports plug-ability in a similar way. Each protocol-specific realization implements the interface *IExportFactory* or *IRemoteBindingFactory,* and is registered as a service to the service registry. Currently, the remoting service has support for exporting and binding services using sockets messaging and UPnP. The Web Service support is under development by a lightweight SOAP engine and small footprint HTTP server [24]. We create dynamic service proxies with the code generation library *CGLIB* [25], and attach communication protocol-specific interceptors to the service proxies. The instrumentation of a service proxy with SLA monitoring mechanisms will be realized by adding monitoring interceptors to the proxy (*e.g.*, to measure the response time).

MUSIC has chosen a set of lightweight frameworks and protocols to offer the best balance between performance in mobile devices and application requirements. The preliminary implementation of the *TravelAssistant* has demonstrated the good behavior of the MUSIC platform in a handheld device.

6 Discussions

As a preliminary validation of our approach, in this section we present a walk-through of how the middleware would behave in the scenario described in section 3.1. Table 2 presents the realizations available for the different services with property predictors for the relevant properties. In addition to the properties defined in table 1, we also introduced cost, which is very relevant for 3rd party services, and extended the utility function as follows: *utility=0.6*norm(acc) + 0.1*(1-norm(bat))+ 0.3*(1-norm(cost))*. Based on this extended model we computed the utilities of the various configurations in different scenes. Table 3 shows the utility of the best configurations in different situations during the scenario.

In the first three scenes of the scenario, the composition *i)* using the RATP Location, Map, and high quality Route services predicts the highest utility and is therefore chosen. In scene 4, the high quality RATP Map service breaks its SLA. The service proxy observes this and notifies the component SLA Monitoring, which terminates the agreement and triggers a re-planning. The Adaptation Manager predicts that using the commercial Map service instead now yields the highest utility and asks the Configuration Executor to reconfigure the application's service binding. This includes generating a corresponding service proxy. In scene 6, the device's GPS discovers the satellites and publishes the associated service plans into the Plan Repository. As this service provided by a local component is free and accurate, the adaptation manager predicts its use to have the highest utility and reconfigures accordingly.

Table 2. Services defined in the TravelAssistant application

Service	Description	Provider	Level	Property predictors
Location	Locates the device geographically	RATP		cost=0, acc=5, bat=1
		Local component using the builtin GPS		cost=0, acc=7 if GPS signal, 0 otherwise, bat=3
Map	Provides a map of a limited area	RATP	basic	cost=0, det=1, bat=2
		RATP	detailed	cost=5, det=9, bat=4
		3rd party		cost=9, det=9, bat=4
Route	Computes best route and estimated travel time	RATP	basic	cost=0, rel=1, bat=1
		RATP	reliable	cost=5, rel=7, bat=1

Table 3. Some alternative configurations and utilities of the TravelAssistant

Configuration			*Utility*		
Location	Map	Route	Scene 1	Scene 4	Scene6
RATP	RATP detailed	RATP reliable	**0,64**	-	-
RATP	3rd party	RATP reliable	0,56	**0,56**	0,56
builtinGPS	3rd party	RATP reliable	-	-	**0,58**

The *InstantSocial* application appears to the user as a centralized application, while under the hood, each user runs its own IS instance in its own adaptation domain. The multi-user behavior emerges from the interactions among the IS instances services—*i.e.*, each IS instance offers services and uses services offered by the others. The utility function determines the composition and behavior of an individual instance depending on the local resource situation and the QoS of the used services, and therefore indirectly also on the composition and resource situation of the other instances. Thus, the user-visible shape of *InstantSocial* appears according to size and quality of the instances in the collection.

The composition of IS describes three roles: *browser proxy* (BP), *presentation* (P), and *content repository* (CR). The *content repository* component is responsible for maintaining an inventory of available content in all the participating devices and providing access to it. CR instances act both as consumers and providers of the *membership* service. When a new CR instance is created, it will use the membership service provided by an existing instance to become included in the common distributed content repository, and later it may provide this service to another new instance. CR instances also implement partial replication of content to ensure a certain stability of the federated repository even if participants leave. *Presentation* components monitor the content repository in order to find relevant content elements, according to user preferences. They present lists of relevant contents and selected content elements to the BP component. *Browser proxy* components execute as demons and invoke the built-in browser to present the user interface when *InstantSocial* is in the foreground.

7 Related Work

Adaptive Service Grids (ASG) [26] and VieDAME [27] are initiatives enabling dynamic compositions and bindings of services for provisioning adaptive services. In particular, ASG proposes a sophisticated and adaptive delivery lifecycle composed of three sub-cycles: *planning*, *binding*, and *enactment*. The entry point of this delivery lifecycle is a *semantic service request*, which consists of a description of what will be achieved and not which concrete service has to be executed. VieDAME proposes a monitoring system that observes the efficiency of BPEL processes and performs service replacement automatically upon performance degradation. Compared to our planning-based middleware, ASG and VieDAME focuse only on the planning per request of service compositions with regards to the properties defined in the semantic service request. Thus, both approaches do not support a uniform planning of both components and services as our planning-based framework for ubiquitous applications does. However, our planning-based middleware can be extended to integrate ASG and VieDAME adaptive services and thus support the dynamic enactment of service workflows.

Menasce and Dubey [28] propose a QoS brokering approach in SOA. Consumers request services from a QoS broker, which selects a service provider that maximizes the consumer's utility function with regards to its cost constraint. The approach assumes that service providers register with the broker by providing service demands for each of the resources used by the provided services as well as cost functions for each service. The QoS broker uses analytic queuing models to predict the QoS values

of the various services that could be selected under varying workload conditions. This approach is of interest both from the viewpoint of a consumer and a provider. While the client is relieved from performing service discovery and negotiation, the provider is given support for QoS management. This approach, however, requires the client device to be able to access the broker, which might not be possible in ubiquitous environments. Our approach differs in that we consider the offered properties as alternatives to determine the best application configuration and allow the client to adapt to the service landscape.

CARISMA is a mobile peer-to-peer middleware exploiting the principle of reflection to support the construction of context-aware adaptive applications [29]. Services and adaptation policies are installed and uninstalled on the fly. CARISMA can automatically trigger the adaptation of the deployed applications whenever detecting context changes. CARISMA uses utility functions to select application profiles, which are used to select the appropriate action for a particular context event. If there are conflicting application profiles, then CARISMA proceeds to an auction-like procedure to resolve (both local and distributed) conflicts. Contrary to MUSIC, CARISMA does not deal with the discovery of remote services that can trigger application reconfigurations. However, the auction-like procedure used by CARISMA could be integrated in the MUSIC middleware as a particular negotiation protocol.

The conceptual models of both SeCSE (`http://secse.eng.it`) and PLASTIC (`http://www.ist-plastic.org`) focus on service-oriented systems. Inspired by the SeCSE model, the PLASTIC model extends it by introducing new concepts, such as context, location, and service level agreements. The MUSIC and the PLASTIC model have in common that both combine SOA and component-based software development. However, the MUSIC conceptual model uses a component-centric approach, while the PLASTIC model uses a service-centric approach.

Finally, *R-OSGi* extends OSGi with a transparent distribution support [30] and uses *jSLP* to publish and discover services [22]. The communication between a local service proxy and the associated service skeleton is message-based, while different communication protocols (*e.g.*, TCP or HTTP) can be dynamically plugged in. In contrast to *R-OSGi*, the discovery and binding frameworks of MUSIC are open to support a larger range of discovery and communication protocols.

8 Conclusion and Perspectives

In this paper we have introduced the design of a QoS-driven generic planning framework for self-adaptive mobile applications, which seamlessly supports and blends component-based and service-based configurations. In particular, we have shown that the framework is able to adapt to changes in a landscape of ubiquitous remote services that dynamically come and go, and where the offered service qualities vary. The framework exploits these changes to maximize the overall utility of applications. To that aim, the paper has shown how the planning middleware discovers remote services and evaluates them as alternative providers for the functionalities required by an application. The planning framework deals directly with SLA protocols supported by the services to negotiate the best QoS for the end-user. The current MUSIC platform

has already implemented the binding and discovery of services with a range of well-known technologies, while the SLA support is currently under development.

As a preliminary validation of our approach, the paper also explained how the planning framework handles a use case in which the *TravelAssistant* and the *InstantSocial* applications of a mobile user exploit ubiquitous services, such as location, map, and content services, to improve their utility whenever such services become available. The *TravelAssistant* has successfully validated the service binding and discovery, and will be enhanced in future releases. *InstantSocial* will be developed by the end of the MUSIC project (`http://www.ist-music.eu`).

Acknowledgements. We would like to thank our partners of the MUSIC project for valuable comments. This work was partly funded by the European Commission through the project MUSIC (EU IST 035166).

References

1. Mascolo, C., Capra, L., Emmerich, W.: Mobile Computing Middleware. In: Gregori, E., Anastasi, G., Basagni, S. (eds.) NETWORKING 2002. LNCS, vol. 2497, pp. 20–58. Springer, Heidelberg (2002)
2. Rouvoy, R., et al.: Composing Components and Services using a Planning-based Adaptation Middleware. In: Pautasso, C., Tanter, É. (eds.) SC 2008. LNCS, vol. 4954, pp. 52–67. Springer, Heidelberg (2008)
3. Geihs, K., et al.: A comprehensive solution for application-level adaptation. Software: Practice and Experience (2008)
4. Brataas, G., et al.: Scalability of Decision Models for Dynamic Product Lines. In: Int. Work. on Dynamic Software Product Line, DSPL (2007)
5. Floch, J., et al.: Using Architecture Models for Runtime Adaptability. IEEE Software 23(2) (2006)
6. Lundesgaard, S.A., et al.: Construction and Execution of Adaptable Applications Using an Aspect-Oriented and Model Driven Approach. In: Indulska, J., Raymond, K. (eds.) DAIS 2007. LNCS, vol. 4531, pp. 76–89. Springer, Heidelberg (2007)
7. Khan, M.U., Reichle, R., Geihs, K.: Architectural Constraints in the Model-Driven Development of Self-Adaptive Applications. IEEE Distributed Systems Online 9(7) (2008)
8. Fraga, L., Hallsteinsen, S., Scholz, U.: InstantSocial – Implementing a Distributed Mobile Multi-user Application with Adaptation Middleware. EASST Communications 11 (2008)
9. Baida, Z., et al.: A shared service terminology for online service provisioning. In: 6th Int. Conf. on Electronic commerce (2004)
10. Sassen, A., Macmillan, C.: The service engineering area: An overview of its current state and a vision of its future. European Commission. Network and Communication Technologies, Software Technologies (2005)
11. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall, Englewood Cliffs (2006)
12. Dan, A., Ludwig, H., Pacifici, G.: Web service differentiation with service level agreements. IBM White Paper (2003)
13. Andrieux, A., et al.: Web Services Agreement Specification (WS-Agreement), Open Grid Forum Recommended Specification (2005)

14. Flores-Cortés, C.A., Blair, G.S., Grace, P.: An Adaptive Middleware to Overcome Service Discovery Heterogeneity in Mobile Ad Hoc Environments. IEEE Distributed Systems Online 8(7) (2007)
15. Keller, A., Ludwig, H.: The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. Journal of Network and Systems Management 11(1) (2003)
16. Morgan, G., et al.: Monitoring Middleware for Service Level Agreements in Heterogeneous Environments. In: 5th Int. Conf. on e-Commerce, e-Business, and e-Government (I3E), Poznan, Poland, vol. 189 (2005)
17. Dedecker, J., et al.: Ambient-Oriented Programming. In: Companion of the 20th Ann. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) (2005)
18. Rellermeyer, J.S., Alonso, G.: Concierge: a service platform for resource-constrained devices. In: 2nd Eur. Conf. on Computer Systems (EuroSys). ACM, New York (2007)
19. JCP. OSGi Mobile Specification (JSR-232), `http://jcp.org/en/jsr/detail?id=232`
20. Eclipse. Embedded Rich Client Platform, `http://www.eclipse.org/ercp`
21. Sprint. Sprint Titan, `https://developer.sprint.com`
22. Rellermeyer, J.S., Kuppe, M.A.: jSLP, `http://jslp.sourceforge.net`
23. Demuru, M., Furfari, F., Lenzi, S.: DomoWare, `http://domoware.isti.cnr.it`
24. Equinox. OSGi HTTP Server, `http://www.eclipse.org/equinox/server/http_in_equinox.php`
25. Baliuka, J., et al.: Code Generation Library (CGLIB), `http://cglib.sourceforge.net`
26. Kuropka, D., Weske, M.: Implementing a Semantic Service Provision Platform — Concepts and Experiences. Wirtschaftsinformatik Journal (1), 16–24 (2008)
27. Moser, O., Rosenberg, F., Dustdar, S.: Non-intrusive monitoring and service adaptation for WS-BPEL. In: 17th Int. Conf. on World Wide Web (WWW). ACM, New York (2008)
28. Menasce, D., Dubey, V.: Utility-based QoS Brokering in Service Oriented Architectures. In: Int. Conf. on Web Services (ICWS) (2007)
29. Capra, L., Emmerich, W., Mascolo, C.: CARISMA: Context-Aware Reflective Middleware System for Mobile Applications. IEEE Trans. on Software Engineering 29(10) (2003)
30. Rellermeyer, J.S., Alonso, G., Roscoe, T.: R-OSGi: Distributed Applications Through Software Modularization. In: Cerqueira, R., Campbell, R.H. (eds.) Middleware 2007. LNCS, vol. 4834, pp. 1–20. Springer, Heidelberg (2007)

Using Architecture Models to Support the Generation and Operation of Component-Based Adaptive Systems*

Nelly Bencomo and Gordon Blair

Computing Department, InfoLab21,
Lancaster University, LA1 4WA, United Kingdom

Abstract. Modelling architectural information is particularly important because of the acknowledged crucial role of software architecture in raising the level of abstraction during development. In the MDE area, the level of abstraction of models has frequently been related to low-level design concepts. However, model-driven techniques can be further exploited to model software artefacts that take into account the architecture of the system and its changes according to variations of the environment. In this paper, we propose model-driven techniques and dynamic variability as concepts useful for modelling the dynamic fluctuation of the environment and its impact on the architecture. Using the mappings from the models to implementation, generative techniques allow the (semi) automatic generation of artefacts making the process more efficient and promoting software reuse. The automatic generation of configurations and reconfigurations from models provides the basis for safer execution. The architectural perspective offered by the models shift focus away from implementation details to the whole view of the system and its runtime change promoting high-level analysis.

Keywords: software architecture, dynamic adaptation, model-driven engineering, middleware, dynamic variability.

1 Introduction

Adaptability is an increasingly important requirement for many applications, in particular those deployed in dynamically changing environments such as environmental monitoring and disaster management [15, 29]. A well established approach to enable adaptation is to use the support of middleware platforms [8, 9, 24]. Such configurable and reconfigurable middleware platforms are ideally situated to monitor changes, and manage individual adaptations. The primary role of middleware platforms is to ease the development and operation of distributed applications. The approach allows the application developers and domain experts to focus on the application logic, rather than complex runtime, adaptation concerns that the middleware platforms deal with.

Making simpler the development of distributed applications by loading the middleware platform with distribution and runtime concerns causes that development of these platforms require highly technical knowledgable developers. Therefore, middleware developers need to work at very low levels of abstraction. Hence, the development of

* This work was partially funded by the Divergent Grid project, an ESPRC funded project and the DiVA project (EU FP7 STREP).

B.H.C. Cheng et al. (Eds.): Self-Adaptive Systems, LNCS 5525, pp. 183–200, 2009.

components and planning of configurations and reconfigurations involve a large number of variability decisions related to fluctuations of the environment. These decisions are frequently implemented using programming environments and tools with low levels of abstraction (i.e. using constructions offered by programming languages like C++ and Java). The above results in a gap between the way application developers, domain experts and middleware developers operate. Furthermore, the development process frequently uses repeated ad-hoc solutions that many times are carried out manually.

The above reveals the need for both new software development approaches and operational paradigms. These approaches and paradigms should be able to (i) promote the overall view of the system that domain experts and application developers need when planning the adaptation logic, and at the same time (ii) deal with the levels of detailed and technical knowledge required by middleware developers, (iii) bridge the gap in between the different levels of abstraction of (i) and (ii), and (iv) be more systematic and efficient, exploiting software reuse whenever possible. The approach presented in this paper is aiming to address these issues.

Modelling architectural information is particularly important in this context because of the crucial role of software architecture in raising the level of abstraction during development. Such a fundamental role is repeatedly emphasized by the numerous definitions of software architecture [2, 11, 17, 26, 27]. For example, according to Oreizy et all [33] "*a software architecture represents software system structure at a high level of abstraction, and in a form that makes it amenable to analysis, refinement, and other engineering concerns*". This definition is particularly relevant because it also highlights the opportunities for high-level analysis provided by architectural descriptions [17]. The importance of architecture has been recently revisited in the Future of Software Engineering session at ICSE 2007 [25, 38].

The authors argue that MDE and generative software development [13] help to produce new development paradigms to support the life cycle of flexible and dynamically configurable middleware platforms. In the MDE area, research has focused mainly on using models during the phases before execution (i.e. design, implementation and deployment) with emphasis on the generation of software artefacts to be used in those phases (e.g. source code or deployment descriptors) [1, 20]. Moreover, abstractions used for model-based transformations have frequently been related to low level design concepts. Abstractions can also be related to the support for dynamic management and evolution of software [16]. In this sense, model-driven techniques can be used to model software artefacts that take into account the architecture (i.e. high-level structural organization) of the system and its changes according to the fluctuation of the environment.

We propose an approach called Genie. Genie uses domain-specific modelling and dynamic variability (i.e. variability that needs to be solved at runtime) as relevant concepts for the construction of models of the dynamic fluctuation of the environment and contexts, and their impact on the variation of the architecture of the middleware and applications during execution. Genie offers management of dynamic variability during development and allows the systematic generation of middleware related artefacts from high level descriptions (models). To this end, two kinds of dynamic variability are identified, namely *structural variability* and *environment and context variability*. Using the mappings from the models to implementation artefacts, generative techniques

will allow the (semi) automatic generation of implementation artefacts making the process more efficient and promoting software reuse. The authors argue that generating the code associated with configurations and reconfigurations directly from the models provides the basis for defining safer execution by reducing coding errors. The architectural view offered by the models improves high-level analysis shifting focus away from implementation details to the whole view of the system and its runtime change.

The remainder of this paper concentrates on the conception, design, and application of the approach proposed. Section 2 discusses the case of dynamic variability for adaptive systems. Section 3 describes the Genie approach in detail. Section 4 discusses the application of the approach in a specific real case study, the development and operation of an adaptive flood warning system. Finally, Sections 6 and 7 present some related work and conclusions respectively.

2 Dynamic Variability

2.1 Overview

One of the reasons for software variability is to delay design decisions [37]. Instead of deciding on what system to develop in advance, a set of components and a common system family (reference architecture) are specified and implemented during a process called *domain engineering* [13]. Later on, during *application engineering*, specific systems are developed to satisfy the requirements reusing the components and architecture. Variability is expressed in the form of *variation points*. A variation point denotes a particular location in a software-based system where decisions are made to express the selected *variant* [37]. Eventually, one of the variants should be chosen to be achieved or implemented. The time when it is done is called *binding time*. Traditionally, decisions have been deferred to architecture design, implementation, compilation, linking, and deployment as shown in [7, 13, 28, 31, 37]. Currently the aim is to postpone these decisions to even later points in time to allow dynamic variability at runtime. This raises several research challenges, such as their impact in the ongoing architecture of the system and the management of variabilities in dynamically adaptive systems. These challenges are further discussed in the next section.

2.2 Dynamic Variability in Adaptive Systems

A dynamically adaptive system operates in an environment that imposes changing contexts and requirements. The challenge comes from the need to support adaptation or customization of the system according to the needs of the fluctuating environment. The conditions associated with the adaptations may not be completely known before installation of the system. These conditions are related to:

(i) Environment or context variability: the evolution of the environment often cannot be completely predicted during development; therefore the total range of contexts and requirements may be unknown before the system is installed to start execution.

(ii) Structural or architectural variability: this covers the variety of components and the variety of their configurations (architecture). This is a consequence of the environmental variability explained above. In order to satisfy the set of requirements for the

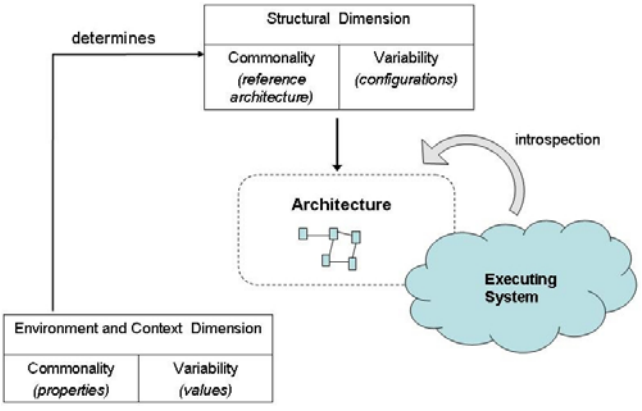


Fig. 1. Dynamic Variability Dimensions

new context, the running system may dynamically add new components or rearrange the current structural configuration (architectural reorganization). Hence, solutions cannot be restricted to a set of known-in-advance configurations and components. New sets of components may be added during execution (see Figure 1).

The system should be prepared to deal with the two dimensions of variability described above. Under new contexts, the system must be prepared to discover and include new components to meet new requirements or simply to improve the current state of the system when new components become available [36] and according to some quality of service (QoS) properties. Solutions to manage the latter structural variability cannot be just the traditional component replacements and/or specializations, but decisions should involve more powerful mechanisms able to manage whole sets of components, their connections and semantics (architecture). Moreover, the correct match between the architectural changes and the environmental context should be maintained.

2.3 Architectural Reorganization Supported by Middleware Platforms

At Lancaster University, we have gained experience developing adaptive systems and middleware platforms using *component frameworks* and *reflective technologies* [12]. We use of component frameworks and reflection as flexible mechanisms for supporting runtime variability of dynamically extensible systems. Component frameworks are collections of components that address a specific area of concern and accept "plug-in" components that add or extend behaviour [12]. Reflective capabilities support introspection to observe and reason about the state of the system to make decisions on architectural reconfigurations. Adaptive behavior is defined by sets of *reconfiguration policies*. These policies are of the form *on-event-do-actions* and *actions* are architectural changes using the component frameworks. During runtime, a *context engine* receives relevant *environmental events* that are employed to identify the reconfiguration policy to be used. Crucially, component frameworks offer the medium to provide architectural variability. Furthermore, reflective capabilities offer the potential to reason

about the possible variation points and their variants during execution. The support offered by the middleware platforms provides the technique to implement variability called ***infrastructure-centered architecture*** [37]. When using this technique, connections between components are treated as first-class entities. This means that required interfaces of components are not hard-coded. Dynamic replacement of a component in the architecture or indeed dynamic reorganization of the architecture is eased if the architecture and the location where such modifications could be carried out is made explicit. "*Used correctly, this technique yields perhaps the most dynamic of all architectures*" [37]. The next section elaborates how the middleware platforms act as the reconfiguration framework and introduces the approach proposed. The approach leverages the middleware platforms to guide the domain experts and developers during the modelling and generation of software artefacts, and during the operation of middleware platforms and applications. This is a key assumption that the underlying middleware platform ensures consistency and integrity using change transactions [12].

3 The Genie Approach

3.1 Overview

The proposed approach is called Genie. Genie uses domain specific languages (DSLs) for the construction of the models associated with both the structural (architectural) and the environment variability. Using models and generative techniques, software artefacts can be generated more efficiently. Supported by the middleware platforms, applications can be dynamically reconfigured from one structural *variant* to another according to changes in the context or environment. The system monitors specific properties of the runtime environment and reacts to given changes while keeping a valid architecture. The system is able to decide what kind of architectural reorganization (reconfiguration) has to be performed, if any.

To model the adaptive behaviour described above it is necessary to define what adaptation means in terms of configurations (architecture) and conditions:

*An **adaptation** is defined in the scope of this research as the process of having the system transforming itself from a given configuration **Ci** to another configuration **Cj** given the set of conditions **Tk**.*

The set of conditions correspond to variants of the context and environment variability and the configurations (components and connections) correspond to variants dictated by the architectural and structural variability. The next section describes the proposed approach.

3.2 Description of the Genie Approach

The Genie approach allows the use of DSLs to specify:

i. **the structural variability.** The DSL associated with the structural variability allows the modelling of the component configurations to be expressed in terms of the architecture dictated by component frameworks. The modelling elements to be used are generic architectural elements such as components, required and offered interfaces, and bindings.

ii. **the environment and context variability.** The *transition diagram* DSL is used to specify the conditions that represent the dynamic nature of the environment and context. Basically, this DSL is used to specify adaptations of the form described above: from the configuration *Ci* and on the set of conditions *Tk*, go to configuration *Cj*. These models are in essence *transition diagrams*.

Using generators capable of traversing the models created with the DSLs and their trace relationships, different software artefacts can be generated:

iii. **components and configurations of components** associated with the component frameworks are generated from the structural variability models. The constraints specified by the component frameworks are captured in the models to allow validation of the configurations and ensure consistency of the resultant artefacts. The middleware platforms allow the newly generated components and component configurations to be added during the execution of the system.
iv. **reconfiguration policies** are generated from the transition models. As in (iii), validation of the diagrams should be performed to avoid inconsistencies. The middleware platforms allow the generated policies to be inserted during execution. The newly added reconfiguration policies are used as long as the "new" component(s) or component configuration(s) for the right match are also provided.

3.3 Levels of Abstraction

An overview of the different levels of abstraction promoted by Genie is given in Figure 2. The figure shows the specific artefacts that populate the layers which correspond to different levels of abstraction (abstraction levels are raised from bottom to top).

(1) The first level at the bottom is populated by different software artefacts like source code, XML configuration files describing the different configurations associated with component frameworks, and the XML files of reconfiguration policies.

(2) The second level corresponds to the architectural models associated with *structural variability*, i.e. the models of component frameworks and their components and configurations. These models offer visual representations of the component configurations, their components and interfaces.

(3) The third level at the top corresponds to the *environment and context variability*. Here the developer plans the adaptations based on transition diagrams. At this level the developer reasons in terms of structural variants (associated with specific domains of concern) and conditions of the environment that trigger the reconfigurations.

The first and second levels are similar to existing approaches using architecture description languages (ADLs) and offering tool chain support for the development of component-based systems. Actually, the DSL associated with the structural variability can be considered as an ADL with generative capabilities. The major contribution of the Genie approach is the support to high level analysis and automation provided by level 3 and its relationships with the other two levels.

Each node in the transition diagrams of level 3 is considered as a *structural variant* of the system. Structural variants are "*coarser grain*" configurations than configurations

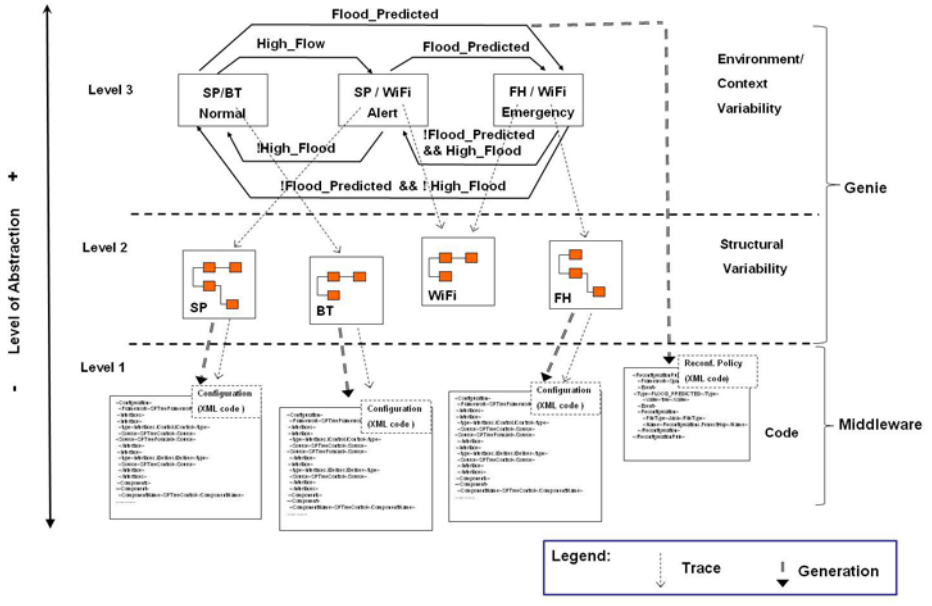


Fig. 2. Different levels of abstraction supported by Genie

associated with individual component frameworks in the sense that they are described by a set (or n-tuple) of component frameworks. Structural variants can be seen as configurations of component frameworks. The set of component frameworks are associated with the problem domain. Thus, for example, if the problem domain identified requires architectural changes (in terms of reconfiguration) of the routing protocols and the topology of nodes in a sensor network, the component frameworks to be used in each structural variant should represent concepts associated with routing protocols and topologies of nodes. The proposed approach aims then at partitioning each structural variant into a set of specialized and focused domains of concern.

Figure 2 shows the *trace* relationships between the levels. At the top, each structural variant has the references to both the related component frameworks in level 2 and the files of reconfiguration policies in level 1. The component frameworks in level 2 make reference to the files associated with the configuration files. In turn, the reconfiguration files point to the executable code associated with the components. In the example of Figure 2, each structural variant of the transition diagram is described in terms of two component frameworks (the *Spanning Tree* component framework for routing protocols and the *Network* component framework) that opportunistically correspond to the case study described in the next section. From the initial architecture (configuration) the system will evolve over time according to the conditions of the environment specified in the arcs of the diagrams. The places where the architecture can be changed and the consequences of the changes will be driven by the transition diagrams.

The use of DSLs in the approach described above promotes higher levels of abstraction beyond programming and code. The benefits from raising the levels of abstraction

using models is twofold. First, automation levels are improved as the models allow the specification and application of repetitive patterns that are used in the generation of software artefacts. As a result software reuse is encouraged. Second, the gap between the way requirements engineers, domain experts, software architects and programmers operate is reduced, thereby promoting their joint collaboration as shown in [21]. Furthermore, the use of generative techniques increases the levels of efficiency and automation. The approach can be applied using different middleware platforms that work with concepts of components and component frameworks like in the case of OpenCOM or Fractal as reported in [3] and [30].

Next section discusses the application of the approach in the specific case of the development and operation of an adaptive flood warning system [23]. The approach has also been applied in the context of dynamic service discovery scenarios for mobile applications with results reported in [3].

4 Case Study: A Wireless Sensor Network for Flood Management

4.1 Overview

The *Genie tool* is the implementation of the Genie approach for the case of the OpenCOM based middleware platforms at Lancaster University and is described in detail in [3, 5]. The DSL associated with the structural variability is called the OpenCOM DSL in the case of the *Genie tool*.

The use of the architectural models supported by Genie and its tool is explained using the case study GridStix [23]. GridStix is a wireless sensor network for flood management that has been deployed in prototype form on the flood plain of the River Ribble in North Yorkshire, England. About 15 nodes have been deployed. Sensors route the data collected in real-time using a spanning tree topology to one or more designated *root nodes*. From these nodes, the data is forwarded (via General Packet Radio Service, GPRS) to a *prediction model* that runs on a remote computational cluster. Each sensor node includes a 400MHz XScale CPU, 64MB of RAM, 16MB of flash memory, and Bluetooth and WiFi Networks. The designated root nodes are also equipped with GPRS. Each GridStix is powered by a 4 watt solar array and a 12V 10Ah battery. Linux 2.6 and the Java virtual machine 1.4 are used in contrast to conventional sensor network deployments, where sensors are simply responsible for transmitting sensor data off-site. This deployment permits the use of local processing. Local processing supports computation for the local prediction of future environmental conditions.

Level 3 of Figure 2 shows the transition diagram that guides the reconfiguration and adaptation process of GridStix. Three possible states were identified: *Normal*, *Alert*, and *Emergency*. Each state of the system has a specific structural (architectural) variant. The problem domain identified requires structural changes (in terms of reconfiguration) of the routing algorithms and the networks interfaces to be used in the sensor network. The component frameworks to be used in each *structural variant* represent concepts associated with the overlays component framework, specifically the *Spanning Tree* and the *Network* framework.

The *Spanning Tree* component framework supports the routing algorithm that has two possible variants: *Shortest Path* (SP) and *Fewest Hop* (FH). The *Spanning Tree*

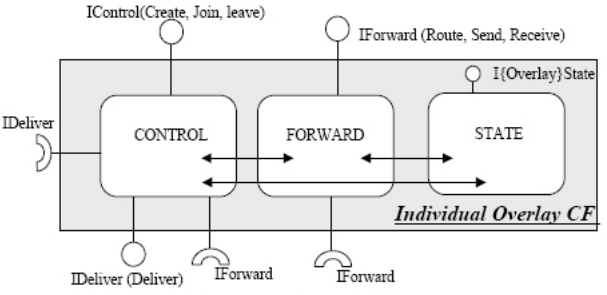


Fig. 3. Overlays Pattern Architecture

component framework and its variants follow the overlay pattern architecture (control, forward, and state). The overlays pattern is shown in Figure 3. This is part of a more generic pattern that allows the overlay of individual plug-ins to be inserted into component frameworks. As such, the reuse of architecture design is one of its main contributions. The use of models to generate artefacts further exploits this contribution. The second one, the *Network* component framework, describes the type of network to be used and offers two possible variants: *BlueTooth*(BT) and *WiFi*.

The 2-tuples associated with the structural variants used in the case study are (*SP,BT*), (*SP,WiFi*), and (*FH,WiFi*) and correspond with the three possible states identified above. Figures 4 and 5 show the *Shortest Path* variant and *WiFi* variant models developed with the *Genie tool* respectively.

The system will evolve over time according to changes in the conditions of the environment specified in the arcs of the diagrams. The places where the architecture can be changed and the consequences of the changes will be driven by the transition diagrams. How different variants of these component frameworks are chosen will depend on the possible multiple variations of conditions in the environment and context. This variation is specified using the triggers associated with the transitions in the diagram (i.e. arcs). Triggers of reconfiguration policies are specified in the arcs between states. The number of transitions in the transition diagrams will depend on how adaptable the system should be or is conceived.

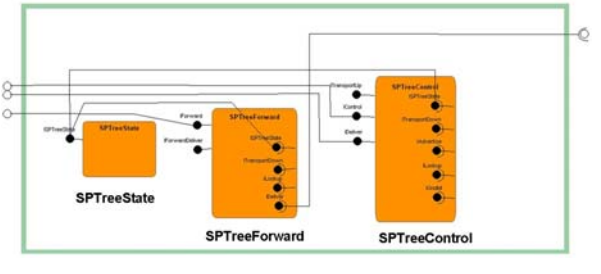


Fig. 4. Shortest Path Variant for the Spanning Three component framework

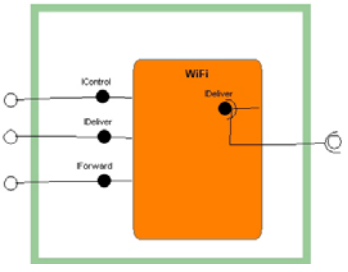


Fig. 5. WiFi Variant for the Network component framework

According to the transition diagram in Figure 6, if the application is operating as *Normal*, and the prediction model of GridStix predicts an imminent flood (i.e. the *FloodPredicted* monitoring condition is true), the nodes adapt to the *Emergency* state bypassing the *Alert* state. This adaptation is effected by reconfiguring the *Network* to use *WiFi* instead of *BlueTooth*, and the *Spanning Tree* to a *Fewest Hop* topology.

One of the advantages of using *transition diagram* models is that they offer a complete view of the reconfiguration opportunities of the system offered to the user. The architectural perspective offered by these models shift focus away from the source code of isolated policies to the whole view of the reconfiguration opportunities and

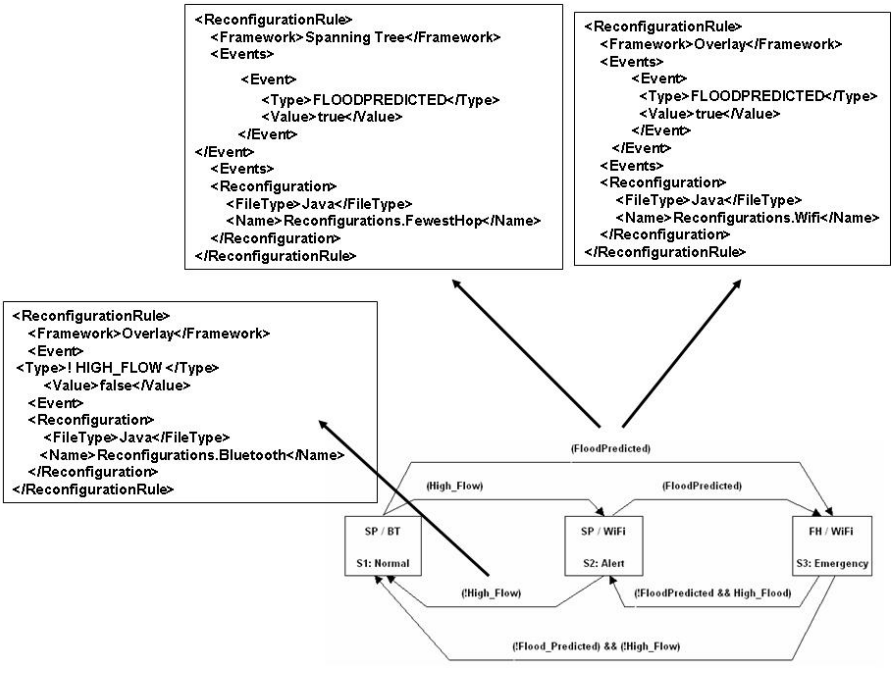


Fig. 6. The transition diagram of the case study and three generated reconfiguration policies

the component frameworks involved. Different stakeholders can abstract away irrelevant implementation-related details and focus on the big picture: the system structure and its runtime change. It should be contrasted with the partial view when working with individual policies using traditional approaches. Figure 6 shows 3 examples of the reconfiguration policies that are invoked when the specified monitoring conditions are met. From this specific example, a total of 8 reconfiguration policies can be generated. For readability purposes, the transition diagram proposed is simple, as the number of policies increases rapidly with the number of triggers considered.

4.2 Orthogonal Variability Models

To complement the approach described above, the orthogonal variability models proposed by *Klaus Pohl et all* [34] are used. An orthogonal variability model (OVM) defines the variability of a system family in a separate model. It relates the variability specified to other software development models such as component models in our case. Figure 7 shows the variability diagrams used to model the variants in the case study. The three structural variants, *Normal*, *Alert*, and *Emergency* are associated with the variation point *VP:Flood App* marked by (a). Each state variant of the graph is described using two component frameworks, i.e. the *Spanning Tree* and the *Network* component framework as seen above. The *Spanning Tree* and the *Network* component frameworks have variation points associated themselves, marked by (b) and (c).

The OVM has mainly been used by developers to document variability. However, in our work these variability models have been useful not just to document variability.

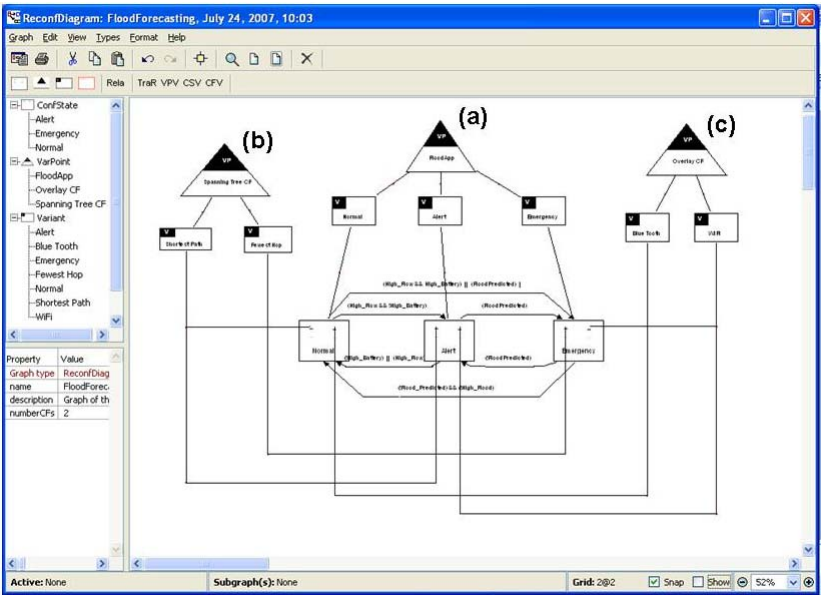


Fig. 7. Variability and Transition Diagrams

Particularly, OVMs have proofed to be useful (i) when traversing the models to generate the reconfiguration policies, (ii) to keep links to adaptation requirements (using goal models) [21], and (iii) when managing the traceability relations between the structural variants of the transition diagrams (level 3) and the component frameworks configurations (level 2).

4.3 Artefacts Generated by the *Genie Tool*

The developer designs models to specify the components, component frameworks and configurations, structural variants and the transition diagrams using the DSLs provided by the *Genie tool*. Using generators that traverse these models, different software artefacts of level 1 can be generated (see Figure 8).

From the models specified using the OpenCOM DSL the source code of components and configurations of components associated with the component frameworks is generated. Similarly, using the models associated with the transition diagrams, the reconfiguration policies are generated. To ensure consistency of the generated artefacts, the constraints specified by the models are used to validate the configurations before any generation is carried out. The middleware platforms enable extensibility and evolution of the system allowing newly generated artefacts (e.g. components, component configurations, and reconfiguration policies) to be added during the execution of the system.

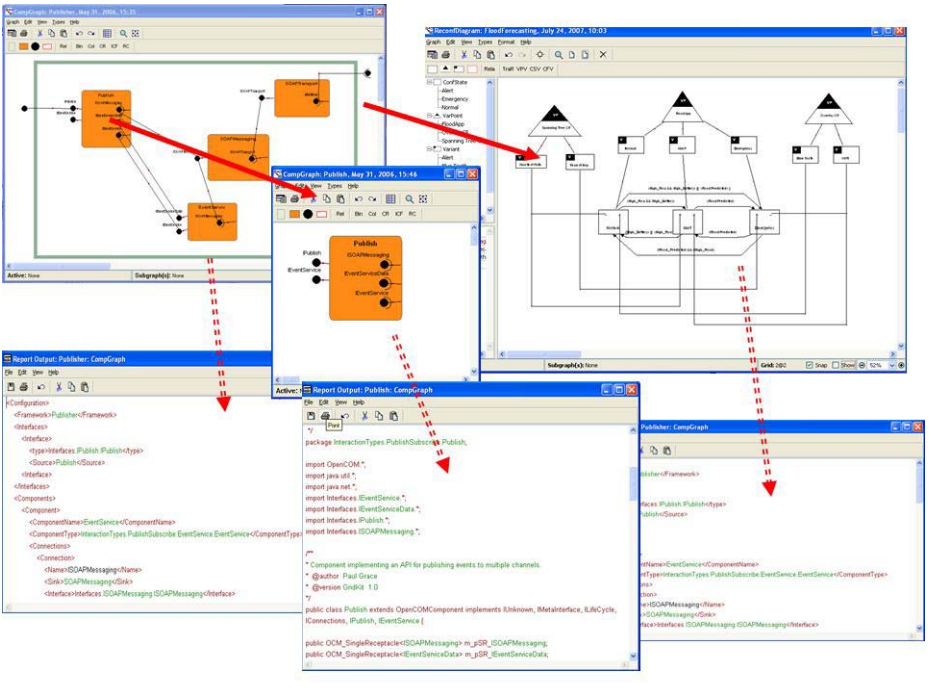


Fig. 8. Genie Models and Generated Artefacts

5 Discussion

In this section we discuss the novel contributions of our research and briefly describe ongoing research that extends the Genie approach.

5.1 Contributions

Architectural changes take place according to environment variations and following the adaptation policies. With Genie, new reconfiguration policies can be modeled and generated off-line while the system is running. Using the capabilities provided by the middleware platforms, the newly generated policies can also be added to the running system, changing dynamically the behaviour of the system. The fact that these policies are explicitly modelled using the Genie approach improves the traceability during the software development process. The overall view offered by the transition diagrams described above contrasts with the partial text-based view offered by each reconfiguration policy. Using only partial views makes it very probable that the developers ignore, or simply lose sight of, important interdependency relationships. Overlooking dependencies can cause failures and inconsistencies during execution. Identifying the source of the error may require significant effort and time [3]. Genie promotes joint collaboration between requirements engineers, domain experts, software architects, and developers [21]. Furthermore, the proposed approach makes explicit the support the middleware platforms provide in separating the *system evolution* and *system adaptation* as two simultaneous processes in self-adaptive software [32]. System evolution ensures the consistent application of change over time, and system adaptation focuses on "*the cycle of detecting changing circumstances and planning and deploying responsive modifications*" [32].

5.2 Ongoing Research

Towards the Use of Models@run.time

We are already extending and improving the Genie approach. As explained above and as Figure 6 shows, the generated policies mainly specify the trigger events and which reconfiguration scripts have to be loaded to adapt the system from one configuration (state) to another. These scripts are currently hand-written using the support offered by the underlying middleware platforms.

The reconfiguration scripts can also be generated from the DSL-based models during design-time or even at runtime. In the case of the dynamic generation during runtime and when adaptations (transitions) are triggered, the current configuration and the target configuration are compared. The comparison results in the identification of the components that should be added or deleted and allows the dynamic generation of the corresponding reconfiguration script. This solution is possible as we are able to maintain a reference model at the meta-level of the reflective middleware platform. The reference model represents the current system and the possible modified model that is the result of the required adaptation (both models are supplied by Genie). Partial results of such ideas are already reported in [30]. This work opens some research questions as for example: what is the correct order of the deletion and incorporation of components during

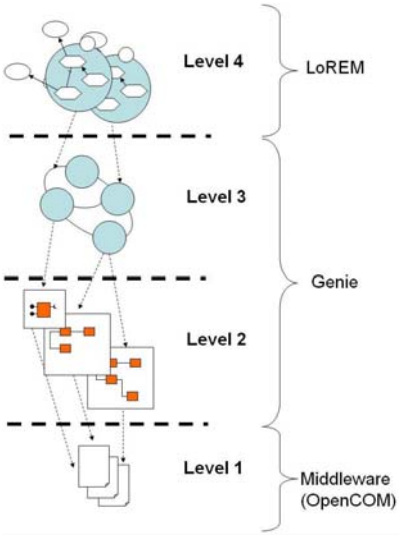


Fig. 9. Genie Models and Generated Artefacts

the reconfiguration process, or what is the impact on the performance of the application. This research is being carried out in the scope of the STREP European project DiVA [14] (work package diva@run.time) and the research topic Models@run.time [4].

Goal-Driven Requirements

Genie is complemented by the approach Levels of RE for Modeling (LoREM) [21]. LoREM is a goal-driven requirements approach that supports the formulation of the requirements of dynamically adaptive systems helping the analyst to understand the characteristics of the operational environment and the adaptation scenarios that the system can go through. The goal models in LoREM are in a fourth abstraction level and are used to derive the DSL-based models used in Genie. At the bottom, the middleware platform underpins the reorganization of the ongoing architecture at runtime providing support as the requirements imposed by the environment change, see Figure 9. Partial results applied to GridStix can be found in [21].

6 Related Work

Research work by *Floch et all* [15] on the use of architecture models for runtime adaptability, shares the basic principles of our approach as for example the use of component frameworks to support variability. They also take into account the benefits of coarse-grained variability mechanisms. However our approach is more general as their focus is only on mobile computing applications. *Sora et all* [36] also use architecture-based abstractions for what they call self-customizable system. They composable components

which are similar to our component frameworks. They apply recursive composition according to external requirements using ADLs what can be to some extent equivalent to our reconfiguration policies. However, they do not offer reflection capabilities, i.e their systems cannot reason about the current state or configuration of the system. Reflection offers potential support to determine where the points for variation are, what the possible set of variations are, or the state of the system at any point in time. However, using reflection has some drawbacks as the effect on performance and integrity issues. When developing reflective systems a trade-off between flexibility and performance has to be studied and a rigorous system development has to be performed. Neither of the solutions in [15] and [36] provides generative capabilities as we offer. In [21] we explain how the policy mechanisms contribute to providing a clear trace from user requirements to adaptation requirements [6] and their implementations. In this sense, the research related to requirements-driven composition in [36] is similar to our research.

Many mechanisms for runtime variability management have been proposed. They are mainly focused on exchange of runtime entities, parametrization, inheritance for specialization, and preprocessor directives [19, 35, 37]. Our approach proposes the model-based architecture management for whole sets of components, their connections and semantics (i.e. we offer a more coarse grained approach).

In [18], *Garlan* and *Schmerl* describe their research work on the use of system monitoring and reflective capabilities using architectural models. Specifically, they describe their approach to monitor the executing system to translate observed events to events that construct and update an architectural model that reflects the actual running system. The final goal is to compare the dynamically-determined model with the correct architectural model. *Garlan* and *Schmerl* argument how inconsistencies found after the comparison can be used to identify implementation errors, or, even possibly, to effect runtime adaptations to correct certain type of faults. Different from *Garlan* and *Schmerl*, we do not deal in this paper with the self-healing issues of adapted systems. Our focus is first, to offer the overall view of the system that domain experts and developers need when planning the adaptations and secondly, to bridge the gap in between the different levels of abstraction used by domain experts and middleware developers. There are different research projects on architecture-based dynamic adaptations that use ADLs including the research by *Garlan* and *Schmerl*. As explained in Section 3.3, the DSL associated with the structural variability in Genie can be considered as an ADL with generative capabilities. A difference between our approach and other ADL-based approaches such as ArchWare, Rainbow, ArchStudio [17, 25, 38] is that our architectural descriptions are always tied to a component framework. Component frameworks offer the architectural principles and constrains that address a specific area of concern (e.g. routing protocol or discovery service). Furthermore, in our case systems can be assembled from component frameworks in a recursive way. A component plugged into a component framework may be an atomic component, but it can also be a compound component that is a component framework itself. The view provided by this recursivity along with the domain oriented nature provided by component frameworks is different from the often flat view offered by the research projects named above.

When performing dynamic reorganization of the architecture we ensure that updates are completed atomically and do not impact the integrity of the network. To do this,

frameworks are placed in a quiescent state ensuring that the reconfiguration is complete and correct. We are investigating the use of architectural patterns to drive the generation of software artefacts related to safe reconfiguration at that level. In this sense the work presented by Gomaa and Hussein [22] is relevant and complementary to our research. Finally, we see potential use in combining our approach with the Fujaba project principles [10]. Fujaba supports gradual transitions from one configuration to another and the specification of timing constraints. The principle of gradual transitions can improve the prescriptive policies of our approach.

7 Conclusions and Future Work

This paper has presented an approach called Genie. The approach uses architecture models to support the generation and operation of component-based adaptive systems. We have identified two dimensions of dynamic variability namely *architectural or structural variability* and *environment and context variability*. Genie supports the use of domain-specific languages to specify and validate models based on abstractions of the dynamic variability dimensions. Models describe the architecture of reconfigurable applications and the conditions of the environment and context that trigger the reconfiguration of the architecture. From the models, different software artefacts (e.g. components configurations and reconfiguration policies) can be generated. These artifacts can be dynamically added to the system during its execution using the middleware platforms. Such artefacts support the dynamic architectural reorganization, runtime decision-making and system adaptation mechanisms. Specifically, transition diagram models allow the developer to work at higher levels of abstraction. An important contribution of Genie is the overall view of the different problem domains and the whole process of reconfiguration that the systems can undergo. The approach has been successfully applied in two different case studies.

Substantial research remains to be done. For example, a concern is the combinatorial explosion related to the number of reconfiguration paths in the reconfiguration graphs and the number of adaptation policies). The number of reconfiguration paths in the case study was manageable. However, it might not be the case for other domains. Furthermore, even if the designer specifies the reconfiguration graphs at a high-level of abstraction, the explosion of the number of configurations and transitions to be specified is a potential problem. We are investigating how to dynamically generate the reconfiguration scripts and how to avoid the enumeration of all possible configurations. Partial results are shown in [30].

References

1. Balasubramanian, K., Gokhale, A., Karsai, G., Sztipanovits, J., Neema, S.: Developing applications using model-driven design environments. IEEE Computer, 33–40 (2006)
2. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. Addison-Wesley Professional, Reading (2003)
3. Bencomo, N.: Supporting the Modelling and Generation of Reflective Middleware Families and Applications using Dynamic Variability. PhD thesis, Lancaster University (2008)

4. Bencomo, N., France, R., Blair, G.: 2nd international workshop on models@run.time. In: Giese, H. (ed.) MODELS 2007. LNCS, vol. 5002, pp. 206–211. Springer, Heidelberg (2008)
5. Bencomo, N., Grace, P., Flores, C., Hughes, D., Blair, G.: Genie: Supporting the model driven development of reflective, component-based adaptive systems. In: ICSE 2008 - Formal Research Demonstrations Track (2008)
6. Berry, D.M., Cheng, B.H.C., Zhang, P.J.: The four levels of requirements engineering for and in dynamic adaptive systems. In: 11th International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ 2005), Porto, Portugal (2005)
7. Beuche, D., Papajewski, H., Schröder-Preikschat, W.: Variability management with feature models. Science of Computer Programming. Special issue: Software variability management 53(3), 333–352 (2004)
8. Blair, G., Coulson, G., Robin, P., Papathomas, M.: An architecture for next generation middleware. In: Seitz, J., Davies, N.A.J., Raymond, K. (eds.) IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 1998), The Lake District, UK, pp. 91–206 (1998)
9. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.-B.: The fractal component model and its support in java. Software: Practice and Experience 36(11), 1257–1284 (2006)
10. Burmester, S., Giese, H., Hirsch, M., Schilling, D., Tichy, M.: The fujaba real-time tool suite: model-driven development of safety-critical, real-time systems. In: ICSE (2005)
11. Clements, P., Kogut, P.: The software architecture renaissance. Crosstalk - The Journal of Defense Software Engineering 7(11) (1994)
12. Coulson, G., Blair, G.S., Grace, P., Joolia, A., Lee, K., Ueyama, J., Sivaharan, T.: A generic component model for building systems software. ACM Transactions on Computer Systems (February 2008)
13. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools and Applications. Addison-Wesley, Reading (2000)
14. DiVA. Diva-dynamic variability in complex, adaptive systems (2008), `http://www.ict-diva.eu/`
15. Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., Gjorven, E.: Using architecture models for runtime adaptability. Software IEEE 23(2), 62–70 (2006)
16. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: Briand, L., Wolf, A. (eds.) FoSE. IEEE-CS Press, Los Alamitos (2007)
17. Garlan, D.: Software Architecture: a Roadmap. ACM Press, New York (2000)
18. Garlan, D., Schmerl, B.: Using architectural models at runtime: Research challenges. In: European Workshop on Software Architectures, St. Andrews, Scotland (2004)
19. Goedicke, M., Pohl, K., Zdun, U.: Domain-specific runtime variability in product line architectures. In: 8th International Conference on Object-Oriented Information Systems, pp. 384–396 (2002)
20. Gokhale, A., Balasubramanian, K., Lu, T.: Cosmic: addressing crosscutting deployment and configuration concerns of distributed real-time and embedded systems. In: OOPSLA 2004 Companion Book, NY, USA, pp. 218–219. ACM, New York (2004)
21. Goldsby, H.J., Sawyer, P., Bencomo, N., Hughes, D., Cheng, B.H.C.: Goal-based modeling of dynamically adaptive system requirements. In: 15th Annual IEEE International Conference on the Engineering of Computer Based Systems (ECBS) (2008)
22. Gomaa, H., Hussein, M.: Model-based software design and adaptation. In: Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2007) (2007)
23. Hughes, D., Greenwood, P., Coulson, G., Blair, G., Pappenberger, F., Smith, P., Beven, K.: Gridstix: Supporting flood prediction using embedded hardware and next generation grid middleware. In: 4th International Workshop on Mobile Distributed Computing (MDC 2006), Niagara Falls, USA (2006)

24. Kon, F., Costa, F., Blair, G., Campbell, R.: The case for reflective middleware. Communications of the ACM 45(6), 33–38 (2002)
25. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: FoSE 2007: 2007 Future of Software Engineering, pp. 259–268. IEEE Computer Society, Los Alamitos (2007)
26. Kruchten, P., Thompson, C.: An object-oriented, distributed architecture for large scale ada systems. In: Tri-Ada 1994, Baltimore, Maryland (1994)
27. Lawson, H., Kirova, V., Rossak, W.: A refinement of the ecbs architecture constituent. In: International Symposium and Workshop on Systems Engineering of Computer Based Systems, Tucson, Arizona (1995)
28. Lee, J., Muthig, D.: Feature-oriented variability management in product line engineering. Communications of the ACM 49(12) (2006)
29. McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C.: Composing adaptive software. IEEE Computer 37(7), 56–64 (2004)
30. Morin, B., Fleurey, F., Bencomo, N., Jezequel, J.-M., Solberg, A., Dehlen, V., Blair, G.: An aspect-oriented and model-driven approach for managing dynamic variability. In: MODELS 2008 Conference, France (2008)
31. van Ommering, R.: Building Product Populations with Software Components. PhD Thesis. PhD thesis, Rijksuniversiteits Groningen (2004)
32. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An architecture-based approach to self-adaptive software. IEEE Intelligent Systems and Their Applications 14(3), 54–62 (1999)
33. Oreizy, P., Rosenblum, D.S., Taylor, R.N.: On the role of connectors in modeling and implementing software architectures. Technical Report 98-04, University of California, Irvine (1998)
34. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering- Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
35. Posnak, E., Lavender, G.: An adaptive framework for developing multimedia. Communications ACM 40(10), 43–47 (1997)
36. Sora, I., Cretu, V., Verbaeten, P., Berbers, Y.: Managing variability of self-customizable systems through composable components. Software Process: Improvement and Practice 10(1), 77–95 (2005)
37. Svahnberg, M., van Gurp, J., Bosch, J.: A taxonomy of variability realization techniques. Software: Practice and Experience 35(8), 705–754 (2005)
38. Taylor, R.N., van der Hoek, A.: Software design and architecture the once and future focus of software engineering. In: International Conference on Software Engineering, ICSE 2007 (FoSE 2007) (2007)

Model-Driven Assessment of QoS-Aware Self-Adaptation

Vincenzo Grassi[1], Raffaela Mirandola[2], and Enrico Randazzo[1]

[1] Dipartimento di Informatica, Sistemi e Produzione
Università di Roma "Tor Vergata", Italy
vgrassi@info.uniroma2.it, randazzo@info.uniroma2.it
[2] Politecnico di Milano, Italy
Dipartimento di Elettronica e Informazione
mirandola@elet.polimi.it

Abstract. One of the main goals of a self-adaptable software system is to meet the required Quality of Service (QoS) by autonomously modifying its structure/behavior in response to changes in the supporting infrastructure and surrounding physical environment. A key issue in the design and development of such system is the assessment of their effectiveness, both in terms of their ability to meet the required QoS under different operating conditions, and in terms of the costs involved by the reconfiguration process, which could outweigh the benefit of the reconfiguration. This paper introduces an approach to support this assessment, with a focus on performance and dependability attributes. Our approach is based on the idea of defining a model transformation chain that maps a "design oriented" model of the system to an "analysis oriented" model that lends itself to the application of a suitable analysis methodology. We identify some key concepts that should be present in the design model of a dynamically adaptable system, and show how to devise a transformation from such a model to a target analysis models, focusing in particular on models of component or service oriented systems.

1 Introduction

Dynamic self-adaptation is becoming an important feature for software systems, due to the emergence of more and more classes of applications that are highly complex and distributed, and operate in heterogeneous and rapidly changing environments like those from the mobile and pervasive computing domains [8,18,25,32]. Thanks to self-adaptation, a software system can cope with changes in the execution environment by modifying at runtime its behavior or structure.

Designing and maintaining such systems is a challenging task. A key issue to be faced concerns the assessment of their effectiveness, in terms of the ability to meet their functional and non-functional requirements. In particular, in the case of non-functional requirements concerning the delivered quality of service (QoS), this assessment should take into account the cost of the adaptation process itself. Adapting a system can require time and system resources to be carried out, and this cost could even outweigh the potential QoS benefit.

In this respect, our goal is to support the design and management of dynamically adaptable software systems by means of the model-based analysis of their

B.H.C. Cheng et al. (Eds.): Self-Adaptive Systems, LNCS 5525, pp. 201–222, 2009.

effectiveness, focusing in particular on their ability to meet non-functional requirements concerning *performance* and *dependability* attributes.

Modeling frameworks for dynamically changing software systems have been already proposed. Some of them are mainly targeted to the analysis of functional requirements [2,5,6,11,18,22], and hence are not suitable for the effectiveness analysis of such systems with respect to performance or dependability, while others address the analysis of non functional requirements and are hence closer to our goal [3,7]. However, existing frameworks do not always consider explicitly the modeling and analysis of the tradeoff between the advantages and costs of the system adaptation. Besides this, such frameworks are often based on formal notations and modeling principles which are quite far from those used by software designers, thus making awkward their integration in the design process.

In this paper we propose a comprehensive framework, where both the costs and benefits of adaptation (in terms of its impact on performance or dependability) can be properly modeled and analyzed. We also address the problem of how our modeling approach can be actually integrated in the design process of a dynamically adaptable system, leveraging ideas from the *Model Driven Development* (MDD) paradigm [4].

MDD typically focuses on a transformation path, supported by automatic transformation tools, from high level to platform specific models (up to the executable code) of a software system [3,4,24]. The idea of exploiting MDD methodologies for QoS assessment has emerged in recent years (see [10] for a review of these methods). Indeed, the construction of a QoS analysis model can be seen as a special type of model transformation whose source is a "design oriented" model of the system (produced during the design process by the system designers), while the target is a suitable "analysis oriented" model, which lends itself to the application of sound analysis methodologies. Recently, Ardagna et al. [3] presented a conceptual map where different methods and models are positioned in a general MDD framework.

One of the main motivations is that embedding within automatic model transformation tools relevant parts of the expertise required to build QoS analysis models should facilitate the integration of QoS assessment in the design and development process. Indeed, it should allow to quickly get QoS analysis results starting from the available design artifacts, and should also facilitate the use of QoS analysis methodologies within design teams with little expertise in this field.

Existing MDD-based methodologies for the generation of QoS analysis models do not consider the modeling of adaptable systems. Moreover, they often devise the transformation path as a single step transformation from the source design oriented model to the target analysis oriented model. This single step transformation could be excessively complex, for several reasons: the large semantic gap between the source and target models, the different notations that could be used in the source model, and the different target notations one could be interested in, to support different kinds of analysis (e.g. queueing networks, Markov processes).

To face these problems, we propose a two-step transformation path from design oriented to analysis oriented models centered around the construction of a *bridge model* expressed in a suitable *intermediate modeling language*.[1] The goal is to split into two parts the complex task of deriving an analysis model (e.g. a queueing

[1] To better manage the process of devising each of these two basic steps, they could be further decomposed into sub-steps.

network) from a high level design model (expressed using a design oriented notation, e.g. UML):

1) extracting from the design model the information which may be relevant for the analysis of performance or dependability attributes and expressing it in a bridge model;

2) generating an analysis model based on the information expressed in the bridge model.

To support this approach, the intermediate language we propose supports the abstract and simplified representation of concepts we may expect be expressed in the source design oriented model of an adaptable system.

The two parts of the proposed transformation path can be tackled independently of each other, which makes simpler to deal with each of them. Moreover, a positive consequence of this splitting centered around a bridge model is that it facilitates the re-use of work already done for one of the two parts. Indeed, given a particular notation used for the design of adaptable systems (e.g. a notation based on a suitable customization of UML [19]), the QoS assessment of models expressed in this notation can take advantage of an already defined transformation from bridge models to some kind of analysis model: what remains to be done is the (presumably) simpler transformation from the source design model to the bridge model, rather than the more complex thorough transformation from the source model to the analysis model. Similarly, once a transformation from a specific kind of design model to the bridge model has been defined, the set of QoS analysis methodologies that can be used (*e.g.*, analytic, or simulation-based) can be extended by simply defining a new transformation from the bridge model to a new kind of analysis model.

The use of bridge models expressed in some suitable intermediate language has been already proposed in the literature to support the generation of analysis oriented models from design oriented models [27,13]. However, as remarked above, also these modeling frameworks only support the modeling of static systems.

This paper builds on and extends results presented in [12,13,14,15]. With respect to [12,13,14], we extend the intermediate modeling language presented there with new features aimed at modeling dynamically changing systems, and discuss how they can be used for QoS assessment of such systems. We have presented in [15] a preliminary version of this extension. Here, we improve and refine that extension, by giving a more structured definition of the underlying modeling approach, and of the overall transformation path.

The paper is organized as follows. In Section 2 we discuss the core concepts to be considered in the modeling of a dynamically adaptable system. In Section 3, we deal with the first part of the proposed transformation path. We present the intermediate modeling language that represents the core element of our MDD-based approach, and we show how this notation can be used to model the concepts highlighted in Section 2. This discussion serves also as a guide for the definition of a mapping from a source model expressed in some existing design oriented notation to a model expressed in the intermediate language. In Section 4, we deal with the second part of the transformation path, by discussing the mapping of a model expressed in the intermediate language to a suitable analysis oriented model. Throughout Sections 2, 3 and 4, we use a simple example of dynamically adaptable system to show the practical application of the presented ideas. Finally, Section 5 concludes the paper.

2 Core Concepts for the Modeling of Dynamically Adaptable Systems

In this section we identify some key concepts and features for dynamically adaptable systems, which should be adequately represented in the source design oriented model of the system, and then mapped onto the target analysis oriented model.

Before discussing this issue, we point out that we refer to systems that can be represented, at some suitable abstraction level, by a set of interacting components. As environment of such systems we consider the *resources* of the platform where the components are deployed, and the *demand* addressed to the system by external actors (which may be humans, or other systems). Systems architected according to the component-based or service-oriented paradigms belong to this category [9, 26].

Changes occurring in these systems may affect both the system architecture and its environment. For the identification of key concepts concerning the modeling of these changes, we refer to ideas and discussions about this issue appeared in the literature, in particular those presented in [17] and [5,6].

Based on them, we group concepts for change modeling under two categories: the *type of change*, and the *change process* that may affect an adaptable system, as detailed below.

Type of change. First of all, we point out that an occurring change can be the cause or the consequence of a system adaptation. In both cases, a modeling framework for adaptable system should support the modeling of *basic changes* like the addition/removal of components and connectors. These basic changes can be considered as an abstract representation of different kinds of changes which may occur in a system (depending on which are the concrete entities modeled by the component and connector concepts), for example:

- a change in the implementation of a software service by the dynamic binding of its interface to a different implementation;
- a change in the logical interconnection among software modules;
- a change in the mapping of a software module to the underlying platform.

These changes correspond, respectively, to the three kinds of reconfiguration identified in [17], called, respectively, *implementation*, *structural* and *geometric* changes.

Besides the basic operations of adding/removing components and connectors, a modeling framework for adaptable systems should also supports the representation of *complex change* operations, obtained by the composition of basic operations using constructs like sequencing, choice, iteration.

Another feature to be considered concerns the *variability of architectural elements*. It refers to the possibility of modeling the dynamic extension of the set of components/connectors belonging to a system, in opposition to having a fixed set of components/connectors included in the system prior to runtime. The variability could also be partial, where the available types for components and connectors are fixed, but new instances can be dynamically created.

Change process. A modeling framework for adaptable systems should support the modeling of the change *initiation* process, that is the occurrence of events which may

trigger an adaptive change. Such events can be *asynchronous,* when they occur asynchronously with respect to the system control flow (e.g. changes occurring in the application environment, like the addition/removal of resources or the variation in the demand addressed to the system, or the failure of a resource), or *synchronous,* when they occur synchronously with respect to the system control flow (e.g., the execution of a given statement). Such events are called, respectively, *external* and *internal* events in [5,6].

Another feature whose modeling should be possibly supported concerns the freedom degree in the *selection* of the change operation to be performed once an adaptive change has been triggered. This freedom degree can range from a *single pre-defined* adaptive change, to a *constrained selection* from a set of pre-defined changes, to an *unconstrained selection* from an unlimited set of possible changes.

The concepts discussed above concerns, in a sense, "functional" aspects of an adaptable system. Since we are interested in the effectiveness analysis of adaptive systems with respect to performance and dependability requirements, a suitable modeling framework should also support the representation of information about the timing and failure characteristics of the activities occurring in the system. This information should concern both the "normal" system activities and the activities related to changes occurring in the system.

Finally, we point out that the clarity and maintainability of models developed within a modeling framework may benefit from the framework ability in supporting a separation of concerns approach, where the modeling of the normal system operations can be kept separate from the modeling of the system changes.

2.1 An Example of Dynamically Adaptable System

This simple example will be used throughout the paper to show the application of the proposed modeling framework. We consider a *server* that offers a service to its *clients*, giving guarantees about the performance levels of the delivered service. These levels, and the related server and clients obligations and expectations are explicitly stated in *service level agreements* (SLA) negotiated between the server and the clients. A suitable infrastructure is needed to check the fulfillment of what stated in the SLA. In this example, it consists of a *monitor* component, which collects low-level raw performance data, and a *SLAchecker* component, which processes and stores data collected by the monitor, to calculate suitable high level indicators about the actual service quality. Processing and storing these data are resource consuming activities. If the *SLAchecker* is allocated on the same node of the server, they could negatively affect the quality of the service itself, in particular when the server already works under critical conditions (e.g. high load) [1]. To alleviate this problem, we could reduce the monitoring activity, or move the *SLAchecker* to a remote node. However, reducing the monitoring activity could cause a non timely detection of SLA violations, while processing and storing monitoring data on a remote node could cause a non negligible network traffic. A possible architectural solution which tries to balance these opposite issues consists of an *adaptive* SLA checking infrastructure which, according to variations of the server load, reduces the monitoring activity and/or moves the *SLAchecker* to a remote node when the server works under critical conditions, while does the opposite when the server load is light.

With respect to the basic concepts discussed above, two kinds of change may occur in this example system: an environment change (load variation), which may cause the system adaptation, and a consequent adaptation change (variation of the SLAchecker allocation and/or of the monitoring rate).

The initiation process corresponds to the occurrence of events (system load above or below some given thresholds) triggering the adaptive change. These events are asynchronous with respect to the system consisting of the server, monitor and *SLAchecker* components, as they occur independently of the system operations.

Finally, we can note that in this example we are considering a constrained selection among different adaptations, which differ for the load threshold that triggers the adaptation and for the adaptation action (variation of the *SLAchecker* allocation or of the monitoring rate). In the following, for the sake of simplicity, we will consider only adaptation actions based on the *SLAchecker* mobility.

The effectiveness of this adaptive system can be evaluated with respect to its ability in achieving a given performance goal under a variable server load. A possible goal could consist in maintaining both the average server response time and the generated network traffic below given thresholds.

3 A Bridge Model between Design and Analysis Oriented Models of Adaptable Systems

In this section, we discuss the first part of the model transformation path outlined in the introduction. We define the intermediate language that plays a central role in our approach, and we argue that a design model of an adaptable system can be suitably mapped to a model expressed in this language.

The intermediate language we propose builds on a previously defined intermediate language, called KLAPER (Kernel LAnguage for PErformance and Reliability analysis) [13]), which was intended to support the generation of performance or dependability models for "static" systems. To remark its derivation from KLAPER, and the fact that it concerns the modeling of dynamically adaptable systems, we call the new language D-KLAPER.

To take advantage of the current state of the art in the field of model transformation methodologies and tools, D-KLAPER is defined as a MOF (Meta-Object Facility) compliant metamodel, where MOF is the metamodeling framework proposed by the Object Management Group (OMG) for the management of models and their transformations within the MDD approach to software development [24].

We point out that D-KLAPER is not intended to be directly used by system designers. Indeed, in the modeling framework we envisage, D-KLAPER plays a role analogous to that played by the bytecode language in a Java environment. Hence, D-KLAPER provides a purposely minimal set of elementary and abstract concepts and notations. More expressive concepts and notations used by system designers to build their models should then be mapped to D-KLAPER, with the support of automatic model transformation tools. In particular, D-KLAPER is built around these two elementary abstract concepts:

• a software system (and its underlying platform) can be modeled as a set of resources which offer and require services;

• a system change can be modeled by a change in the binding between offered and required services.

Figures 1 and 2 show the structure of the D-KLAPER metamodel which expresses in a more articulated way these basic concepts. We refer to [21] for the complete MOF specification and an up-to-date view of its implementation status.

As shown in Figure 1, different sub-models concur to build a D-KLAPER model:

• *System sub-model (corresponding to Resource and Service metaclasses)*: it provides an abstract representation of the software system and the part of its environment consisting of the platform where it is deployed. This representation is based on the consideration that systems are often structured according to a layered architecture, where components at a given layer actually play the role of resources exploited by upper layers; hence, this overall architecture is modeled as a set of *Resources* which offer *Services* (services, in turn, may require the services of other resources to carry out their own task). A D-KLAPER Resource is thus an abstract modeling concept used to represent both software components and physical resources like processors and communication links.

• *Usage sub-model (corresponding to Workload metaclass)*: it models the part of the system environment consisting of the demand arriving to the system from external "users" (which may be human beings, or other systems), represented by a set of *Workloads*.

• *Trigger sub-model (corresponding to TriggerProcess metaclass)*: it models the occurrence of events in the system and/or its environment, which may trigger an adaptation, and is represented by a set of *Trigger Processes*. When a triggering event occurs, the corresponding adaptation is modeled by the invocation of a suitable AdaptationService (see below).

• *Adaptation sub-model (corresponding to AdaptationService metaclass)*: it models the activities which implement a reconfiguration, whose goal is to adapt the system to an occurred triggering event, and is represented by a set of *Adaptation Services*.

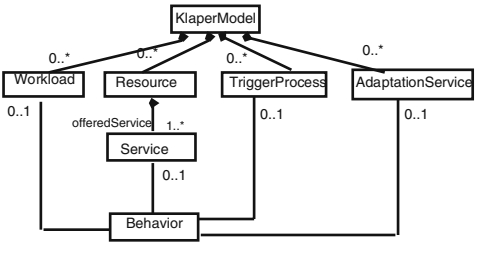


Fig. 1. The D-KLAPER MOF metamodel: overall system modeling

All the metaclasses belonging to these sub-models share the *Behavior* metaclass, which provides a common representation for the dynamics of the activities occurring within each submodel. As shown in Figure 2, a Behavior is modeled as a directed graph of *Steps*. Each Step may be a:

• *Activity* step: it models an activity that may take time to be completed, and/or which may fail before completion, thus providing the basic information for

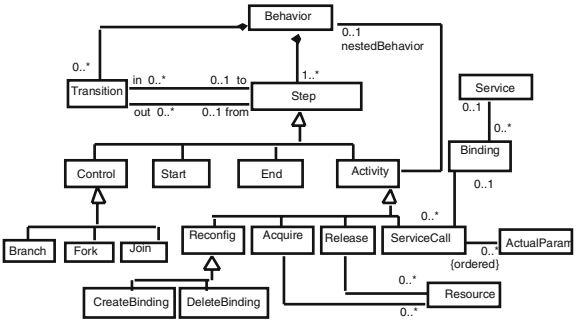


Fig. 2. The D-KLAPER MOF metamodel: behavior modeling

performance or dependability analysis. A special kind of Activity is a *ServiceCall*, which models the request for the service provided by some Resource. A ServiceCall may have *Parameters*, which provide the mean for an abstract representation of the actual parameters that characterize the service requests addressed to some hardware or software resource. The relationship between a ServiceCall and the actual recipient of the call is represented by means of instances of the *Binding* metaclass.

• *Control* step: it models transition rules from step to step, like a branch or a fork/join.

• *Reconfiguration* step: it models a basic change operation, corresponding in D-KLAPER to the addition or removal of a Binding between a ServiceCall step and the corresponding Service. Only the behavior associated with a TriggerProcess or an AdaptationService is allowed to contain Reconfiguration steps.

The semantics of a Behavior is similar to that of other behavioral models like Execution Graphs [29] or UML Activity Diagrams [30]. As D-KLAPER is intended to support the stochastic analysis of performance or dependability attributes, timing, failure and control information associated with steps of a behavior is specified according to a stochastic setting: thus, information like the time to failure or the time to completion of an Activity are defined by suitable random variables; analogously, control information like the selection among alternative transitions, or the number of repetitions of a loop is expressed by suitable probabilities and random variables. D-KLAPER supports the specification of random variables in different ways, ranging from their mean value, to higher order moments, up to the complete distribution. It depends on the target QoS analysis methodology whether this information can be thoroughly exploited (*e.g.*, analytic methodologies for queueing network models usually consider only mean values).

We argue below that most of the concepts and features highlighted in Section 2 can be represented in a D-KLAPER model. Hence, D-KLAPER can be used effectively as intermediate language in a transformation path from a design model to a performance/dependability analysis model.

Type of change. Basic change operations like the addition/removal of components and connectors can be mapped to the creation/deletion of a Binding to a suitable D-KLAPER Service. In this respect, we have given in [12,13,14] examples of Resource instances modeling different types of components and connectors.

Complex change operations can be modeled in D-KLAPER by composing a Reconfiguration step with other steps in more complex "reconfiguration behaviors". Control steps can be used in such behaviors to model the selection or iteration of reconfiguration activities.

With regard to the variability of architectural elements, D-KLAPER does not support the modeling of the dynamic creation/deletion of Resources. However, we can simulate the dynamic creation/deletion of components and connectors in the source design model by defining at the D-KLAPER model level a suitable finite "resource pool". In this way the dynamic creation/deletion of components and connectors can be modeled as the creation/deletion of bindings with resources in the pool. The size of the pool can be selected based on information about the maximum number of components/connectors that can be created in the scenario we want to analyze.

Change process. The dynamics of the occurrence of an asynchronous change initiation event can be modeled in D-KLAPER by the reaching of a given step in the behavior associated with a TriggerProcess. Once this step has been reached, the execution of the corresponding adaptive change can be modeled by a ServiceCall step bound to a suitable instance of an AdaptationService.

For the modeling of synchronous change initiation events, a basic premise is that the software system model is mapped to a D-KLAPER model consisting of a set of resources, each offering one or more services. Hence, the system operations correspond, in a D-KLAPER model, to the execution of the behaviors associated with those services. The synchronous triggering of a reconfiguration caused by the reaching of a given step in a service behavior can be modeled by inserting before (or after) that step a ServiceCall step bound to a suitable instance of AdaptationService.

With regard to the selection among different kinds of change, the pre-defined selection of a single change associated with a trigger can be modeled by the invocation of a single D-KLAPER AdaptationService. The constrained selection from a pre-defined set can be modeled by a D-KLAPER Behavior that uses a suitable combination of Control steps to invoke different AdaptationServices. The unconstrained selection can only be simulated, to some extent, in D-KLAPER, by defining a suitable "pool" of AdaptationService instances, analogously to the simulation of the dynamic addition/deletion of components and connectors.

Change cost. As pointed out above, each D-KLAPER Activity step includes timing and failure information that can be exploited to support the analysis of the system performance or dependability. Moreover, ServiceCall steps can be used to model the request for external resources (whose services, in turn, can require time to be completed, or may fail). These steps can be included in the behavior associated with a TriggerProcess or an AdaptationService, thus providing the basis for an analysis of the cost (impact on performance or dependability) of system changes, besides their possible benefits.

Separation of concerns. From the discussion above, it is clear that D-KLAPER has been designed according to the separation of concern principle. Indeed, the System and Usage sub-models are intended to support the modeling of a basically static system, while the Trigger and Adaptation sub-models are intended to model the dynamics of the system adaptation. This separation is also enforced by the constraint that the

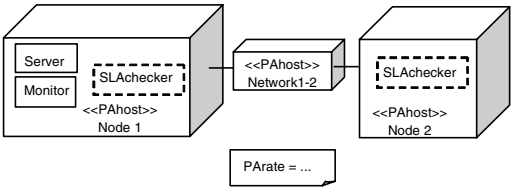


Fig. 3. Deployment Diagram of the example system

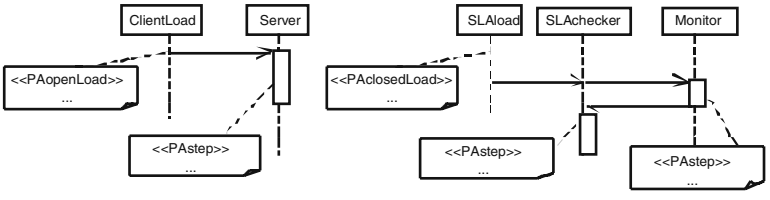


Fig. 4. Sequence Diagram for the example system

basic change operations (addition or removal of a Binding between a ServiceCall step and the corresponding Service) can only be used in the Behavior associated with a Trigger process or an AdaptationService.

3.1 D-KLAPER Modeling of the Example System

Figures 3 and 4 depict a possible UML model of the example system, with two possible allocations of the SLAchecker. The figures also partially show some annotations (based on the SPT/MARTE profiles [31], used to express performance related information). This model only provides a static vision of the system. We do not model the dynamics of the adaptation described in Section 2.1, as we are not aware of a commonly agreed UML notation for this.

Figures 5 and 6 partially show the corresponding D-KLAPER models, expressed in textual form. In particular, Figure 5 shows the D-KLAPER models of the two platform elements *Node_1* and *Network_1-2*, derived from the UML model of Figure 3. *Node_1* is modeled as a D-KLAPER Resource. The service it offers has a formal parameter (*n_op*) that can be used to specify the number of requested operations when this service is invoked. This information, together with the value of the *speedAttr* attribute allows determining the time needed to complete a given invocation of this service. *Network_1-2* is modeled analogously.

On the other hand, Figure 6 shows the D-KLAPER model of the *Server* software component and its *ClientLoad* workload, derived from the UML model in Figure 4. *Server* is modeled as a Resource. The service it offers has two formal parameters (*q_size* and *ans_size*) that can be used to specify the size of the query sent by the client and of the corresponding answer (under the assumption that this is the only relevant information for performance analysis). The behavior of this service consists of the invocation of a service offered by a cpu-type resource. Its actual parameter *nq1*

```
Resource Node_1
{       type = cpu
        schedulingPolicy = ps
        capacity = *
        offeredService = service_1
        description = "a processing node"
}

Service service_1
{       formalParam = [n_op : double]
        speedAttr = 10^9 /* number of operations per
                    unit time; time unit is assumed
                    = 1 sec */
        description = "processing service offered by
                    the resource Node_1"
        Behavior
        { Start st1 ->
          Activity a1
           { ExecTime =  mean n_op/speedAttr; } ->
          End e1
        }
}
```

```
Resource net_1-2
{       type = network
        schedulingPolicy = ps
        capacity = *
        offeredService = service_1-2
        description = " a network connecting Node_1
                    and   Node_2"
}

Service service_1-2
        {formalParam = [n_bytes : double]
        speedAttr = 10^6 /* number of bytes
                    transmitted per unit  time; time
                    unit is assumed = 1 sec */
        description = " transmission  service offered
                    by net_1-2"
        Behavior
        { Start st1 ->
          Activity a1
         { ExecTime = mean n_bytes/speedAttr; } ->
         End e1
        }
}
```

Fig. 5. D_KLAPER models of platform resources

```
Resource Server_1
{       type = Server
        schedulingPolicy = FIFO
        capacity = *
        offeredService = serviceS_1
        description = " a server"

}

Service serviceS_1
{       formalParam = [q_size :double]
        formalParam = [ans_size : double]
        description = " service offered
          by the server Server_1"
Behavior
        { Start st1 ->
          Call cl1
            { resourceType = cpu
              isSynch = true
            actualParam  = mean nq1
            } ->
         End
        }
}
```

```
Workload wl_1
{       type = OPEN
        arrivalProcess = Poisson lambda1
        Behavior
        { Start st1 ->
          Call cl1
            { resourceType = Server
              actualParam =  mean qo
              actualparam = mean qi
            } ->
         End
        }
}
```

Fig. 6. D_KLAPER models of a SW component and its workload

can be used to specify the computational load (number of operations) to calculate the answer for a client request. As we are in a stochastic setting,*nq1* is actually a random variable. For the sake of simplicity, in this example we specify random variables by their mean value (as indicated in fig. 6). This service invocation must be bound to the service offered by a suitable resource. In this example, it will be bound to the service offered by the *Node_1* resource.

Thus, Figures 5 and 6 shows the modeling of both software components and hardware nodes by the D-KLAPER Resource concept.

Instead, the *ClientLoad* represented in the UML model of Figure 4 is modeled in Figure 6 as a D-KLAPER Workload. Its semantics consists in the generation, accord-

ing to a Poisson process, of instances of its Behavior. Each instance consists of an invocation of the service offered by a Server-type resource, with suitable actual parameters (which again are specified as mean values). In this example, this invocation will be bound to the service offered by the *Server_1* resource, by means of an instance of the Binding metaclass.

Explicit transformation rules to get these D-KLAPER models from the source UML models can be quite easily envisaged. Some definitions of these rules can be found in [12,13,14].

Even if we do not have given a UML model for the dynamics of the system changes, we give below examples of D-KLAPER models that could reasonably be derived from a source high-level model. We recall that in this example the adaptation is triggered by some sensible variation of the load addressed to *Server*. Figure 7(a) shows a possible D_KLAPER model of the adaptation triggering. It has been built under the assumption that the following information has been extracted from the source design model:

- the possible load values (arrival rates of the client requests) can be discretized using three different values of the Poisson process parameter representing the arrival of client requests ($\lambda 1$, $\lambda 2$ and $\lambda 3$);
- the load value changes cyclically, from $\lambda 1$ to $\lambda 2$ to $\lambda 3$ and then back to $\lambda 1$, with a 20 minutes average interval between two consecutive variations.

Each load value can be modeled by a different instance of a D-KLAPER Workload, which we name *wl_1*, *wl_2* and *wl_3,* respectively (*wl*_1 is shown in Figure 6). They share the same structure, and differ only in the value of the Poisson parameter. As shown in Figure 7(a), the behavior of the trigger process *TP_1* periodically binds the service invocations generated by each of these instances to the service offered by *Server* (immediately after deleting the binding with the previous workload instance). Activities *a1*, *a2* and *a3*, whose mean completion time is 20 minutes (expressed in seconds), are used to model the time interval between two load changes. Moreover, the behavior of *TP_1* includes some "placeholder" activities (*reconf1*, *reconf2*, *reconf3*) where we can model the invocation of suitable AdaptationServices.

Figure 7(b) shows a possible AdaptationService (*AS_1*) for this example. According to the discussion in Section 2.1, its behavior models the transfer of the *SLAchecker* from *Node_1* to *Node_2*, which should reasonably happen when the load on *Node_1* exceeds some threshold. In the figure, we assume that the *SLAchecker* and the *Monitor* have been modeled by D-KLAPER Resources named *SLA-C* and *MON_1*, with respective offered services *serviceSLA_1* and *serviceM_1*.

The operations available in D-KLAPER are elementary, so to model the behavior of *AS_1* we need a sequence of such operations. Operations *db1* and *cb1*, respectively, delete the direct binding between *serviceSLA_1* and *serviceM_1*, and bind *serviceSLA_1* to the service offered by a *RPC* Resource (not shown here, modeling a Remote Procedure Call connector) which provides a remote connection to *serviceM_1* (the behavior of the service offered by *RPC* will include appropriate invocations of network and cpu services). On the other hand, operations *db2* and *cb2* model the transfer of the computational load generated by the *SLAchecker* from *Node_1* to

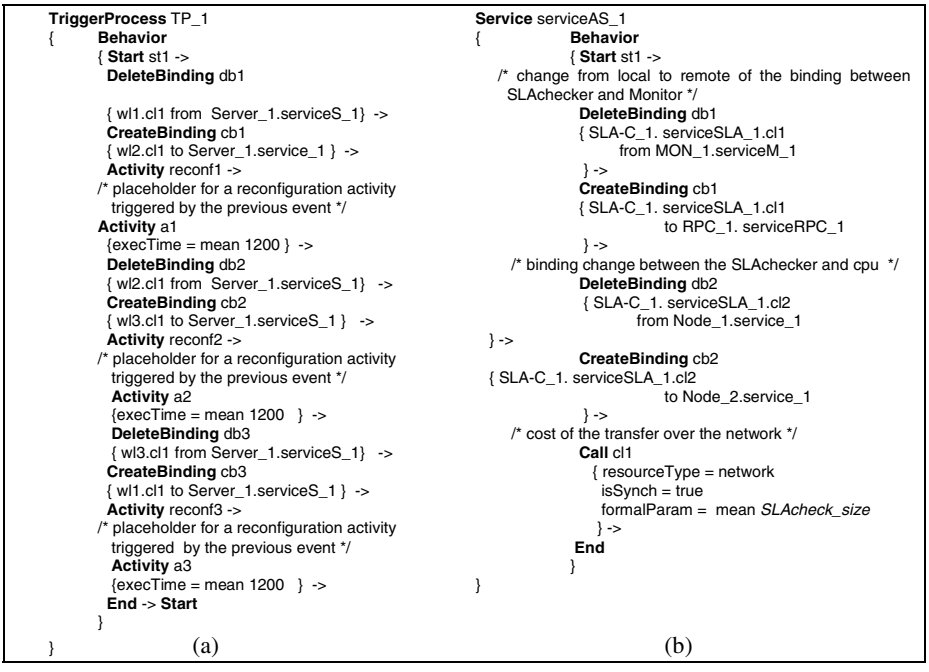


Fig. 7. D-KLAPER models of a triggering process and an adaptation service

Node_2. Finally, operation *cl1* is an invocation of a network service modeling the transfer of the *SLAchecker* (at least, of its state) over the network.

We point out that in this way we have modeled not only the long term effects of the adaptation (change in the target resources for the operations of *SLAchecker*), but also the cost of such adaptation (network traffic and time spent to get the new configuration).

4 Building an Analysis Oriented Model from the Bridge Model

In this section, we discuss the second part of the proposed model transformation path, concerning the generation of an analysis oriented model from a D-KLAPER model.

We first discuss some general issues concerning this problem, independently of a specific target notation. Then, we outline the generation of models expressed in a specific notation (Markov Processes) and finally we use the example already used in the previous sections to show a concrete application of these ideas.

4.1 Approaches to the Generation of an Analysis Model

First of all we note that, abstracting from the different parts of the source model from which the elements of a D-KLAPER model have been derived, an overall D-KLAPER model basically consists of a set of Behavior instances, interconnected by means of Binding instances. Each Behavior instance models a flow of activities

which take time to complete and may fail before completion (and may synchronize with other Behavior instances through ServiceCall instances). Hence, the variety and richness of constructs and concepts which were present in the source model have been abstracted into these more elementary concepts, which are closer to those generally used in analysis oriented notations.

Several notations (and corresponding analysis methodologies and tools) exist which can be used to express a suitable target analysis oriented model derived from a D-KLAPER model. Independently of any particular notation, two different approaches can be basically followed in the generation of such a model: a monolithic approach and a hierarchical approach [16, 29].

A monolithic approach leads to the construction of an overall flat model, which encompasses all the activities and events occurring in the system. This model basically gives an "exact" representation of the dynamics of the source model, but could be too complex to be analyzed (because of the state space explosion, and of numerical problems caused by the difference in the time scale for the occurrence of different classes of events).

A hierarchical approach may overcome these problems, at the cost of introducing some intrinsic approximation. It is based on the consideration that events concerning the normal system operations (e.g. service requests, service completions) occur at a much higher rate than adaptation triggering events (and consequent adaptation actions). This leads to a two-level hierarchy of models, with the first level consisting of a family of so-called *normal operation models*, each modeling the system operation in one of the possible system configurations, and the second level consisting of an *adaptation model*, which concerns the modeling of changes in the system configuration. The two levels are combined by including in the latter model results (e.g. performance values) obtained from the solution of the former models. Because of the different time scale between the two levels, it can be assumed with reasonable approximation that most of the time the system is in steady state in a given configuration between two consecutive configuration changes. This reduces the complexity of each model to be generated in the hierarchy, and allows to solve it independently of the others. In the following we only discuss the hierarchical approach.

The algorithm in Figure 8 summarizes the steps to be performed to implement in our framework the hierarchical approach, independently of the selected target notations used to express the models in the two hierarchy levels.

Step 1 in Figure 8 is a preliminary step, consisting in the generation of D-KLAPER models of all the possible "stable" system configurations. By stable configuration we mean a configuration reached by the system at the end of the adaptation triggered by some event. This step can be carried out by "executing" the Behavior instance(s) belonging to the Trigger sub-model *T* (which in turn may cause the activation of the AdaptationService instance(s) belonging to the Adaptation sub-model *A*). This execution leads to the generation of instances *(S1+U1)*, ..., *(Sm+Um)* of the System and Usage sub-models *S* and *U*, where the different set of Bindings in each instance models a different system configuration. Since we assume a finite number of Resource and Workload instances in a D-KLAPER model, there is a finite number of possible configurations.

Steps 2 and 3 correspond to the construction of the first level of the hierarchical model. At step 2 we use each *(Si+Ui)* instance built at step 1 to derive from it a

Input: a D-KLAPER model consisting of:
• a System sub-model *S* and a Usage sub-model *U*, representing the system under "normal operation conditions" (with partially specified Bindings);
• a Trigger sub-model *T* representing the occurrence of adaptation triggering events (with corresponding invocations of AdaptationService instances);
• an Adaptation sub-model *A*, whose AdaptationServices *AS1*, ..., *ASk* model the creation/deletion of Bindings, and (possibly) the associated resource usage.
Output: a two-level hierarchical model.
Algorithm:
1. use *T* and *A* to derive from *S* and *U* a set of models *(S1+U1)*, ..., *(Sm+Um)*, with completely specified Bindings, corresponding to all the possible "stable" system configurations;
2. for each *(Si+Ui)* **in** {*(S1+U1)*, ..., *(Sm+Um)* } **do**
- derive from *(Si+Ui)* a performance/dependability model *PNi*;
- solve *PNi* to get performance/dependability values *ri*

endfor
3. for each *ASj* **in** { *AS1*, ..., *ASk* } **do**
- derive from *ASj* a performance/dependability model *PRi*;
- solve *PRi* to get performance/dependability values *ci* (reconfiguration "cost")

endfor
4. derive from *(S1+U1)*, ..., *(Sm+Um)*, *T*, *r1*, ..., *rm* and *c1*, ..., *ck* a model of the overall system performance/dependability and/or a model of the overall adaptation cost

Fig. 8. Transformation algorithm from a D-KLAPER model to a two-level analysis model

performance/dependability model *PNi*. The solution of *PNi* provides estimates of the system performance/dependability indices when the system operates in the corresponding configuration (in steady state conditions, according to the discussion above).

At step 3 we use the adaptation services *AS1*, ..., *ASk* belonging to the Adaptation sub-model *A* to derive from them corresponding performance/dependability models *PR1*, ..., *PRk*. The solution of these models allows the estimation of the cost (in terms of elapsed time and consumed resources) of each system adaptation.

Finally, step 4 corresponds to the construction of the second level of the hierarchy, where information coming from the previous steps is used to build a model of the overall performance/dependability which can be achieved thanks to the adaptation, and also of the overall incurred cost.

We assume that all these steps are carried out with the support of automatic model transformation tools. The target model for the transformations at step 1 is again a D-KLAPER model, while the target models for the transformations at steps 2, 3 and 4 are to be expressed into suitably selected analysis-oriented notations. The selection of these notations depends on the analysis goal (and also the availability of adequate solution tools).

Target notations that can be used at steps 2 and 3 are, for example, Queueing Networks [23](for performance analysis) or Markov Processes (for dependability analysis) [16]. In this respect, we have given transformations rules from KLAPER to Extended Queueing Networks, Discrete Time Markov Processes and Layered Queueing Networks in [12,13,14].

On the other hand, at step 4 we need a target notation able to express the dynamics of the occurrence of adaptation triggering events, and also able to embed the solutions calculated at steps 2 and 3. In the next section we present an example of such a notation, and discuss the implementation of the mapping to it.

4.2 Generation of a Semi-markov Reward Model

Markov or Semi-Markov reward models [16] are state-based models where a reward is associated with each state and/or transition. Within a hierarchical modeling approach, *states* can be used to model the possible system configurations; *transitions* to model the system evolution; *rewards* associated with states to represent the system performance/dependability in the corresponding configurations; *rewards* associated with transitions to model the cost of reconfigurations (alternatively, the cost of reconfigurations may also be modeled by the reward associated with some "reconfiguration" states.

These models can be used to calculate probabilistic *reward measures* like the reward at some time instant t, or the total reward accumulated over some time interval $[0, t]$, with t possibly going to infinity if we want to consider steady-state measures. Depending on which kind of reward we associate to the model states and transitions, these reward measures correspond to some overall performance/dependability measure for a reconfigurable system. For example, if the reward associated with a state is the system throughput in that state, then the reward accumulated in $[0, t]$ is the number of service requests processed in that interval, while this accumulated reward divided by t is the average throughput in the same interval. Alternatively, if the reward associated with a state is the system availability in that state, then the reward accumulated in $[0, t]$ divided by t is the average availability in that interval.

A vast literature exists about this modeling approach. We refer to [16] (in particular chapts. 3, 4, 9) for further details and references.

Generating a SMR model from a D-KLAPER model actually means specifying:

- the *set of states*;
- the *sojourn time* in each state, before a transition occurs;
- the *transition probabilities* to other states;
- the *reward* associated with each state/transition.

The set of states is defined by a one-to-one correspondence with each possible configuration *(Si+Ui)* generated at the step 1 of the Figure 8 algorithm. The possible one-step transitions from a state to other states can be determined at the same time: the target states correspond to the set of configurations that can be generated by the invocation of an AdaptationService *ASj*, when the system is in the configuration corresponding to the source state.

The sojourn time in a state, and the probabilities associated with transitions from that state, can be derived from information obtained from the Trigger sub-model. In particular, the *internalExecutionTime* attribute of each step in the behavior of the Trigger model, and the *probability* attributes of Transition steps in the same behavior can be used to determine the timing (defined by a random variable) and the probability for the occurrence of a trigger. This information can be used to determine the overall residence time and transition probabilities for the SMR states.

Finally, regarding the SMR rewards, the performance/dependability models generated at step 2 of the algorithm are associated with the SMR states modeling the corresponding system configuration, while the performance/dependability models generated at step 3 are associated with the SMR transitions (or states) that model the occurrence of a reconfiguration.

We point out that a MOF metamodel for SMR models should be defined to exploit MOF-based transformation methodologies and tools for the implementation of the D-KLAPER to SMR mapping outlined above. Defining such a metamodel is a simple task and an example of this metamodel can be found in [15]. The SMR model obtained as result of this transformation process can then be solved to calculate a suitable reward measure, using existing tools (e.g. [28]).

4.3 Analyzing the Example System Effectiveness

As discussed in Section 2.1, our goal is the analysis of the effectiveness of a class of adaptation policies based on the mobility of the *SLAchecker* component. Policies in this class differ for the load level used to decide whether the *SLAchecker* should move from *Node_1* to *Node_2* or vice versa. Having assumed in Section 3.1 three different levels ($\lambda 1$, $\lambda 2$ and $\lambda 3$, with $\lambda 1<\lambda 2<\lambda 3$) for the load level of the *Server* component, two possible policies in this class could be defined as follows:

- *policy 1*: - *SLAchecker* moves from *Node_1* to *Node_2* when load $\geq \lambda 2$
  - *SLAchecker* moves from *Node_2* to *Node_1* when load $< \lambda 2$
- *policy 2*: - *SLAchecker* moves from *Node_1* to *Node_2* when load $\geq \lambda 3$
  - *SLAchecker* moves from *Node_2* to *Node_1* when load $< \lambda 3$

We evaluate their effectiveness with respect to the ability to keep the overall network traffic and the *Server* component response time below given thresholds. For this purpose, we also compare them with two "static" policies, named *no-adapt_1* and *no-adapt_2*, where the *SLAchecker* has a fixed location at *Node_1* or *Node_2*, respectively.

In Figure 7(a) we have presented a possible D-KLAPER Trigger sub-model for the occurrence of load changes. This sub-model can be specialized for *policy 1* by replacing the "placeholder" activity *reconf1* with a ServiceCall bound to the Adaptation Service *AS_1* shown in Figure 7(b) (which models the transfer of *SLAchecker* to *Node_2*), and replacing the "placeholder" activity *reconf3* with a ServiceCall bound to an Adaptation Service *AS_2* (not shown in Figure 7) which models the transfer of *SLAchecker* to *Node_1*.

Analogously, the model of Figure 7(a) can be specialized for *policy 2* by replacing the "placeholder" activities *reconf2* and *reconf3* with ServiceCalls bound to Adaptation Services *AS_1* and *AS_2*, respectively.

The D-KLAPER models obtained in this way, together with the System and Usage sub-models (partially shown in Figures 5 and 6) are the input for the algorithm in Figure 8.

At step 1 the algorithm generates the set of D-KLAPER models corresponding to stable system configurations. In our example, each configuration is characterized by a specific load level and *SLAchecker* allocation. Hence, we denote each configuration with the pair (i,j), $1 \leq i \leq 3$, $1 \leq j \leq 2$, where i refers to the load level and j to the *SLAchecker* allocation. Each D-KLAPER model (i,j) obtained in this way has a different set of Bindings between ServiceCall steps and corresponding Services. For example, in configurations (1,-) (see Figure 6) *serviceS_1* is bound to the ServiceCall step *cl1* of the *wl_1* workload, while in configurations (2,-) *serviceS_1* is bound to the analogous ServiceCall step *cl1* of the *wl_2* workload (not shown in Figure 6).

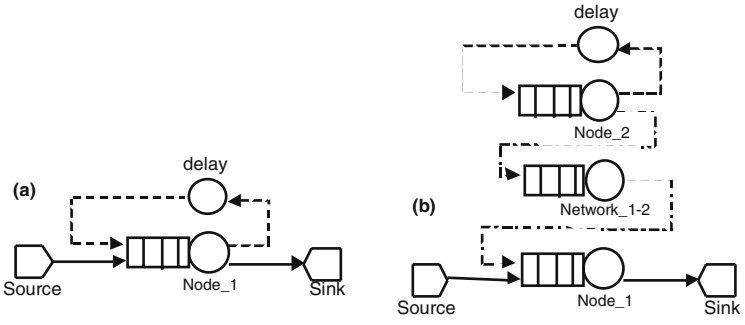


Fig. 9. EQN models: (a) configurations (-, 1); (b) configurations (-, 2)

The possible configurations according to *policy 1* are (1,1), (2,2), (3,2), while for *policy 2* they are (1,1), (2,1), (3,2).

For each D-KLAPER model (*i*,*j*), we have to derive (step 2 of the algorithm) corresponding performance models. We assume as target model an Extended Queueing Network (EQN) [23]. Transformation rules from D-KLAPER to EQN have been presented in [13]. Using these rules we get the models depicted in Figure 9. In both models 9(a) and 9(b) we can see a closed workload (dashed line) corresponding to the periodic activity of the *SLAchecker*, and an open workload (continuous line) corresponding to the arrival of requests to the *Server*. The 9(b) model has two additional service centers with respect to the 9(a) model because of the allocation of the *SLAchecker* to a remote node and the consequent network traffic. We may actually have three instances of both 9(a) and 9(b) models, corresponding to the three different values of the open workload arrival rate. Solving these models, we get a prediction for the steady state *Server* response time in the different configurations.

Analogously, we can get models of the adaptation cost (step 3 of the algorithm) by deriving suitable performance models from the D-KLAPER adaptation model (see Figure 7(b)). In our example, this cost is caused by the transfer of *SLAchecker* from *Node_1* to *Node_2* and vice versa.

The models built at steps 2 and 3 correspond to the first level of the overall hierarchical model. Then (step 4 of the algorithm), we have to build the SMR model that we have selected as second level model. The states of the SMR models for the two considered policies correspond to the configurations identified at step 1. The sojourn time in each state is derived from information expressed in the Trigger sub-model (according to what shown in Figure 7(a), we assume an exponentially distributed sojourn time with mean equal to 1200 sec.). Reward rates associated with states and/or transitions can be used to embed in this second level model the results obtained from the first level models. In particular, to calculate the overall average response time, we associate with each state a reward equal to the response time in the corresponding configuration. To calculate the average traffic, we associate with each state a reward equal to the network traffic (if present) caused by the interaction between *SLAchecker* and *Monitor*, and with each transition a reward equal to the network traffic (if present) caused by the transfer of the *SLAchecker* (we remark that this latter reward

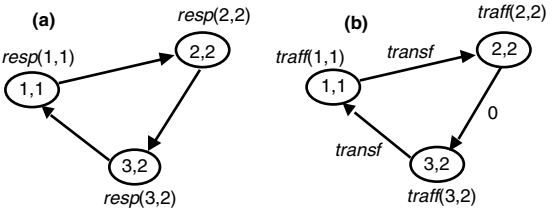


Fig. 10. SRMs for policy 1: (a) response time; (b) network traffic

actually represents the cost to be paid for each adaptation action). Figure 10 depicts the two SRMs obtained in this way for *policy 1*. For the SRM in Figure 10(a), the reward *resp*(*i*,*j*) associated with each state (*i*,*j*) denotes the response time in that configuration. *resp*(*i*,*j*) is calculated using the EQN of Figure 9(a) or 9(b) when *i*=1 or *i*=2, respectively, with the arrival rate of the open workload equal to λj.

For the SRM in Figure 10(b), the reward *traff*(*i*,*j*) associated with each state (*i*,*j*) denotes the network traffic between the *SLAchecker* and *Monitor* when the system is in the corresponding configuration. Hence, *traff*(1,1)=0, while *traff*(2,2)=*traff*(3,2)>0. The reward *transf* associated with the transitions from (3,2) to (1,1) and from (1,1) to (2,2) is calculated from the *SLAchecker* size, while the reward associated with the transition from (2,2) to (3,2) is equal to 0 as this transition models a change in the load level which does not cause a *SLAchecker* transfer, according to *policy 1*. The SRMs for *policy 2* can be obtained analogously.

Solving these models, we can evaluate the effectiveness of the two policies for the modeled scenario.

We assume the following values of the system parameters (average values):

Node_1 and *Node_2* speed: $1x10^9$ op./sec; $\lambda 1 = 50$; $\lambda 2 = 100$; $\lambda 3 = 150$;
Server service demand: $6.6x10^6$ op.; *Monitor* service demand: $1x10^6$ operations;
SLAchecker service demand: $6x10^6$ op.; *Server-Monitor* traffic: 7 Kbytes/sec;
closed workload delay: 1 sec.

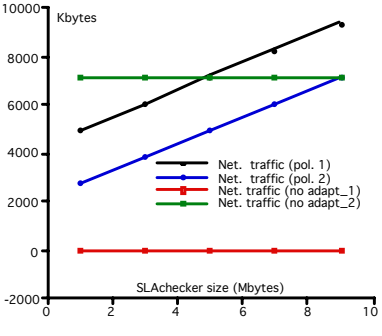


Fig. 11. Network Traffic

With these parameter values, we get the following values for the overall average response time of the *Server* component under the considered adaptation and static policies:

- *policy 1*: average response time = 0.3722 sec.
- *policy 2*: average response time = 0.3724 sec.
- *no-adapt_1*: average response time = 2.8402 sec.
- *no-adapt_2*: average response time = 0.3721 sec.

The values for the network traffic are instead reported in Figure 11, as a function of the *SLAchecker* size. We see that the *no-adapt_1* policy causes zero network traffic (all the interactions between *SLAchecker* and *Monitor* are local), but with a considerably worse response time. Conversely, the *no-adapt_2* policy guarantees the best response time, as the *SLAchecker* never interferes with the client load at *Node_1*, but causes a high network traffic. For the two adaptation policies, *policy 2* has a response time quite close to the best, but at lower cost in terms of network traffic with respect to *no-adapt_2*. Instead, the other adaptation policy (*policy 1*) causes a higher network traffic (in some cases even worse than *no-adapt_2*), without a sensible improvement of the response time.

Depending on the target thresholds for the average response time and network traffic, these results can be used to determine which policy is more effective in meeting the target.

5 Conclusions

In this paper we have presented a Model-Driven approach whose goal is to support the QoS assessment of self-adaptable systems. Our approach builds on the existence of intermediate modeling languages and extends one of them, to capture the core features (from a performance/dependability viewpoint) of a dynamically adaptable architecture model.

A potential drawback of the methodology we have presented in section 4 to generate an analysis model from the intermediate model, is that it requires the enumeration of all the configurations of the system model. This could lead to a kind of “state explosion” problem. In that case, a different methodology (possibly non state-based, as the proposed one) could be more adequate. How to devise such a methodology is a non trivial issue, and we leave it for future work.

At present, we are working on the automation of the proposed approach, since this represents a key point for its successful application and complete validation by applying it to industrial case studies.

Acknowledgements

Work partially supported by the Italian Project ArtDeco (FIRB RBNE05C3AH) and by the project Q-ImPrESS (215013) funded under the European Union’s FP7.

References

1. Agarwala, S., Chent, Y., Milojicic, D., Schwan, K.: QMON: QoS- and Utility-Aware Monitoring in Enterprise Systems. In: 3rd IEEE Int. Conference on Autonomic Computing (ICAC 2006), pp. 124–133 (2006)
2. Allen, R., Douence, R., Garlan, D.: Specifying and Analyzing Dynamic Software Architectures. In: Astesiano, E. (ed.) FASE 1998. LNCS, vol. 1382, p. 21. Springer, Heidelberg (1998)
3. Ardagna, D., Ghezzi, C., Mirandola, R.: Rethinking the Use of Models in Software Architecture. In: Becker, S., Plasil, F., Reussner, R. (eds.) QoSA 2008. LNCS, vol. 5281. Springer, Heidelberg (2008)
4. Atkinson, C., Kühne, T.: Model-Driven Development: A Metamodeling Foundation. IEEE Software 20(5), 36–41 (2003)
5. Bradbury, J.S., Cordy, J.R., Dingel, J., Wermelinger, M.: A survey of self-management in dynamic software architecture specifications. In: Proceedings of WOSS 2004, Newport Beach, CA, USA (October 2004)
6. Bradbury, J.S.: Organizing definitions and formalisms for dynamic software architectures. Tech. Report 2004-477, Queens' University (2004)
7. Bracchi, P., Cukic, B., Cortellessa, V.: Performability Modeling of Mobile Software Systems. In: ISSRE 2004, pp. 77–88 (2004)
8. Cheng, B.H.C., Giese, H., Inverardi, P., Magee, J., de Lemos, R.: 08031 – Software Engineering for Self-Adaptive Systems: A Research Road Map. In: Software Engineering for Self-Adaptive Systems, Dagstuhl Seminar Proceedings (2008)
9. Crnkovic, I., Larsson, M. (eds.): Building Reliable Component-Based Software Systems. Artech House (2002)
10. Di Marco, A., Mirandola, R.: Model Transformation in Software Performance Engineering. In: Hofmeister, C., Crnković, I., Reussner, R. (eds.) QoSA 2006. LNCS, vol. 4214, pp. 95–110. Springer, Heidelberg (2006)
11. Garlan, D., Monroe, R., Wile, D.: ACME: Architectural Description of Component-Based Systems. In: Leavens, G.T., Sitaraman, M. (eds.) Foundations of Component-Based Systems. Cambridge University Press, Cambridge (2000)
12. Grassi, V., Mirandola, R., Sabetta, A.: A Model Transformation Approach for the Early Performance and Reliability Analysis of Component-Based Systems. In: Gorton, I., Heineman, G.T., Crnković, I., Schmidt, H.W., Stafford, J.A., Szyperski, C., Wallnau, K. (eds.) CBSE 2006. LNCS, vol. 4063, pp. 270–284. Springer, Heidelberg (2006)
13. Grassi, V., Mirandola, R., Sabetta, A.: Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach. Journal of Systems and Software 80(4), 528–558 (2007)
14. Grassi, V., Mirandola, R., Randazzo, E., Sabetta, A.: KLAPER: An Intermediate Language for Model-Driven Predictive Analysis of Performance and Reliability. In: Rausch, A., Reussner, R., Mirandola, R., Plášil, F. (eds.) The Common Component Modeling Example. LNCS, vol. 5153, pp. 327–356. Springer, Heidelberg (2008)
15. Grassi, V., Mirandola, R., Sabetta, A.: A model-driven approach to performability analysis of dynamically reconfigurable component-based systems. In: WOSP 2007, pp. 103–114 (2007)
16. Haverkort, B.R., Marie, R., Rubino, G., Trivedi, K. (eds.): Performability Modelling: Techniques and Tools. J. Wiley and Sons, Chichester (2001)
17. Hofmeister, C.: Dynamic reconfiguration of distributed applications, PhD dissertation, Dept. of Computer Science, University of Maryland (1993)

18. Irmert, F., Fischer, T., Meyer-Wegener, K.: Runtime Adaptation in a Service-Oriented Component Model. In: SEAMS 2008 (2008)
19. Kacem, M.H., Miladi, M.N., Jmaiel, M., Kacem, A.H., Drira, K.: Towards a UML profile for the description of dynamic software architectures. In: COEA 2005, pp. 25–39 (2005)
20. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer 36(1) (2003)
21. KLAPER project website, http://klaper.sf.net
22. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: FOSE 2007: Future of Software Engineering, Washington, DC, USA, May 23-25 (2007)
23. Lavenberg, S.S.: Computer Performance Modeling Handbook. Academic Press, New York (1983)
24. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture (TM): Practice and Promise. Addison-Wesley Object Technology Series (2003)
25. Oreizy, P., Medvidovic, N., Taylor, R.N.: Runtime software adaptation: framework, approaches, and styles. In: ICSE Companion 2008, pp. 899–910 (2008)
26. Papazoglou, M., Georgakopolous, D.: Service-Oriented Computing. Communication of the ACM 46(10)
27. Petriu, D.B., Woodside, M.: An intermediate metamodel with scenarios and resources for generating performance models from UML designs. Software and Systems Modeling 2, 163–184 (2007)
28. SHARPE, http://www.ee.duke.edu/~kst/
29. Smith, C.U., Williams, L.: Performance solutions: A Practical Guide to Creating Responsive, Scalable Software. Addison Wesley, Reading (2002)
30. UML 2.0 Superstructure Specification, OMG Adopted Specification ptc/03-08-02, http://www.omg.org/docs/ptc/03-08-02.pdf
31. UML Profile for Schedulability, Performance, and Time Specification, OMG Adopted Specification ptc/02-03-02, http://www.omg.org/docs/ptc/02-03-02.pdf
32. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: ICSE 2006, pp. 371–380 (2006)

Automatic Generation of Runtime Failure Detectors from Property Templates

Mauro Pezzè[1] and Jochen Wuttke[2]

[1] University of Milan Bicocca, Italy, and
University of Lugano, Switzerland
mauro.pezze@unisi.ch
[2] University of Lugano, Switzerland
wuttkej@lu.unisi.ch

Abstract. Fine grained error or failure detection is often indispensable for precise, effective, and efficient reactions to runtime problems. In this chapter we describe an approach that facilitates automatic generation of efficient runtime detectors for relevant classes of functional problems. The technique targets failures that commonly manifest at the boundaries between the components that form the system. It employs a model-based specification language that developers use to capture system-level properties extracted from requirements specifications. These properties are automatically translated into assertion-like checks and inserted in all relevant locations of the systems code.

The main goals of our research are to define useful classes of system-level properties, identify errors and failures related to the violations of those properties, and produce assertions capable of detecting such violations. To this end we analyzed a wide range of available software specifications, bug reports for implemented systems, and other sources of information about the developers intent, such as test suites. The collected information is organized in a catalog of requirements-level property descriptions. These properties are used by developers to annotate their system design specifications, and serve as the basis for automatic assertion generation.

1 Introduction

Software engineering research has developed numerous paradigms, methodologies, and technologies to improve the quality of software systems. Despite the progress that has been made, software systems fail while deployed and running in the field. Research in fault tolerant systems has produced several techniques that can mask some classes of critical errors in a way that externally visible failures do not occur[1].

[1] The community working on fault tolerant systems usually distinguishes between errors, failures and faults. In this taxonomy an *error* is a system state, or a sequence of system states that deviate from the correct state. A *failure* manifests when an error becomes visible on the external interface of a component or system. A *fault* is the cause of an error. In software systems the boundaries between components are not well defined (is the boundary on the method, class, package, or system level?), and the distinction between *error* and *failure* is difficult to make. Hence, in this chapter we use the term *failure* to mean a system state that deviates from the correct state.

B.H.C. Cheng et al. (Eds.): Self-Adaptive Systems, LNCS 5525, pp. 223–240, 2009.

Most of these classical fault tolerant techniques have been developed for safety critical applications and rely on expensive design approaches, such as redundant implementation, that may not fit well other classes of software systems [1,2].

In recent years, research in different fields has converged on the definition and deployment of *self-adaptive* and *autonomic* software systems, that is systems able to autonomously recover from problems at runtime [3]. *Self-adaptation* refers to various classes of problems and techniques, and is specialized in different self–* techniques depending on the classes of problems. For example, *self-configuring* systems are able to assemble and configure themselves based on a description of high-level goals, *self-protecting* systems are capable of taking autonomous action when threatened by attempts to violate their security and safety guarantees. In this chapter we discuss *self-healing* systems, that is systems that can recover from functional failures of their constituent components.

Building on key ideas expressed by Kephart and Chess, most self-adaptive systems rely on a variant of the "autonomic cycle" [3]. In their model, an autonomic element, that is a component or a whole system, is under the control of an autonomic manager, which *monitors* and *analyzes* the execution of the element, and in the case of problems *plans* and *executes* changes to the systems configuration.

Most research on self-healing systems has addressed issues directly relating to adaptability, that is the *planning* and *execution* phases of the autonomic cycle. Such work usually assumes suitable monitoring and analysis mechanisms exist, instead of treating this as a research problem. Our research tackles the problem of precise failure detection, and thus develops techniques for the monitoring phase of self-healing systems. Even though detection of functional failures has been explored extensively in the literature on software validation and verification, in the context of self-healing systems we face new challenges. To be acceptable as monitors in production systems, automatic failure detectors (1) cannot rely on human operators to arbitrate the validity of detected problems, (2) must have only limited performance overhead, (3) must detect failures precisely and produce only few, if any, false alarms, and (4) must detect failures as early as possible. The results of our research facilitate the automatic generation of runtime monitors that meet these criteria. A fifth criterion that might be considered in self-adaptive systems regards the effects of adaptations on the correctness of failure detectors. However, the assumptions we make in Sec. 4 allow us to set aside this consideration for the discussion in this chapter.

We argue that a complete and consistent set of well tailored assertions can meet the requirements above. Encoding thoroughly analyzed system invariants into assertions produces automated oracles, and removes human operators from the loop. Careful choice of logic constructs in assertions can assure low overhead. Assertions suitably placed in critical locations can detect failures precisely and early enough to support efficient fixing.

In current practice, assertions are either added directly to the code by programmers [4,5], or are generated from formal specifications that describe invariants of data-structures and algorithms [6,7]. In both cases getting the specification right is non-trivial and highly error prone [8,9]. Additionally, when writing such

specifications, developers focus on implementation details, hence they might miss constraints stemming from the larger context in which the code will be used. Concentrating on code-level specifications also makes it difficult to express constraints that are not directly related to how the system is implemented, but are imposed by domain specific limitations the system has to adhere to.

We address the problem of producing well tailored assertions by defining a technique to map end-user requirements onto code assertions. We provide developers with a catalog of *property templates* that help developers create explicit specifications of constraints that are implicit in the requirement specifications. However, we do not require complete formal specifications, which are usually hard to write and maintain, but we rely on simple annotations in system models to generate code assertions.

In this chapter we present the methodology (Secs. 2 and 3), define the structure and use of the property catalog, which lies at the heart of the methodology, and report results of our research to derive properties for the catalog (Sec. 4). Related work (Sec. 5) and a discussion of future directions of research (Sec. 6) conclude this chapter.

2 Mapping Requirements to Assertions

To support self-healing systems, failure detection techniques must have a clear notion of what constitutes a failure, must provide means to detect failures at runtime, and must provide enough information about the failures to allow subsequent analysis to determine the cause of the failures. Even though the first two items seem similar, they have to be treated separately: The first requires an explicit specification of what the system should do. The second requires techniques to monitor the system execution, and means to decide when an execution violates the system specifications.

Our goal is to create high-quality runtime failure detectors for system-level requirements, and to reduce the effort required from developers when using our technique. Therefore, the purpose of the methodology we developed is to automate the creation of such detectors as much as possible. To do so we have to address two orthogonal concerns: First, we have to address the efficiency and quality concerns associated with runtime monitoring techniques, and second, we have to bridge the semantic gap between system goals, which are the source of the constraints we monitor, and the implementation details of the system.

During our studies we observed that assertions for high-level properties are often distributed across substantial parts of the system. Creating and adding *all* assertions in the right places is therefore tedious and error-prone. Our technique focuses on two aspects: (1) How to derive and specify system-level constraints, and (2) how to automatically translate constraints into assertions that meet the desired performance and precision requirements.

Figure 1 shows how our methodology addresses the two key aspects in a process centered around a catalog of property templates. In the first step, developers derive properties from requirements specifications, and annotate the system design model with constraints that reflect these properties. This step is difficult to

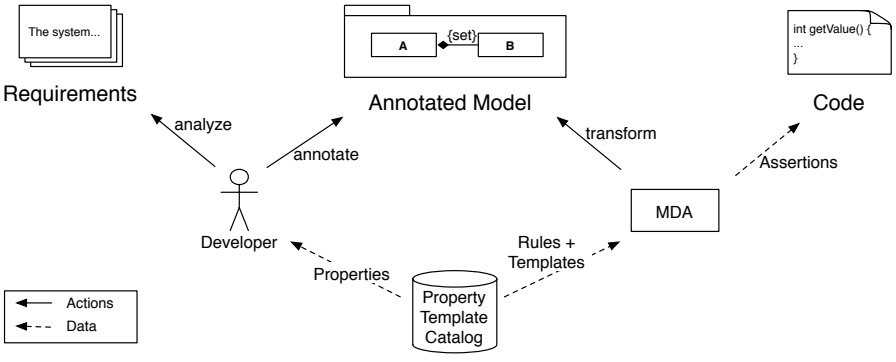


Fig. 1. Methodology activities

automate, because it requires understanding and analyzing the semantics of requirements specifications, which are usually provided in natural language or informal notations that do not lend themselves well to automatic analysis. The annotation language we provide for our technique is not a fully fledged assertion language like JML or Spec#. Instead, we provide a set of concrete annotations with well-defined semantics that readily translate into effective code-level assertions. This has the advantage that model annotations are very simple and can easily be placed. On the other hand, this limits the expressiveness of our technique to exactly the predefined set of annotations.

In the second step, a model-driven assertion generator transforms the annotated model into assertions at the code level. It builds on the concept of property templates, discussed in detail in Sec. 4. Property templates allow developers to express high-level goals with few, simple annotations to system models. The annotations are then automatically translated into the necessary sets of assertions, and inserted in all relevant locations in the code. Like for every model-based technique, for the automatic assertion generation to work, there must be a semantic link between the annotated model and the system implementation. This also means that the model and the code must evolve together when the system changes. Keeping models and code synchronized is a well-researched problem, but still provides significant challenges. In our technique, we alleviate these problems by requiring only partial models of the system. The model has to contain only the entities relevant to each annotation, and assertion generation can proceed. This does not remove the need of keeping model and code synchronized, but it reduces the work of synchronizing the much smaller model to code changes.

3 Asserting Correctness of Phone Bills

In this section, we exemplify the approach through a simple billing application for a telecommunication company.

In the first step of our methodology, developers analyze the system requirements to identify properties using the property template catalog. For instance, the

requirements of our billing system may state that *the phone calls being charged to a customer's account may appear only once in a bill*, because otherwise the calculated total charge would be incorrect. This requirement expresses that some items, here phone calls, have to be unique in the context of a particular collection of items, here the phone bill. This notion of requiring elements of a collection to be *unique* occurs frequently in specifications and represents a constraint on the system. The property template catalog, discussed in detail in Sec. 4, contains a property template `unique`, which matches this requirement.

Having identified properties at the requirements specification level, developers have to examine the design model, to identify the conceptual entities that are constrained by the properties, and have to annotate the identified entities to reflect the property constraints. For instance, the diagram in Fig. 2 shows the conceptual entities relevant to this example. Since the constraint identified above states that phone connections listed in a bill have to be unique, the developers must annotate the association between bills and connections with this constraint, as shown in Fig. 2.

We automatically generate assertions from the annotated model through a set of templates that describe the structure of the assertions, and a set of rules that identify the position of the assertions in the code. The rules for positioning assertions in the code share some common aspects, but many details are domain and platform specific, since the implementation of conceptual entities depends on the programming language, deployment platform, and target environment. For example, the notion of a *component* is implemented differently in a J2EE application and in a highly distributed traffic monitor for a telecommunications network.

In general, the implementation structure may be quite different from the design model. For example, the simple class diagram in Fig. 2 may be naïvely implemented with a phone bill as a container that aggregates the connections initiated by a customer in specific period of time. However in realistic systems, traffic monitoring, customer management and billing are implemented as separate components, whose structure and physical location are dictated by practical

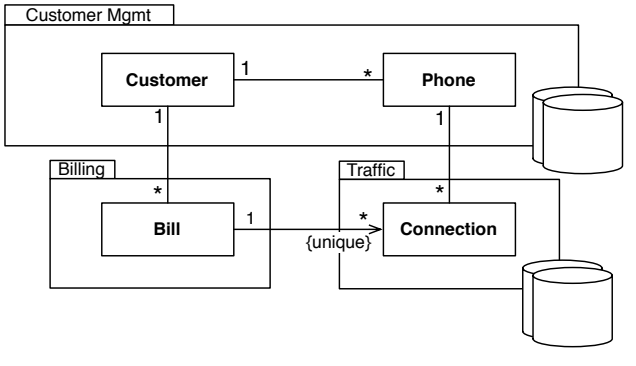


Fig. 2. An excerpt of the phone system data model that shows the conceptual relationships between the entities relevant for the example and the added *unique* constraint

considerations. For example, traffic must be monitored where the traffic is initiated, but capacity constraints might make it prohibitively expensive to transmit and store all the connection data at the location of the billing component. Therefore, the containment relationship between `Bill` and `Connection` will not be reified in the systems implementation. The relation between bills and connections is only reified through the customer management database and pointers to traffic records. Thus, the assertions that monitor the conceptual containment relationship must be transformed to match the system implementation.

If the conceptual model from Fig. 2 was to be implemented directly as classes in an object-oriented language, the uniqueness check could be performed when constructing the bills, making sure that no duplicates are added to each bill. However, in the concrete implementation of the system, the traffic monitor, which records phone calls, and the customer management, which maintains customer and phone contract related data, are implemented by different components that are distributed both physically and logically, and bills are created on the fly from the available data when needed. Therefore uniqueness must be checked already when updating the databases to avoid inconsistencies that would lead to wrong bills.

Since there is no direct containment relation between bills and calls in the implementation, we need to insert assertions between relations that model the containment relation as their composition. In this example, the containment relation is indirectly modeled by the composition of `Bill --> Customer --> Phone --> Connection`. Because of the well-defined structure of the conceptual model, the uniqueness constraint can be pushed down any set of relations whose composition models the same containment relation, and assertions can be inserted at the last relation. In this example, the uniqueness constraint can be placed on the association between `Phone` and `Connection` to detect duplicates as early as the records are created.

Moving constraints along containment relations can strengthen or weaken the constraints themselves, depending on the direction of the associations and their multiplicity. Analyzing this strengthening or weakening is tedious, error prone and easily automatable. Having thresholds in the placement rules allows complete automation, with potential for developer feedback if desired, of the movement of annotations.

The listing in Fig. 3 shows the relevant parts of the code generated for the moved annotation. The advice containing the assertion is triggered after an element is added to a collection. The actual check for uniqueness is implemented in the method `#assertUnique`, and in the nested methods it calls. A lot of the complexity is hidden in the methods `#contains` and `#computeSize`, because these methods have to take into account possible nested checks when the annotation in the model has been moved. Several additional helper methods and classes not shown in the figure are also generated to facilitate the monitoring.

Even though this example contains only four classes and few associations, it indicates that automating the process of moving annotations, and finding the code locations associated with the new model locations provides substantial benefits for developers, since placing a sufficient set of assertions at the right code locations

```
public privileged aspect Phone_Unique {

  pointcut addElement(Connection param): target(Collection+)
    && call(* add(..)) && args(param);

  before (Collection collection, Connection param):
    addElement(param) && target(collection)
    && !this(UniqueCollectionTracker+) {
    pre_contains = contains(collection, param);
    pre_size = computeSize(collection);
  }

  after (Collection collection, Connection param):
    addElement(param) && target(collection)
    && !this(UniqueCollectionTracker+) {
      assertUnique(collection, param);
  }

  private void assertUnique(Collection collection,
                            Connection param) {
    if (pre_contains &&
      computeSize(collection) != pre_size)
      throw new ConstraintViolatedException();
    if (!pre_contains &&
      computeSize(collection) != pre_size+1)
      throw new ConstraintViolatedException();
  }

  private boolean contains(Collection c,
                           Connection object) {
    //return true if the collection c already
    //contains the object
  }

  private boolean computeSize(Collection c) {
    //computes the sum of recursively contained elements
  }

  //more helper methods
}
```

Fig. 3. Excerpt of the code for the annotation in Fig. 2

may be difficult and error prone. In our experiments, the assertions for `unique` have to be inserted in over 100 locations, even in comparatively simple examples. Assume now that the phone provider has roaming contracts with a number of other providers. The billing system will then have to implement gateways to the billing systems of these roaming partners, and at each interface uniqueness checks have

to be performed when data passes through the interface. In this scenario the number of locations where assertions have to be inserted would increase linearly with the number of partners. Automation also relieves developers from the problem of having to think about which assertions to add and remove when maintaining or reengineering a system, because this is handled transparently by appropriate tools.

We automatically translate constraints at the design level to code assertions through property templates. The assertions that characterize templates are derived from properties of abstract entities that generalize the concrete elements, for example mathematical concepts or design patterns. Our example property of unique elements in a collection can be abstractly described by the mathematical concept `Set`. By definition a set does not contain duplicate elements under a given equality relation. Thus, templates for assertions that check for violations of the `unique` property can be based on constraints that characterize an algebraic specification of the abstract data-type `Set`.

To check that the `unique` property holds on a container, we can generate assertions that check that all operations on the container, including the constructor, maintain the property. To reduce execution overhead, we may omit checking operations that do not alter the state of the container, according to trusted specifications.

When executing an operation that may alter the contents of a container, a complete check has to verify that the container has been modified as required. This means "remembering" the content of the container as it was before the operation, and to check if the state has been changed only as specified. For a container behaving like a *set* this means, for example, that an `#add` operation may only add an element if it is not already in the set, and may not change any other element. Maintaining a copy of the state of a container before each operation can consume a lot of memory, and when the container is large, post-operation comparison of the two containers can take a long time. Therefore, weakening the checks may be a necessary trade-off to achieve acceptable overall performance.

The listing in Fig. 4 shows an assertion template that reduces overhead by limiting the check to the size of the container state before and after an `#add` operation. This template trades precision for runtime efficiency in two ways. First, it does not keep track of the state of the container, but uses the containers size as an

```
//pre-operation code
int pre_size = <container>.<size>;
boolean pre_contains =
 <container>.<contains>( param1 );

//post-operation code
if ( pre_contains && <container>.<size> != pre_size )
 throw new ConstraintViolatedException();
if ( !pre_contains && <container>.<size> != pre_size+1 )
 throw new ConstraintViolatedException();
```

Fig. 4. Assertion template for the `unique` property

approximation. Second, it does not scan the whole set to check that all previous elements are unchanged. In particular, it does not check whether an observed size change is actually caused by the element `param1`.

The tokens in angular brackets are place-holders in the template and need to be replaced with appropriate expressions addressing the target elements in the implementation code. In our example, when the checks are pushed down the alternate path through customer and phone, the `<container>` in the instantiated assertion would be an identifier for a particular phone, and the `<size>` and `<contains>` operations would be formulated as queries to the traffic database, where the containment relationship is reified with the phone identifier as a foreign key to delimit the different container instances. In addition to general placement rules that allow moving assertions to improve efficiency and early detection, domain specific placement rules must describe how to identify code that creates new database entries in the traffic database. The translation engine can then combine both rule-sets to find relevant locations and create assertions tailored to each location.

Deciding what is a good trade-off between precision and performance depends on many factors, and should consider the scope and particularities of the system domain. Currently, we consider our technique a complement to traditional testing and validation. Hence, any error we detect can be considered an improvement. On the other hand, adaptations usually suspend the normal system execution, move the system to a safe state, and alter the execution of the system to overcome the problem, thus reducing the system availability during that time. False alarms that unnecessarily trigger expensive adaptations can unacceptably reduce system availability and are thus seldom tolerated. The assertion templates we developed for this chapter not only trade precision for speed, but are also designed to keep false positives to a minimum at the expense of possibly missing some failures. For example, the assertion template shown in Fig. 4 may miss faulty operations, but does not generate false alarms, assuming that the container object's mutating methods do not have unexpected side-effects.

Potential healing actions, or more generally reactions to failures detected by our assertions, are application and domain dependent. In the example of the phone company many possible reactions are conceivable: The offending duplicate entry might be dropped or stored in a separate database for later inspection. If the fault can be determined from the failure, then stronger actions, like inserting additional filtering components or simply replacing the faulty component with one that is more reliable are possible. Classical fault tolerance techniques like roll-back and re-execution are also thinkable. The clear separation of responsibility in the sense-plan-act loop for adaptation allows the configuration of these behaviors in the *planning* phase independent of the detection in the *sensing* phase.

4 Property Templates

Our research focuses on identifying useful property templates. Because property templates link system level properties with classes of failures and faults, the methodology we use to identify property templates starts with the natural sources

of information for both. On the one hand we studied requirements specifications, for example user manuals and API specifications, and collected properties and patterns that occur frequently in several contexts. On the other hand, we checked bug reports in the bug database of the same applications and libraries, and clustered the reported failures by symptoms and causes where possible. We analyzed several open source applications from different domains: Apache Tomcat[2] 6.0.9, a server application, Apache Cocoon[3], a framework for web applications, Apache Lucene[4], a large library for text search.

Requirements properties that occur frequently in more than one application are good candidates to be representative constraints in property templates. When one or more failure clusters match one of the properties collected during our analysis of requirements documents, we establish a link between a property and its failure class. These connections between high-level property and classes of failures, and the thorough study of the faults in the code and the fixes applied to remove the fault guide the definition of assertion templates and translation rules for each property template.

Table 1 lists the main classes of common property constraints that we found. A noticeable aspect of the classes listed in Tab. 1 is that many of them match common design patterns. Basic libraries like the Java SDK have gone to great lengths to provide frameworks for code that maintains these properties. For example, the Java Collection Framework contains interfaces to make classes comparable, and there exist well known design patterns for some aspects of the `immutable` and `unique` properties [10]. Nonetheless, our analysis indicates that these well-understood and often used properties and patterns represent a major source of software faults. This lends additional motivation to our approach to automatically generate checks detecting violations of them.

We produce property templates by augmenting classes of property constraints with assertion templates and rules. We used the classes of constraints listed in Tab. 1 to generate an initial catalog of property templates. Each catalog entry consists of a property template, which is a triple ⟨**C**onstraint, assertion **T**emplate, translation **R**ule⟩. The catalog lists property templates by giving a *Property* identifier, an informal *Description* of the property, an *Elements* entry that defines which elements in a model may be constrained by this property, *Assertion* templates and translation rules, described by relevant *Context* elements and target *Location*.

Table 2 shows an excerpt of the `unique` property template. *Property* and *Description* are the same as in the list in Table 1. *Property* gives the property template a name, and is also used as the label for model annotations. The description is intended to give developers an intuitive understanding of the property; the remaining entries in the table refine and formalize that notion. The entries *Elements* and *Parameters* specify which type of model elements may carry this property, and which additional parameters the annotation may take. The property `unique`, for example, may annotate both, classes (or more generally

[2] `http://tomcat.apache.org`
[3] `http://cocoon.apache.org`
[4] `http://lucene.apache.org/java/`

Table 1. Classes of constraints for property templates

Property	Description
`comparable <C>`	The constrained class must implement a comparison operation matching interface `C`.
`immutable`	The constrained entity may not change its visible state once it is created.
`initialized`	The constrained entity must complete all custom initialization before becoming accessible to clients.
`language <L>`	The constrained entity must be a string and must match a regular expression defining the language `L`.
`unique`	The constrained entity must be unique within its context. If the constrained entity is a relation, tuples in the relation must be unique.

Table 2. Catalog entry for `unique`

Property:	`unique`
Description:	Tuples in the constrained relation must be unique.
Elements:	Classifier, Association
Parameters:	UniquenessProperty
Context:	annotated entity, association ends
Location:	annotated entity

Assertion:	`//pre-operation code`
	`int pre_size = <container>.<size>;`
	`boolean pre_contains =`
	`  <container>.<contains>( param1 );`
	
	`//post-operation code`
	`if ( pre_contains && <container>.<size> != pre_size )`
	`  throw new ConstraintViolatedException();`
	`if ( !pre_contains && <container>.<size> != pre_size+1)`
	`  throw new ConstraintViolatedException();`

classifiers) and associations connecting classifiers, and has an additional parameter `UniquenessProperty`. The parameter value must be the name of an attribute of one of the annotated elements, which determines uniqueness of the annotated entities. These first four entries define the constraint **C**. *Context* and *Location* describe more precisely which parts of the system model are relevant to create correct assertions, and where they need to be placed in the system implementation. Thus, they form the rule **R** of the property template. The *Assertion* templates **T** are templates that abstractly describe the necessary assertions.

For example, the *Context* of the property `unique` is both the annotated association and the associated classes. The information about associated classes is important when the mapping of properties onto assertions is not straightforward,

Table 3. Properties identified during requirements analysis. For each application, the first column represents the number of properties found during requirements analysis, the second column represents the number of reported failures.

Property	**Cocoon**		**Lucene**		**Tomcat**	
Total instances	151	85	–	63	14	109
`caching`	–	2	–	1	–	–
`comparable`	19	–	–	3	1	–
`concurrency`	47	5	–	2	1	4
`immutable`	22	–	–	–	3	2
`initialized`	32	3	–	1	4	6
`language`	1	5	–	1	2	3
`resource mgmt`	8	3	–	–	–	–
`unique`	22	–	–	–	3	–

but involves several entities, as discussed in the example of Sec. 3, where the relation between `Bill`s and `Call`s is implemented by several relations over different sets of entities. The target *Location* for the `unique` assertions is the annotated element. That means that assertions have to be placed around code that implements or uses the annotated association. As discussed in the previous section, the precise meaning of *implements* is application specific and may be defined by customized rules. We omitted listing the translation rules in the catalog entry because of their domain dependent nature. Showing a rule valid for an example domain may not give more intuition about how they work in general than the discussion in the main text (However, the pointcuts in Fig. 3 represent one possible realization of these rules). An example where the location of the assertions is not the annotated element is the property `initialized`. Here the clients of components that promise to be initialized when they are visible have to check for this property. Thus, the assertions have to be placed in client code.

We identified candidates for property templates by analyzing the relation between high-level properties and clusters of reported failures. We considered both the common faults that could be avoided by proper assertions derived from properties, and, where possible, the changes applied to the systems code to fix the identified faults. We took into account performance and precision concerns as discussed in the previous section. Table 3 summarizes some of the results of that comparison. For each application (with the exception of Lucene) we analyzed the requirements, for example API specifications, to identify properties (column 1), and then analyzed bug reports accumulated for each application (column 2). The results show that some properties show up only very rarely (for example `caching`), or only in one application (for example `resource mgmt`). We omitted these from further consideration later on, and the remaining properties we used to develop property templates are listed in Tab. 1. To obtain actual assertion templates we used the clusters of faults identified in our study, considered potential failures these faults could cause, and abstracted away case specific information to obtain sufficiently general descriptions of conditions that signify a failure for each fault class. We

Table 4. Effectiveness of property templates

Application	Faults		Properties	Coverage
	total	relevant		
Tomcat	109	16	4	94 %
Cocoon	85	22	5	86 %
Lucene	63	13	5	84 %

Total faults = number of known faults in the code
Relevant faults = number of faults with a clear connection to a requirement in the user or API specification
Properties = number of different property templates to which more than one relevant fault can be mapped
Coverage = percentage of relevant faults that belong to one of the classes addressed by a property template

encoded these conditions in the assertion templates for each property in the catalog. We discussed a detailed example of deriving assertions templates considering these constraints in Sec. 3.

To check the usefulness of the assertion templates, we studied the relation between the assertions generated from templates and the known faults in the applications we studied (Apache Tomcat 6.0.9, Apache Cocoon and Apache Lucene). Table 4 summarizes the amount of known faults that can be mapped to a property template, and thus indicates the potential effectiveness of our technique. The number of known faults that can be addressed with assertions derived from the property templates defined so far (relevant/total faults) varies between 15% and 25% of the total amount of known faults, and represents a relevant fraction. The studies indicate that well modularized applications and applications that are used as modules in a larger framework are more amenable to our technique than monolithic applications. Additional details are available in [11].

5 Related Work

The idea to use specifications to derive recurring patterns also appears in the work by Dwyer et al. [12]. They study a large set of finite state specifications from the research literature and academic course-work, and collect typical properties appearing in these specifications. Cobleigh et al. build on this work and provide tools to support developers in selecting and applying the derived patterns in their own verification tasks [13]. Our work on the identification of properties that can lead to the generation of property templates is similar in that it also studies software artifacts to derive patterns. However, our study uses two complementary approaches to obtain patterns. We study not only what the software does wrong, but also relate it to the high level specification as an additional internal validation step that is missing from previous work.

There is substantial work on the boundaries between software engineering and programming languages, which addresses problems related to some of the properties we have listed in our catalog. For example, Zibin et al. and Unkel and Lam address questions regarding immutable objects and variables [14,15]. Zibin et al. discuss a language extension to Java that allows the explicit specification of immutability properties of objects and references, which go far beyond the possibilities of Java's `final` qualifier. Unkel and Lam introduce the notion of *stationary fields*, that is fields where all write operations to the field happen before any read operation. Thus, at read time these variables can be treated similar to `final` fields. The work by Unkel and Lam introduces a static analysis algorithm to identify such fields, but no method to specify them beforehand, which means there is no way to dynamically check for violations of such a property. Boyland and Retert address the notion of object *uniqueness* and how languages can enforce the semantics of unique objects [16]. The similarity between these approaches and our methodology is that they address similar properties, partially by explicitly specifying them to enable static or dynamic checks, and partially by analyzing real system to determine which properties occur frequently. However, in all cases their analysis and specification happens at the code level, and is thus far from the semantics of the end-user specifications, which we use to identify the properties to monitor.

Chan et al. and Beckman et al. work on static analysis methods dealing with concurrency problems [17,18]. In both cases the basic technique uses annotations in the code to specify additional constraints on classes and methods. Chan et al. provide annotations to explain the behavior of objects or methods, for example *reading* or *writing* certain variables. Beckman et al. specify constraints that declare the allowed accesses to objects, and use the typestate system to declare usage protocols of methods. In both cases, a static analysis uses these annotations to detect potential or real race conditions and deadlocks.

Work addressing self-healing, that is autonomic repairing of functional problems, usually addresses either transient failures like race conditions, or resource management problems like memory exhaustion. For example, RX by Qin et al. propose an approach based on checkpoints and dynamic changes of the *environment* of the system to fix failures due to memory leaks and buffer overflows [19]. Goldstein et al. try to delay system crashes due to memory exhaustion by monitoring object activity on the heap and transparently removing unused objects from main memory [20]. This effectively delays and reduces the effects of memory leaks in languages with managed memory. From a direction that appears to be more inspired by traditional fault tolerant systems, Candea et al. introduce *micro-reboot*, a technique that restarts individual components of a system, either in reaction to a detected failure, or regularly in the hope that this rejuvenation occurs before faults like memory leaks lead to irreparable damage or system crashes [21]. From this they develop the more general idea of software rejuvenation. Interestingly, none of these techniques *heals* the problem, rather they reduce the likelihood for failures to occur. This is why these techniques can be reasonably successful without precise failure detection.

Using specifications to automatically derive oracles is one of the objectives of specification- or model-based testing. However, in this field, approaches that attempt to derive oracles usually require a *complete* and *formal* specification [22]. Furthermore, most practical approaches require developers to specify constraints on the level of individual methods or classes, which makes it hard to keep the bigger picture of system level requirements in mind [23,24]. Our property template based methodology explicitly addresses this issue by providing a link from end user requirements all the way down to assertions in the code. Our approach only requires a partial, annotated structural model of the parts of the system that should be augmented with failure detectors.

There are several fully or partially automated failure detection approaches that do not require formal specifications, but use capture and replay techniques or dynamic invariant inference to build models of the system. Hangal and Lam use dynamic invariant inference to build a model of system executions [25]. Since their goal is complete automation of the model building and monitoring process, their system only issues warnings to be analyzed off-line by developers. Approaches that explicitly try to solve the oracle problem usually rely on a separate training phase to learn the model [26,27]. After the learning phase this model remains fixed and serves as the oracle to distinguish between valid and invalid executions. The approaches by Baah et al. and Lorenzoli et al. combine trace information with static invariants to improve the quality of the models prediction. However, since dynamic behavior inference relies only on the implementation of a system, it is not able to incorporate notions expressed in end-user requirements.

6 Conclusions and Outlook

In this chapter we presented a method for automatically generating assertions to detect violations of end-user requirements. We target self-healing software, which requires both detecting failures while the system is running in a production environment, and providing enough information about the detected failures to enable automatic healing actions. The self-healing requirements add the challenges of full automation, low runtime overhead, and high detection precision to the problem of failure detection as it would occur in traditional validation and verification.

Our approach is centered around the concept of property templates, which link requirements and design, drive assertion generation and incorporate rules that specify where to place assertions in the code. Property templates enable easy annotation of models, and correct and complete generation of assertions for high-level properties. We address the technical concerns of performance and precision discussed above by carefully connecting the three core elements that we need to address in failure detection: faults, failures, and reusable information on how to detect the failures.

In this chapter we present the property template catalog: We discuss how the catalog drives the whole generation process, and how we can derive useful catalog entries. In future work we plan to refine and extend the initial catalog of property templates to match a wider range of requirements. At the time of writing, all properties refer to structural and functional features of systems, such as uniqueness of

object and limits to parameter values. We believe it is relatively straightforward to extend the technique to the monitoring of simple temporal properties, like they might be specified in Service Level agreements. Such work will be similar to more complex MOF based approaches (for example [28]), but might be easier to use thanks to the simple specification language we provide.

Our experiments indicate that for some relevant classes of constraints, developers are relieved from the burden of deriving, tuning, and inserting matching checks for these constraints. As such, the technique improves developer productivity and software quality. Since assertion generation and insertion are fully automated, this process also allows efficient maintenance of constraints even when the systems structure changes, because matching assertions can simply be re-generated and will be inserted, even if new relevant locations have been created by the change.

The applicability of property templates listed in the catalog is limited by the transformation rules available. These rules are domain specific, which implies that most deployment platforms would require modifications to those rules. In future work we plan to define techniques to extract common patterns for domain independent translation rules, to separate domain independent aspects from domain dependent ones and increase catalog portability.

In the introduction we discussed five criteria that runtime failure detectors for self-adaptive systems should meet. The technique we discuss in this chapter addresses four of these criteria, but does not consider how adaptations to the system may also require adaptations to the failure detectors. Since the properties we define in the property template catalog refer to system goals, the properties in the models only have to change when the goals of the system change. Further, the generated failure detectors are dynamically deployed to match component interfaces. Hence, as long as components added due to adaptations match annotated interfaces, they can be augmented with already existing failure detectors. Part of our ongoing work is to elicit and clarify conditions when our generated failure detectors will have to adapt, and to define adaptation strategies for these cases.

References

1. Koren, I., Krishna, C.M.: Fault-Tolerant Systems. Morgan Kaufmann, San Francisco (2007)
2. Laprie, J.C., Avizienis, A., Kopetz, H. (eds.): Dependability: Basic Concepts and Terminology. Springer, New York (1992)
3. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer 36(1), 41–50 (2003)
4. Rosenblum, D.S.: A practical approach to programming with assertions. IEEE Transactions on Software Engineering 21(1), 19–31 (1995)
5. Das, M.: Formal specifications on industrial-strength code—from myth to reality. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, p. 1. Springer, Heidelberg (2006) (invited talk)
6. Machado, P.D.L.: Testing from Structured Algebraic Specifications: The Oracle Problem. PhD thesis, University of Edinburgh (2000)

7. Richardson, D.J., Aha, S.L., O'Malley, T.O.: Specification-based test oracles for reactive systems. In: Proceedings of the 14th International Conference on Software Engineering, ICSE 1992, pp. 105–118 (1992)
8. Ciupa, I., Meyer, B., Oriol, M., Pretschner, A.: Finding faults: Manual testing vs. random testing+ vs. user reports. Technical Report 595, Department of Computer Science, ETH Zurich, Switzerland (2008)
9. Voas, J.M., Miller, K.W.: Putting assertions in their place. In: Proceedings of the 5th International Symposium on Software Reliability Engineering, pp. 152–157 (1994)
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, Reading (1994)
11. Wuttke, J.: Property classes and assertions supporting runtime failure detection. Technical report, University of Lugano, Switzerland (2008)
12. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 1999 International Conference on Software Engineering, ICSE 1999, pp. 411–420 (1999)
13. Cobleigh, R.L., Avrunin, G.S., Clarke, L.A.: User guidance for creating precise and accessible property specifications. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT 2006/FSE-14 2006, pp. 208–218 (2006)
14. Zibin, Y., Potanin, A., Ali, M., Artzi, S., Kieżun, A., Ernst, M.D.: Object and reference immutability using java generics. In: ESEC-FSE 2007: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 75–84. ACM, New York (2007)
15. Unkel, C., Lam, M.S.: Automatic inference of stationary fields: a generalization of java's final fields. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, pp. 183–195 (2008)
16. Boyland, J.T., Retert, W.: Connecting effects and uniqueness with adoption. SIGPLAN Notices 40(1), 283–295 (2005)
17. Beckman, N.E., Bierhoff, K., Aldrich, J.: Verifying correct usage of atomic blocks and typestate. In: Proceedings of the 23rd ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2008, pp. 227–244 (2008)
18. Chan, E.C., Boyland, J.T., Scherlis, W.L.: Promises: Limited specifications for analysis and manipulation. In: Proceedings of the 20th International Conference on Software Engineering, ICSE 1998 (1998)
19. Qin, F., Tucek, J., Sundaresan, J., Zhou, Y.: Rx: Treating bugs as allergies — a safe method to survive software failures. In: Proceedings of the 20th ACM Symposium on Operating Systems Principles, SOSP 2005, pp. 235–248 (2005)
20. Goldstein, M., Shehory, O., Weinsberg, Y.: Can self-healing software cope with loitering? In: Proceedings of the 4th International Workshop on Software Quality Assurance, SoQUA 2007, pp. 1–8 (2007)
21. Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., Fox, A.: Microreboot - a technique for cheap recovery. In: Proceedings of the 6th Symposium on Operating Systems Design and Implementation (2004)
22. Antoy, S., Hamlet, R.: Automatically checking an implementation against its formal specification. IEEE Transactions on Software Engineering 26(1), 55–69 (2000)
23. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)

24. Cheon, Y., Leavens, G.T.: The JML and JUnit way of unit testing and its implementation. Technical Report TR #04-02, Department of Computer Science – Iowa State University (2004)
25. Hangal, S., Lam, M.S.: Tracking down software bugs using automatic anomaly detection. In: Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, pp. 291–301. ACM, New York (2002)
26. Baah, G.K., Gray, A., Harrold, M.J.: On-line anomaly detection of deployed software: a statistical machine learning approach. In: Proceedings of the 3rd International Workshop on Software Quality Assurance, SOQUA 2006, pp. 70–77. ACM, New York (2006)
27. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: Proceedings of the 30th International Conference on Software Engineering, ICSE 2008, pp. 501–510. ACM, New York (2008)
28. Skene, J., Lamanna, D.D., Emmerich, W.: Precise service level agreements. In: Proceedings of the 26th International Conference on Software Engineering, pp. 179–188 (2004)

Using Filtered Cartesian Flattening and Microrebooting to Build Enterprise Applications with Self-adaptive Healing

J. White[1], B. Dougherty[1], H.D. Strowd[2], and D.C. Schmidt[1]

[1] Department of Electrical Engineering and Computer Science,
Vanderbilt University, Nashville, TN, USA
{jules,briand,schmidt}@dre.vanderbilt.edu
[2] Institute for Software Research
Carnegie Mellon University
Pittsburgh, PA, USA
hstrowd@andrew.cmu.edu

Abstract. Building enterprise applications that can self-adapt to eliminate component failures is hard. Existing approaches for building adaptive applications exhibit significant limitations, such as requiring developers to manually handle healing side-effects, such as lock release, thread synchronization, and transaction cancellation. Moreover, these techniques require developers to write the complex recovery logic needed to self-adapt without exceeding resource constraints.

This paper provides two contributions to R&D on self-adaptive applications. First, it describes a microrebooting technique called Refresh that uses (1) feature models and a heuristic algorithm to derive a new and correct application configuration that meets resource constraints and (2) an application's component container to shutdown the failed subsystems and reboot the subsystem with the new component configuration. Second, we present results from experiments that evaluate how fast Refresh can adapt an enterprise application to eliminate failed components. These results show that Refresh can reconfigure and reboot failed application subsystems in approximately 150ms. This level of performance enables Refresh to significantly improve enterprise application recovery time compared to standard system or application container rebooting.

1 Introduction

Current trends and challenges. Enterprise applications are large-scale software systems that execute complex business processes, such as order placement and inventory management. Since many enterprise applications receive considerable client traffic, they are often hosted on multiple *application servers* distributed across a local network. Most enterprise applications utilize component middleware, such as Enterprise Java Beans (EJB), to reduce the effort of developing the distributed communication infrastructure by managing the complex distributed interactions between application components and ensuring data integrity through distributed transaction controls.

B.H.C. Cheng et al. (Eds.): Self-Adaptive Systems, LNCS 5525, pp. 241–260, 2009.

The failure of an enterprise application can have considerable negative impact (*e.g.*, lost orders, customer irritation, etc.) on an organization. As a consequence, high availability is important for most enterprise applications. Regardless of how much testing and system validation is done, systems can and often do fail [10]. In these situations, speedy recovery of system functionality is critical.

Many organizations use manual processes to recover from failures of enterprise applications [10]. For example, when an EJB application fails, system administrators may restart a group of application servers to attempt to remedy the error. If the error is not fixed by the restart, the administrators may begin collecting logs from the application servers and scanning them for errors. These manual processes are time consuming and error-prone and can leave an application offline for an extended period while the root cause of the failure is identified and remedied.

To address the limitations of human-based recovery of application failure, self-adaptive capabilities are needed that can identify failed components and perform self-adaptive healing to quickly recover. Rather than being off-line for minutes or hours, self-adaptive systems should be able to heal in milliseconds or seconds. Despite the potential payoff associated with self-adaptive healing capabilities, enterprise applications are rarely developed using these techniques since (1) developing the complex logic to determine how to fix a failure cleanly is hard and (2) implementing healing actions requires handling a plethora of challenging side-effects, such as the need to roll-back distributed transactions.

Rather than focusing on fine-grained self-adaptive healing systems, most organizations today leverage clustering and other redundancy mechanisms to ensure availability. Although these macro-level approaches can improve availability, they require additional hardware and complex system administration. Moreover, there are many types of failures that macro-level approaches cannot fix. For example, if a database or remote service that an enterprise application relies on becomes inaccessible due to a network failure, an entire cluster of redundant application instances will be brought down. In this situation, however, if the application could self-heal by loading additional components to communicate with an alternative but not identically accessed database, it could continue to function.

Since software development projects already have low success rates and high costs, building an application capable of healing is hard [20,3]. Moreover, building adaptive mechanisms greatly increases application complexity and can be hard to decouple from application code if the development of the adaptive mechanism is not successful. In addition, most self-adaptive healing approaches are not suitable for enterprise applications because they do not take into account transaction state, clean release of resources, and other critical actions that must be coordinated with an enterprise application server.

Solution approach → Microrebooting and Feature-based Reconfiguration. Our approach to reducing the complexity of developing self-adaptive healing enterprise applications is called *Refresh*. Refresh uses a combination of *feature models* [15] (which describe an application in terms of points of variability and their affect on each other) and *microrebooting* [8] (which is a technique for rebooting a small set of failed components rather than an entire application server) to

significantly reduce the complexity of implementing an application with self-adaptive healing capabilities. When an application component fails, Refresh (1) uses the application's feature model to derive a new application configuration, (2) uses the application server's component container to shutdown the failed component, and (3) reboots the component in the newly derived configuration. Refresh relies on the ability to transform a feature model into a constraint satisfaction problem (CSP) and use a constraint solver to autonomously derive a new configuration.

Our previous work [25,23] showed how Refresh's CSP-based healing could be used to reduce the complexity of implementing self-adaptive healing applications. When the self-adaptive healing mechanism needs to respect resource constraints, such as bandwidth or memory limits, a CSP-based approach for deriving application configurations from feature models becomes too slow for enterprise applications. Selecting a feature configuration that adheres to resource constraints is an NP-Hard problem that is time-consuming to solve with a CSP-solver.

This paper extends our previous work by showing how *Filtered Cartesian Flattening* and multidimensional multiple-choice knapsack heuristic algorithms can be used as the feature selection mechanism to drastically reduce feature selection and consequently, self-adaptive healing time. We show how these algorithms can be combined with microrebooting, component middleware container hotswap capabilities, and feature models to create self-adaptive enterprise applications. We also present empirical results that show the increase in scalability and speed provided by Filtered Cartesian Flattening (FCF) versus a CSP-based reconfiguration approach. We provide empirical results comparing our original Refresh + CSP technique to the new Refresh + FCF technique. Furthermore, we provide an extensive comparison of the pros and cons of our Refresh + CSP self-adaptive approach versus our new Refresh + FCF approach.

Paper organization. The remainder of this paper is organized as follows: Section 2 presents the e-commerce application we use as a case study throughout the paper; Section 3 enumerates current challenges in applying existing MDE techniques for building self-adaptive healing applications that must adhere to resource constraints; Section 4 describes Refresh's approach to using feature models, microrebooting, and Filtered Cartesian Flattening to reduce the complexity of modeling and implementing an application that can heal; Section 5 analyzes empirical results obtained from applying Refresh to our case study; Section 6 compares Refresh with related work; and Section 7 presents concluding remarks.

2 Case Study: ICred

Enterprise applications have a number of complex considerations that make it hard to build an application capable of self-adaptive healing. To showcase these challenging aspects of enterprise applications, we present a case study based on an enterprise application that provides instant credit decisions for in-store purchases. Throughout the paper, we refer to our case study application as *ICred*. The high-level architecture of ICred is shown in Figure 1.

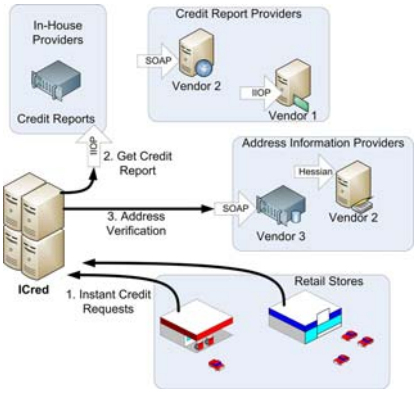


Fig. 1. The ICred Instant Credit Enterprise Application

When a customer in a retail store wishes to purchase an expensive item, such as a computer projector, the store clerk can offer the customer an instant line of credit to make the purchase and pay later. If the customer is interested in obtaining the line of credit, the store clerk keys in the customer's information and a request for credit is sent to the remote ICred server for approval. ICred must pull the customer's credit report and other needed information to make the credit decision.

ICred is used for a number of different retailers and each retailer has a specific set of requirements for validating a credit application and issuing an approval. Stores that sell less expensive and less durable items, such as computer equipment, may require a simple validation of the customer's residence information and bank accounts. Vendors of more expensive items, such as car dealerships, require more extensive sets of information, such as a full credit report and verification of a previous address. Each customer is supported by a custom configuration of ICred that is not shared.

Instances of ICred are run and managed by an information supplier on behalf of retail chains. Each piece of information needed for the credit decision can either be obtained in-house or from another information supplier. Whenever ICred requests a piece of information on a customer from another supplier, a small fee is paid to the information vendor that services the request. Information can be purchased from multiple vendors at varying prices based on volume.

An ICred configuration receives instant credit requests from thousands of retail locations and must be continuously available. A failure to make a credit decision could result in a customer not making a large purchase. When one of ICred's information suppliers becomes unavailable, ICred can fail over to another supplier. For example, Figure 2, shows the different sources of information that can be used to obtain credit reports.

Figure 2 shows a feature model for an e-commerce application called `CreditReportProvider` that represents a service for obtaining credit reports. The `CreditReportProvider` feature has different sub-features, such as different potential vendors that can serve as the credit report provider service. If the `Vendor 1` feature is chosen, it excludes the other potential providers' services from being used

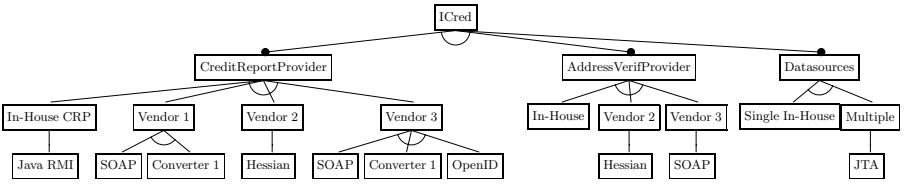


Fig. 2. Feature Model of the Available Credit Report Providers

(it constrains the other features). If `Vendor 1` service fails, a new feature selection can be derived that does not include the failed service's feature. When a component failure occurs, Refresh uses an application's feature model and a constraint solver to derive an alternate but legal configuration of the application's component that eliminates the failed component implementation.

Failing over to another supplier involves a number of complex activities. Information vendors represent the same information using slightly different formats and leverage different request protocols. Depending on the vendor chosen, it may be necessary to load various special converter and protocol handlers into the application. Moreover, since ICred receives a high request volume, it must try to ensure that the combination of protocols used by its current configuration of information vendors will not saturate the network. Finally, since per request prices vary across information vendors, ICred must also try to minimize the cost incurred by the configuration of external information vendors.

To showcase the complexity of performing self-adaptive healing in an enterprise application, we explore the difficulty of failing over between local and external information services in ICred. Section 3 presents the complexities of developing healing logic and adaptation actions. Section 4 shows how Filtered Cartesian Flattening can be used to derive a new application configuration to eliminate a failure and boot the configuration using the application's component container.

3 Self-Adaptive Healing Challenges for Enterprise Applications

This section describes the challenges associated with implementing a self-adaptive healing enterprise application. First, we show that the need to adhere to resource constraints, such as total available network bandwidth, makes finding a way of healing an enterprise application an NP-Hard problem. Second, we discuss how even if a way of healing the application can be found, numerous accidental complexities, such as the need to properly handle in-process transactions, make it hard to implement healing actions.

3.1 Challenge 1: Resource Constraints Make Adaptation Actions Extremely Complex

When an application component fails and requires healing, adaptation actions must be run to reach a new and valid state. We term the sequence of

adaptation actions that are run to fix a failed application subsystem as a *recovery path*. A chief complexity of implementing an application capable of self-adaptive healing is building the logic to select a recovery path for a given application failure.

Recovery actions are used to perform two key types of activities: (1) performing resource cleanup and release from failed application components and (2) determining what new application components can be loaded to heal a failure. The difficulty in building recovery logic is that the second critical activity, selecting the new components to load, requires finding a series of application components that fit into the resource limits of the application. Selecting a series of components that adheres to a resource limit is an instance of the NP-Hard knapsack problem.

For example, consider the failure of the `In-House CRP`. ICred's `In-House CRP` can be swapped out to one of three remote services. When the local `In-House CRP` fails, the recovery logic must determine the optimal subset of these remote services to fail-over to in order to fix the error. Furthermore, the recovery logic must attempt to minimize the cost of the information provider services that are used in the new configuration.

Network bandwidth consumption must be accounted for in the healing process. Each remote service uses a different protocol for communication and consumes varying amounts of network bandwidth. The Java RMI service uses the efficient binary IIOP protocol. The SOAP service, however, sends comparatively large XML messages over HTTP and consumes significantly more bandwidth. Depending on what combination of services are currently being used by the application, the network may or may not have sufficient bandwidth to fail over to the SOAP-based service. Even if the `Vendor 1` SOAP-based service is the cheapest to fail-over to, it may not be possible due to network bandwidth limitations.

If the SOAP-based service is the only of the three alternate remote services that is reachable after the failure, the healing logic may need to shutdown and swap other parts of the application (*e.g.*, AddressVerifProvider, etc.) to less bandwidth consumptive remote services so that the SOAP service can be used. For example, if the `CreditReportProvider` is using a SOAP-based remote service, it may need to be swapped to `Vendor 2`'s Hessian-based service to allow the SOAP-based product service to be used. Finding the right set of services to swap in and out of the application is NP-Hard and difficult to do quickly at runtime. Performing simultaneous cost optimization is even harder.

Designing this type of complex adaptive logic to choose a recovery path is hard. For most enterprise application development projects, this type of complex adaptation logic is not feasible to develop from scratch. Moreover, with nearly 53% of software development projects being completed over-budget and 18% of projects canceled [26,17] adding this type of complex adaptive logic adds significant risk to a project. In Section 4.2, we show how we use feature models and the Filtered Cartesian Flattening algorithm to eliminate the need to write complex recovery path selection logic.

3.2 Challenge 2: Accidental Complexity Makes Adaptation Actions Hard to Develop

Enterprise applications are typically built on top of component middleware, such as Enterprise Java Beans. Component middleware provides an *application container*, which manages the intricate details of thread synchronization, distributed/local transaction control, and object pooling. One key challenge of developing self-adaptive healing mechanisms for enterprise applications is properly and cleanly handling the nuanced considerations related to these aspects of the application. For example, if a credit report provider fails, the application must ensure that any distributed transactions associated with the provider are rolled back and cleanly terminated before a new provider is swapped in. Figuring out the right way to terminate transactions, release locks, terminate network connections, and release other resources when healing occurs is hard.

When healing takes place, a further challenge of properly handling transactions and other container managed services is that the application does not have direct control over them. For example, EJBs are not allowed to perform thread synchronization or manually obtain locks. If a failure occurs in a multi-threaded application, therefore, it is hard for an EJB to ensure that data corruption does not occur if it reconfigures the application's internal structure.

An issue further complicating the healing process is that healing may require changing the policies the container uses to manage these services. In ICred, for example, if ICred is using all local data sources, it can use standard local transaction management through the container. If ICred fails over to a remote datasource, however, it must also force the container to reconfigure itself to use the Java Transaction API (JTA) to manage distributed transactions across both the local and remote datasources. It is hard to perform these numerous complex reconfiguration processes manually. Section 4 describes how we use the application component container's standard lifecycle mechanisms to perform healing and eliminate the need to write custom recovery actions.

4 Solution Approach→Combining Refresh and Filtered Cartesian Flattening

The challenges in Sections 3.1-3.2 stem from two primary causes: (1) the need for developers to implement complex recovery path selection logic that accounts for resource constraints and (2) the need for developers to implement complex recovery actions that correctly coordinate and handle the side-effects of healing, such as graceful transaction failure. This section presents an overview of *Refresh* [25] and shows how we extend it with the Filtered Cartesian Flattening algorithm to address these challenges.

4.1 Overview of Refresh

Refresh uses feature models to capture the rules for what is a correct system state, which when combined with the Filtered Cartesian Flattening feature selection algorithm, can be used to automate the selection of a new configuration to reboot

into. After a new and valid configuration is found, Refresh uses the application's container to swap out the failed components and boot the new alternate configuration. Automating the reconfiguration process eliminates the need for developers to design and implement the recovery path selection logic, which addresses Challenge 2 from Section 3.1.

Using the container's normal lifecycle facilities to perform healing (*e.g.*, rebooting and hotswapping), eliminates the need for developers to manage the side-effects of healing since they are automatically managed by the container when lifecycle management activities are performed. As shown in Section 5, using Filtered Cartesian Flattening and container rebooting to perform resource constrained healing provides fast recovery at a significantly reduced development cost compared to recovery action oriented techniques.

Refresh is designed for enterprise applications where 1) failing components can safely be rebooted, 2) the application container's ability to handle transaction and other failures provides a sufficient guarantee of safety for the developers, and 3) developers do not want to implement custom fine-grained healing. The technique is not suited for safety-critical applications outside the enterprise computing domain, such as flight avionics. If the three conditions outlined above do not hold, Refresh is not applicable.

Refresh is based on the concept of microrebooting [8]. When an error is observed in the application, Refresh uses the application's component container to shutdown and reboot the application's components. Using the application container to shutdown the failed subsystem takes milliseconds as opposed to the seconds required for a full application server reboot. Since it is likely that rebooting in the same configuration (*e.g.* referencing the same failed remote service) will not fix the error, Refresh derives a new application configuration from the application's feature models that does not contain the failed features (*e.g.*, remote services).

The application configuration dictates the remote services used by the application. The application configuration determines any local component implementations, such a SOAP messaging classes, needed to communicate and interact properly with the remote services. After deriving the new application configuration and service composition, Refresh uses the application container to reboot the application into the desired configuration. The overall Refresh healing process is shown in Figure 3.Throughout the healing process, Refresh does not use any

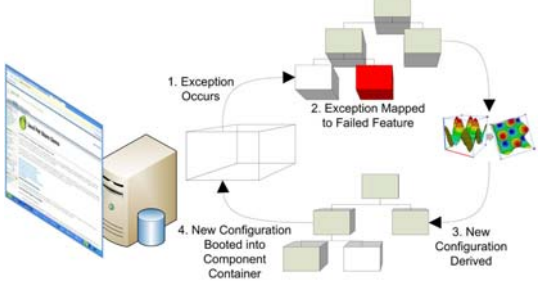


Fig. 3. Refresh Healing Process

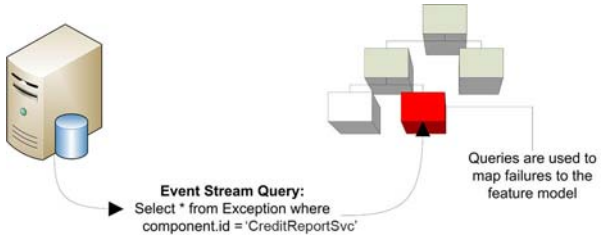


Fig. 4. Mapping Failures to Features

custom recovery actions. All error states are transitioned out of through a single recovery path, shutting down the application components via the container, automatically deriving a new and valid configuration/service composition, and restarting the application components. No application-specific recovery action modeling or recovery application implementations are required. Refresh interacts directly with the application container, as shown in Figure 3. During the initial and subsequent container booting processes, Refresh transparently inserts *application probes* into the application to observe the application components. Observations from the application components are sent back to an *event stream processor* that runs queries against the application event data, such as exception events, to identify errors. An example event stream query and mapping to the feature model is shown in Figure 4. Whenever an application's configuration requires healing, *environment probes* are used to determine available remote services and global application constraints, such as whether or not JTA is present.

4.2 Feature Model Configuration Healing

At the core of the Refresh approach is its ability to derive a new configuration for the application that both eliminates any failed components and adheres to resource limitations. Refresh uses a feature model of the application to capture the rules for reconfiguration. When a failure occurs, the configuration space defined by the feature model is searched for a new and valid configuration.

A feature model is used to define the configuration space of an enterprise application by defining configuration rules, such as:

- What alternate implementations of components are available
- What dependencies (such as libraries, configuration files, etc.) must be used with each component
- What combinations of components form a valid and complete application composition
- Annotations describing how much RAM, Bandwidth, etc. is consumed by each feature

Searching a feature model's solution space for a valid configuration is an instance of the NP-complete circuit satisfiability problem. The feature model can define an

arbitrary boolean formula. Each boolean term represents the presence of a specific feature. The constraints in the feature model are the AND, OR, and NOT constraints used to form the circuit satisfiability clauses. Numerous research approaches have applied techniques such as SAT solvers [4,18], Binary Decision Diagrams (BDDs) [9], and Constraint Satisfaction Problem (CSP) solvers [21,5], to find valid feature model configurations.

Our initial implementation of Refresh used the CSP-based approach proposed by Benavides [5] and extended by us to include resource constraints [24,22]. CSP-based feature selection techniques work well when resource constraints are not included. Through experiments that we performed [23], however, we observed significant scalability problems for CSP-based feature derivation with resource constraints, as shown in the results in Section 5.3. Other exponential exact derivation techniques, such as SAT solvers and BDDs, suffer from these same scalability problems [23].

A number of heuristic techniques can be applied to improve the performance of these exact solving techniques. For example, by choosing the correct variable ordering, many BDD-based problems can be simplified significantly. Choosing the best variable ordering, however, is an NP-Hard problem and must be performed on a per-problem basis. Similar techniques can be applied to CSP-based configuration derivation, but must also be performed on a per problem basis.

Since the goal of Refresh is to simplify the implementation process of applications capable of self-adaptive healing, it would not be reasonable to expect these heuristic techniques to be learned and applied by normal developers. Moreover, the application of these techniques requires significant skill. Just as good application design is an art form, knowing which of these heuristics to apply and how to apply them is also an art. We do not think is reasonable to expect developers are willing and/or able to become experts in these techniques. We have therefore not considered these techniques for Refresh.

4.3 Filtered Cartesian Flattening

To overcome the scalability issues associated with finding a new and valid feature configuration, we incorporated the Filtered Cartesian Flattening feature selection algorithm into Refresh. Filtered Cartesian Flattening is a polynomial-time algorithmic technique that approximates a feature configuration problem with resource constraints as a multidimensional multiple-choice knapsack problem (MMKP) [23]. A standard knapsack problem attempts to find a subset of a series of items that fits into a knapsack of limited size and maximizes the value of the items inside the knapsack. An MMKP problem is a variant of a knapsack problem where the items are subdivided into disjoint sets and exactly one item must be chosen from each set to put into the knapsack. Both variants of the problem are NP-Hard [19].

The reason that Filtered Cartesian Flattening approximates the feature configuration problem as a MMKP problem is that there are a number of excellent polynomial-time heuristic algorithms that have been developed for MMKPs. For example, the M-HEU and C-HEU heuristic MMKP algorithms can solve large MMKPs in milliseconds with an average of over 95% optimality [19]. Once a fea-

ture configuration problem is represented as a MMKP, these heuristic algorithms can be used to derive a feature selection. When a failure occurs, the speed of Filtered Cartesian Flattening, which uses MMKP heuristic algorithms, is far more important than its minor tradeoff in healing solution optimality.

Filtered Cartesian Flattening approximates a feature model as an MMKP problem by finding a series of independent subtrees in the feature model that can be configured independently. Each of these subtrees is represented as an MMKP set. The items within the MMKP sets represent the valid configurations of their respective subtrees. Because each MMKP set represents a subtree of the feature model, by choosing a configuration from each MMKP set and composing them, a complete feature model configuration will always be reached.

Since there may be an exponential number of possible configurations of each subtree, Filtered Cartesian Flattening employs an approximation technique. As Filtered Cartesian Flattening enumerates the possible configurations of each feature model subtree, it bounds the MMKP set size and selectively filters which configurations are propagated into the sets. Typically, a heuristic that selects configurations with the best ratio of value/resource consumption is used as the selection criteria.

To derive a configuration that omits the failed feature while still adhering to resource constraints, Refresh utilizes Filtered Cartesian Flattening. During the enumeration process, Filtered Cartesian Flattening disallows the inclusion of the failed feature to any of the MMKP sets. Due to this exclusion, the feature can not belong to any configuration that can be derived from the resulting MMKP problem, thus disallowing the failed feature to be present in the new feature set. After deriving the new feature configuration, the application container is used to shutdown the old configuration and boot the new configuration.

5 Refresh and Filtered Cartesian Flattening Performance

This section presents results from experiments we performed to empirically evaluate the performance of Refresh's feature reconfiguration and container-based healing. We used a reference implementation of an enterprise request processing application, implemented on top of the Java Spring Framework [13], that could fail over between a number of different remote and local data sources. The implementation was comprised of roughly 15,000 lines of code using a combination of Java, Java Server Pages, XML, and SQL.

Our prior work [25] conducted experiments to measure the reduction in implementation complexity provided by Refresh. This paper extends our prior work by evaluating the performance of feature model and container-based healing. Moreover, we analyze how automated feature selection techniques can be made more scalable to handle resource constraints and optimization goals.

5.1 Hardware and Software Testbed Configuration

The experiments with the application were performed on a Pentium Core DUO 2.4ghz processor, with 3 gigabytes of RAM, running Windows XP. A Java Virtual

Machine, version 1.6, was run in client mode for all tests. We used Apache Tomcat 6 as the web container for the application.

To test the performance of Refresh, we implemented a self-adaptive healing version of the application and compared its performance to the conventional (non-adaptive) implementation. The first set of experiments compared the performance of the Refresh-based application to the conventional unmodified application to measure the overhead of using a container-based healing approach. The second set of experiments extended the Refresh application to adhere to a bandwidth constraint. We measured the configuration derivation times of both the Filtered Cartesian Flattening configuration derivation technique and the CSP-based technique to compare scalability.

5.2 Refresh Performance

To create an initial performance baseline to compare against, we used Apache JMeter to simulate the concurrent access of 30 different customers to the application and the time required to complete 1,000 requests. Figure 5 shows the average time required to complete various parts of the request process throughout the experiment.

We also used Apache JMeter to simulate the concurrent access of 30 different customers to the Refresh-enabled application and the time required to complete 1,000 requests. To measure Refresh's worst case performance overhead, we used the CSP-based configuration derivation technique for this experiment since it was slower than the Filtered Cartesian Flattening technique. The performance results were identical to the conventional application implementation. This result was expected since the time-consuming healing process is only invoked during component failures. Moreover, our Refresh application implementation used very lightweight Spring interceptors to monitor components for exceptions. We saw no measurable performance penalty for the use of these interceptors.

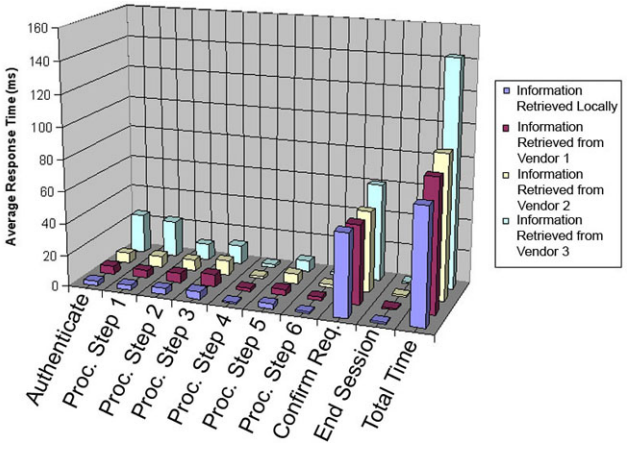


Fig. 5. Average Response Time for the Application

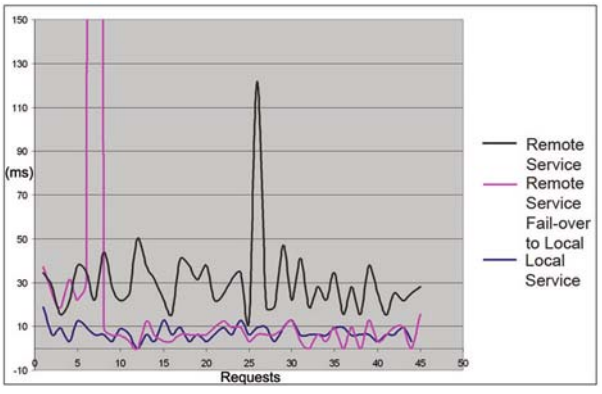


Fig. 6. Application Performance Before and After Healing

To determine how quickly the Refresh application could self-heal, we ran a further trial of Apache JMeter tests to simulate an additional 1,000 requests. During the experiment, we used fault injection to randomly simulate the failure of different services. The faults were injected by adding code to the local services to throw Java runtime exceptions that would force Refresh to heal the application by swapping remote services for the failed local services. After the local services were swapped to remote services, we randomly shutdown the remote services used by the application to force the failover to alternate remote services or back to a local service that had become available.

Over the tests, shutting down a failed subsystem and rebooting the container into a new configuration averaged roughly 140ms. The CSP technique required an average of an additional 10ms to find the new configuration to reboot into. When this result is compared to Figures 5, it can be seen that the healing time is slightly more than the average time to complete an order.

Figure 6 overlays the application's worst case response time using a local information provider, a remote information provider, and a remote information provider that is swapped back to a local provider because of a failure.

The failure of the remote service is easily discernible on request 7. There is also a visible slow down in the network but not a failure at request 25 of the remote service. Before the failure occurs, the application has the same average performance as the conventional application using a remote service. Once the failed service is healed, the application again has the same average performance as the conventional application with the local service. This result indicates that container-based healing incurs little or no pre- or post- healing performance penalties.

5.3 Filtered Cartesian Flattening vs. CSP-Based Configuration Derivation

The next set of experiments compared the scalability and speed of Filtered Cartesian Flattening versus a CSP-based configuration derivation technique. We

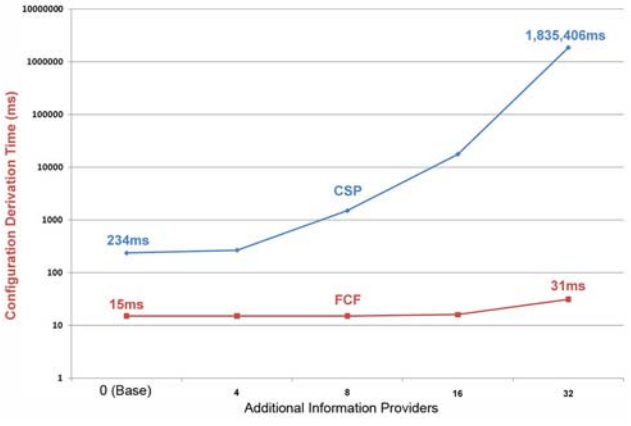


Fig. 7. Filtered Cartesian Flattening vs. CSP-based Configuration Derivation Time for the Application

extended the Refresh application's healing configuration to attempt to respect a bandwidth constraint while healing. Moreover, we directed the healing mechanism to also attempt to minimize the total cost consumed by the new configuration's services. Our CSP-based configuration solver was based on the Java Choco open source constraint solver [1].

First, we compared the time for Filtered Cartesian Flattening and the CSP-based techniques to derive a new configuration for the standard points of variability in the application. We then iteratively added 32 additional information providers to consider in the configuration derivation process. Both techniques found solutions for each size configuration problem. The results from this experiment are shown in Figure 7.

Initially, the CSP technique requires 234ms to configure the conventional application implementation with the additional resource constraints and bandwidth minimization goal. In the experiments presented in Section 5.2, the CSP-based technique was not required to adhere to a resource constraint. The new constraints and optimization goal cause a significant increase in the solving time to 234ms. Furthermore, by the time the 32 additional information providers were added into the configuration, the CSP-based technique required over 30 minutes (1,835,406ms) to derive a new configuration.

The speed at which a CSP solver can produce a solution is dependent on the complexity of the constraints in each CSP instance. To illustrate the increase in complexity of reconfiguring the application to find a valid solution versus reconfiguring to find an optimal solution that meets a resource constraint, we eliminated the resource constraints and optimization goals. We then resolved the Pet Store configuration problem with the simplified CSP that would produce a correct configuration but not one that necessarily respected resource constraints. The results are shown in Figure 8. Without the resource constraints and optimization, the 32 information provider problem instance that original took 1,835,406ms to solve only required 47ms. This result shows the significant increase in complexity that the re-

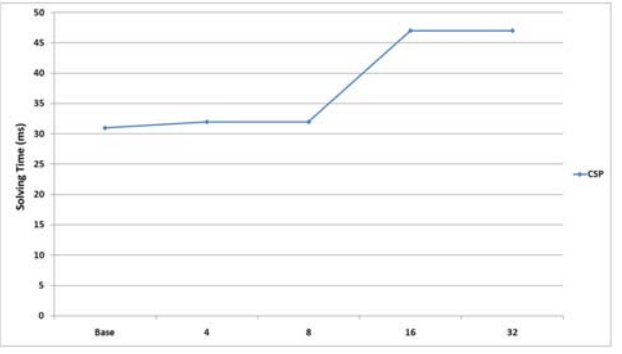


Fig. 8. CSP Solving Time without Resource Constraints & Optimization

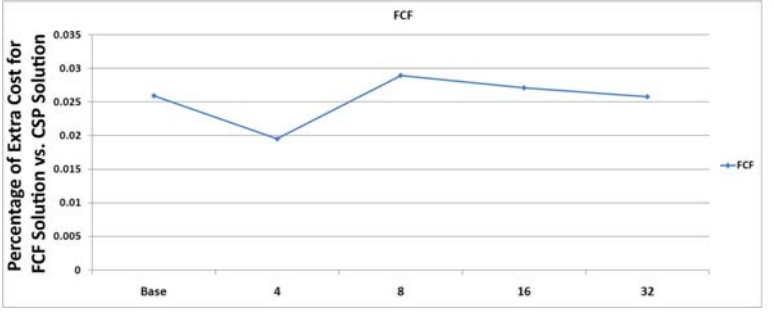


Fig. 9. FCF Solution Increased Cost as a Percentage of CSP Solution Cost

source constraints and optimization goal add. FCF's running time, in contrast, is not affected by adding or removing the bandwidth constraint.

The time for Filtered Cartesian Flattening to derive a new configuration across the different configuration sizes and incorporating resource constraints and optimization is shown by the red line in the lower part of Figure 7. Initially, Filtered Cartesian Flattening requires 15ms to derive a configuration, which is substantially less than the CSP-based technique's 234ms. Moreover, when the 32 additional providers are added, Filtered Cartesian Flattening is able to derive a configuration in 31ms, which is faster than the CSP technique can solve the problem without resource constraints. Filtered Cartesian Flattening's 31ms is many orders of magnitude less than the ∼30mins for the CSP-based technique. This result shows that Filtered Cartesian Flattening is significantly more scalable than the CSP-based technique for our case study application.

Another question that we sought to answer was how optimal the solutions produced by FCF were compared to the CSP-based solutions. We tracked the total cost of the external information providers chosen by the two techniques. We used both techniques to attempt to minimize the total cost of the information provider configurations. Figure 9 shows the increase in solution cost of choosing the FCF

technique over the CSP technique. As shown in Figure 9, the FCF technique produced solutions that ranged from roughly 2-3% more expensive. Thus, the dramatic reduction in solving time shown in Figure 7 came at a very low increase in overall solution cost.

5.4 Results Analysis and Comparison of FCF and CSP Reconfiguration

In this section, we provide an analysis of the pros and cons of our new Refresh + FCF approach versus the Refresh + CSP technique that we developed in prior work [25,23]. There are a number of criteria to consider when selecting whether to use CSP or FCF based reconfiguration with Refresh. We analyze the results of the experiments and the capabilities of the techniques along a number of critical axes. We evaluate the capabilities of each technique in terms of 1) scalability, 2) solution optimality, 3) tractability guarantees, and 4) amenability to different constraint types.

Solution Optimality/Quality: Refresh + CSP provides guaranteed optimal results. Refresh + FCF, in contrast, is a heuristic algorithm that does not provide guaranteed solution optimality. As we showed in the results depicted in Figure 9, FCF can provide near optimal results. Furthermore, other research results [19] have shown generally high optimalities of 90%+ optimal for the MMKP heuristic algorithms that can be used by FCF.

A key tradeoff to consider when evaluating Refresh with CSP versus FCF is that there is no hard guarantee on the optimality of the solutions generated by FCF. Furthermore, we are not aware of any fast runtime algorithms for producing a good estimate of the optimal solution value. Thus, there is no practical way at runtime to estimate the optimality of an FCF healing solution for Refresh. FCF does, however, guarantee that resource constraints are respected. If the goal is to find a solution that meets resource constraints with a "best effort" on solution optimality, then FCF can readily be applied.

Scalability: As shown in Figure 7, Refresh + FCF scales significantly better than Refresh + CSP when resource constraints must be adhered to in the reconfiguration process. As shown in Figure 8, if resource constraints are not included, Refresh + CSP can provide very fast configuration healing times. If scalability and resource constraints are a prime concern, Refresh + FCF should be used.

Whereas FCF does not provide a guarantee on solution optimality, it does provide a guarantee on scalability. FCF is a polynomial-time algorithm and hence will scale accordingly. CSP solvers use algorithms with exponential worst-case time. In some cases, as shown in Figure 8, these algorithms perform well. In other cases, such as in Figure 7, the same algorithms can perform poorly. If guarantees on healing speed are more important than guarantees on solution optimality, the unpredictability of CSP-based healing is not appropriate.

Tractability Guarantees: FCF does not guarantee that a configuration will be found. If FCF cannot find a viable configuration, it is still possible that one exists

that eluded the FCF heuristics. Refresh + CSP does guarantee that a viable solution will be found if it exists. The down side is that, for large enough problems, the CSP technique could take longer to find the solution than fixing the root cause of the original failure.

Although FCF does not guarantee a solution is found, its solving speed provides a number of potential options to developers. If reconfiguration speed is critical, developers can instruct Refresh to first search for a configuration using FCF and then to fallback to CSP if no solution is found. As shown in Figure 7, FCF can have runtimes in the tens of milliseconds. Thus, an initial attempt at finding a reconfiguration solution using FCF should add minimal overhead to the healing process.

Amenability to Different Constraint Types: FCF is designed to work with feature models that primarily have a tree-like structure. As the number of cross-tree constraints in a feature model increase, FCF typically becomes less and less effective at finding solutions. That is, FCF is good when the resource constraints are more restrictive than the cross-tree constraints.

CSP, in contrast, tends to operate better when the cross-tree constraints in a feature model are more restrictive than the resource constraints. A CSPs constraint network and propagation algorithms can leverage cross-tree constraints to quickly eliminates large portions of the solution space. Thus, for applications with feature models including large numbers of cross-tree constraints that have a less tree-like structure, Refresh + CSP is most appropriate.

6 Related Work

This section compares our work on Refresh and Filtered Cartesian Flattening with other research work. First, we compare Refresh to the original technique of microrebooting. Next, we compare and contrast Refresh with other feature-based self-adaptive healing techniques. Finally, we compare Refresh to other non-feature based self-adaptive healing techniques.

Microrebooting Related Work. Refresh is based on the idea of Microrebooting [8]. Microrebooting restarts individual components or collections of components to fix an error. The number of components that are rebooted continues to grow if successive reboots do not eliminate the error. Microrebooting can help to eliminate some types of problems but may not fix all issues.

For example, if one of the information provider services fails, restarting the application will not fix the error since it is in a remote component. Instead, the local application must be rebooted in an alternate configuration to eliminate the error. As we showed in Section 3.1 determining how to eliminate failed components is a challenging problem. Refresh uses the Filtered Cartesian Flattening algorithm to eliminate this problem by dynamically deriving a new application configuration to reboot the failed subsystem into. This type of reconfiguration and rebooting can eliminate both local errors and references to failed remote services, which microrebooting alone cannot fix.

Feature-Based Healing Related Work. Other approaches to application healing have been developed that leverage a combination of goal modeling and feature models [16]. In the approach by Lapouchnian et al. feature models are used to find points of variation in the application. The application adaptation is driven by Statecharts. As we showed in Section 3.1, specifying the logic to solve the NP-Hard problem of reconfiguring the application subject to resource constraints is hard to implement in either Java, C++, or Statecharts.

The approach of using Statecharts to drive the adaptive healing of the application burdens enterprise application developers with a extremely complex problem. Moreover, there can be an exponential number of states that may need to be modeled to properly adapt in all resource availability scenarios. In contrast, Refresh does not require an explicit adaptation plan but instead a model of how the application can be reconfigured. Refresh then automates the complex problem of deriving a new application configuration that fits the current available resources.

Self-Adaptive Healing Related Work. Other approaches also use the idea of identifying error conditions and then planning adaptation actions that should be triggered [7,14,11,6,2,16,12]. These approaches also require developers to handle the complex problem of determining how to best adapt the application's configuration while adhering to a resource constraint. Determining how to reconfigure in the face of a resource constraint is an NP-Hard problem. In contrast, Refresh automates this recovery logic by using the Filtered Cartesian Flattening approximation algorithm to derive a new application feature set that can be used to continue functioning.

7 Concluding Remarks

A common approach to simplifying the development of self-adaptive healing applications is to use a model of an application's adaptation logic to generate self-adaptive healing code or guide self-adaptive healing at runtime [7,14,11,6,2,16,12]. This approach to simplifying the development of self-adaptive healing applications does not, however, eliminate the key complexity, which is the logic needed to deduce how to heal the application. Moreover, when resource constraints must be considered in the adaptation process, determining how to adapt the application without exceeding the resource limitations is an NP-Hard problem.

This paper showed how our Refresh technique—based on a combination of microrebooting and dynamic reconfiguration using feature models—can simplify the development of self-adaptive healing applications. Rather than simply rebooting in the same configuration (which could cause errors involving remote services to persist), Refresh dynamically derives a new application configuration to reboot into using the application's feature model. Moreover, we showed that by using the FCF algorithm to perform the derivation of the new feature selection, Refresh could respect resource constraints and still find alternate feature configurations fast.

The following list presents the lessons we have learned from our experiences building self-adaptive healing enterprise applications using Refresh:

- **CSP-based reconfiguration techniques are sufficient if no resource constraints are present.** If resource constraints are not considered in the reconfiguration process, CSP and other exact techniques, such as SAT solvers, provide sufficient performance to derive new configurations. Only when resource constraints are added is FCF needed.
- **Container lifefcycle methods can managing accidental healing complexities.** Containers must be able to release resources, roll back transactions, and perform other cleanup whenever an application container is shutdown. By reusing this lifecycle mechanism to perform healing, significant accidental complexity is managed by the container on the developer's behalf.
- **Optimization goals may not be easy to formalize.** In many domains, resource constraints and optimization goals can be hard to formalize since it is not clear how choosing one service over another affects cost and resource consumption. Interactions between organizations, however, often do have a known resource consumption and cost associated with them.

Refresh is available in open-source form as part of the *GEMS Model Intelligence* project at `www.sf.net/projects/gems`.

References

1. Choco constraint programming system, `http://choco.sourceforge.net/`
2. Barbier, F.: MDE-based Design and Implementation of Autonomic Software Components. In: 5th IEEE International Conference on Cognitive Informatics, 2006. ICCI 2006, vol. 1 (2006)
3. Barki, H., Rivard, S., Talbot, J.: Toward an assessment of software development risk. Journal of Management Information Systems 10(2), 203–225 (1993)
4. Batory, D.: Feature Models, Grammars, and Prepositional Formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
5. Benavides, D., Trinidad, P., Ruiz-Cortés, A.: Automated Reasoning on Feature Models. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 491–503. Springer, Heidelberg (2005)
6. Bhat, V., Parashar, M., Liu, H., Khandekar, M., Kandasamy, N., Abdelwahed, S.: Enabling Self-Managing Applications using Model-based Online Control Strategies. In: Proceedings of the 3rd IEEE International Conference on Autonomic Computing, Dublin, Ireland (June 2006)
7. Calinescu, R.: Model-Driven Autonomic Architecture. In: Proceedings of the 4th IEEE International Conference on Autonomic Computing, Jacksonville, Florida, USA (June 2007)
8. Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., Fox, A.: Microreboot-a technique for cheap recovery. In: Proceedings of the 6th Symposium on Operating Systems Design and Implementation, pp. 31–44 (2004)
9. Czarnecki, K., Antkiewicz, M., Kim, C., Lau, S., Pietroszek, K.: FMP and FMP2RSM: Eclipse Plug-ins for Modeling Features Using Model Templates. In: Conference on Object Oriented Programming Systems Languages and Applications, pp. 200–201 (October 2005)
10. Oppenheimer, D.P.D., Ganapathi, A.: Why do Internet Services Fail, and What can be Done about It? In: Proceedings of the USENIX Symposium on Internet Technologies and Systems (March 2003)

11. Denaro, G., Pezze, M., Tosi, D.: Designing Self-Adaptive Service-Oriented Applications. In: 4th IEEE International Conference on Autonomic Computing, Jacksonville, Florida (June 2007)
12. Elkorobarrutia, X., Izagirre, A., Sagardui, G.: A Self-Healing Mechanism for State Machine Based Components. In: Proceedings of the 1st International Conference on Ubiquitous Computing: Applications, Technology and Social Issues, Alcalá de Henares, Madrid, Spain (June 2006)
13. Johnson, R., Hoeller, J.: Expert one-on-one J2EE development without EJB. Wrox (2004)
14. Joshi, K., Sanders, W., Hiltunen, M., Schlichting, R.: Automatic Model-Driven Recovery in Distributed Systems. In: The 24th IEEE Symposium on Reliable Distributed Systems (SRDS 2005), pp. 25–38 (2005)
15. Kang, K., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M.: FORM: A feature-; oriented reuse method with domain-; specific reference architectures. Annals of Software Engineering 5, 143–168 (1998)
16. Lapouchnian, A., Liaskos, S., Mylopoulos, J., Yu, Y.: Towards Requirements-driven Autonomic Systems Design. In: Proceedings of the 2005 workshop on Design and evolution of autonomic application software, pp. 1–7 (2005)
17. Linberg, K.: Software developer perceptions about software project failure: a case study. The Journal of Systems & Software 49(2-3), 177–192 (1999)
18. Mannion, M.: Using First-order Logic for Product Line Model Validation. In: Chastek, G.J. (ed.) SPLC 2002. LNCS, vol. 2379, pp. 176–187. Springer, Heidelberg (2002)
19. Mostofa Akbar, M., Sohel Rahman, M., Kaykobad, M., Manning, E., Shoja, G.: Solving the Multidimensional Multiple-choice Knapsack Problem by constructing convex hulls. Computers and Operations Research 33(5), 1259–1273 (2006)
20. Schach, S.: Object-oriented and classical software engineering. McGraw-Hill Higher Education, Boston (2005)
21. Trinidad, P., Benavides, D., Durán, A., Ruiz-Cortés, A., Toro, M.: Automated error analysis for the agilization of feature modeling. Journal of Systems and Software (in press) (2007)
22. White, J., Czarnecki, K., Schmidt, D.C., Lenz, G., Wienands, C., Wuchner, E., Fiege, L.: Automated Model-based Configuration of Enterprise Java Applications. In: EDOC 2007 (October 2007)
23. White, J., Dougherty, B., Schmidt, D.: Filtered Cartesian Flattening. In: Workshop on Analysis of Software Product-Lines at the International Conference on Software Product-lines (October 2008)
24. White, J., Nechypurenko, A., Wuchner, E., Schmidt, D.C.: Optimizing and Automating Product-Line Variant Selection for Mobile Devices. In: 11th International Software Product Line Conference (September 2007)
25. White, J., Strowd, H., Schmidt, D.C.: Creating Self-healing Service Compositions with Feature Modeling and Microrebooting. In: The International Journal of Business Process Integration and Management (IJBPIM), Special issue on Model-Driven Service-Oriented Architectures (2008)
26. Whittaker, B.: What went wrong? Unsuccessful information technology projects. Information Management And Computer Security 7, 23–29 (1999)

Author Index