Lecture Notes in Computer Science 5846

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

Editorial Board

David Hutchison
Lancaster University, UK

Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler
University of Surrey, Guildford, UK

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

Alfred Kobsa
University of California, Irvine, CA, USA

Friedemann Mattern
ETH Zurich, Switzerland

John C. Mitchell
Stanford University, CA, USA

Moni Naor
Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz
University of Bern, Switzerland

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

Bernhard Steffen
University of Dortmund, Germany

Madhu Sudan
Microsoft Research, Cambridge, MA, USA

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

Doug Tygar
University of California, Berkeley, CA, USA

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

Jeremy Gibbons José Nuno Oliveira (Eds.)

Teaching Formal Methods

Second International Conference, TFM 2009
Eindhoven, The Netherlands, November 2-6, 2009
Proceedings

Springer

Volume Editors

Jeremy Gibbons
Oxford University
Computing Laboratory
Wolfson Building
Parks Road
Oxford OX1 3QD, UK
E-mail: jeremy.gibbons@comlab.ox.ac.uk

José Nuno Oliveira
Universidade do Minho
Departamento de Informática
Campus de Gualtar
4710-057 Braga, Portugal
E-mail: jno@di.uminho.pt

Library of Congress Control Number: 2009935818

CR Subject Classification (1998): D.2.4, D.3.1, F.4.3, D.4.5, F.3.1, I.2.2, G.4

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-642-04911-7 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-04911-8 Springer Berlin Heidelberg New York

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

springer.com

© Springer-Verlag Berlin Heidelberg 2009
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12774869 06/3180 5 4 3 2 1 0

Preface

This volume contains the proceedings of TFM 2009, the Second International FME Conference on Teaching Formal Methods, organized by the Subgroup of Education of the Formal Methods Europe (FME) association. The conference took place as part of the first Formal Methods Week (FMWeek), held in Eindhoven, The Netherlands, in November 2009.

TFM 2009 was a one-day forum in which to explore the successes and failures of formal method (FM) education, and to promote cooperative projects to further education and training in FMs. The organizers gathered lecturers, teachers, and industrial partners to discuss their experience, present their pedagogical methodologies, and explore best practices.

Interest in FM teaching is growing. TFM 2009 followed in a series of events on teaching FMs which includes two BCS-FACS TFM workshops (Oxford in 2003, and London in 2006), the TFM 2004 conference (Ghent, 2004, with proceedings published as Springer LNCS Volume 3294), the FM-Ed 2006 workshop (Hamilton, co-located with FM 2006), FORMED (Budapest, at ETAPS 2008), and FMET 2008 (Kitakyushu, co-located with ICFEM 2008).

FMs have an important role to play in the development of complex computing systems—a role acknowledged in industrial standards such as IEC 61508 and ISO/IEC 15408, and in the increasing use of precise modelling notations, semantic markup languages, and model-driven techniques. There is a growing need for software engineers who can work effectively with simple, mathematical abstractions, and with practical notions of inference and proof.

Original contributions were solicited providing insight, opinions, and suggestions for courses of action regarding the teaching of FMs, including but not limited to the following aspects: experiences of teaching FMs, both successful and unsuccessful; educational resources including the use of books, case studies and the Internet; the education of weak and mathphobic students; the integration, or otherwise, of FMs into the curriculum, including contributions to the definition of a Formal Methods Body of Knowledge (FMBoK); the advantages of FM-trained graduates in the workplace; changing attitudes towards FMs in students, academic staff, and practitioners; the necessary mathematical background.

There were 19 submissions, each being reviewed by at least four Programme Committee members. Ten papers were accepted, some of which were subject to a round of shepherding before completion. Two of the accepted papers addressed the conference theme of 'widening access to FMs' in terms of reaching students with limited enthusiasm for FMs as an end in themselves: Catano and Rueda describe two courses taught in Colombia—one on JML and one on B—that venture into the 'unconquered territory' of conventional software companies, and Ölveczky discusses an approach based on rewriting logic using the Maude tool.

Two more papers address 'widening access' in a different dimension, extending that access beyond traditional university education: Ferreira et al. describe the MathIS project that aims to reinvigorate high school mathematics teaching, and Ishikawa et al. present some of their experiences with the Top SE programme of education for professional software engineers in Japan.

The next two papers describe synergistic combinations of similar FMs: Tarkan and Sazawal on using Alloy as a tool in the teaching of Z, and Poll on the combination of the JML specification language and the ESC/Java verification tool. Two more papers describe teaching approaches based on the integration of complementary FMs: Hallerstede and Leuschel on integrating model-checking with formal proof, and Ahrendt et al. on integrating formal and informal approaches to verification. Finally, the last two papers illustrate the typical evolution of an education programme in FMs: from graduate-level courses in new fields, as discussed by Kofroň et al., in which the state of the art is rapidly evolving and the only materials available are primary sources, through a decade or two of development, to more mainstream undergraduate courses in more mature fields, in which secondary sources such as the textbook described by Aceto et al. may be used.

The programme commenced with an invited talk by Jeff Kramer on abstraction and modelling, arguing that these two crucial skills for software engineers are complementary partners. The conference closed with a panel discussion on the idea of building a *Guide to the Formal Methods Body of Knowledge* (FMBoK), inspired by similar efforts for software engineering (SWEBoK) and for project management (PMBoK); such a resource would provide guidance to teachers, managers, and developers on what should be expected from a comprehensive, balanced programme of education in FMs.

During the electronic Programme Committee meeting, there was a spirited discussion about the intent and extent of events of this kind; we feel this discussion is worth summarizing here. We are convinced that the papers included in this volume will be useful to those teaching FMs: reporting on experiences; describing approaches that worked, and those that didn't; identifying common issues, to increase our understanding of which aspects are generic and which specific to a particular context; and generally providing inspiration for others. Nevertheless, it will be clear to readers that the papers do not present detailed qualitative or quantitative data collected during robust, repeatable, scientific experiments; the authors, the Programme Committee, and the intended audience may be leading researchers in FMs, but they are by and large mere practitioners in pedagogy. Formal assessments of the efficacy of particular approaches to teaching FMs would be valuable in informing our teaching, but they are very difficult to conduct—we leave this as a challenge for future events.

The conference was managed, and these proceedings were prepared, using the EasyChair conference management system (`http://www.easychair.org/`), whose valuable service we are happy to acknowledge.

We are very grateful to the members of the Programme Committee and their additional referees for their care and diligence in reviewing the submitted papers. We are also grateful to Tijn Borghuis and Erik de Vink, FMWeek coordinators, for their help and support, and to the sponsoring institutions.

August 2009

Jeremy Gibbons
José Oliveira

Organization

TFM 2009 was organized by the Subgroup of Education of the Formal Methods Europe (FME) association, in close collaboration with the organization of the *Formal Methods Week* (*FMWeek*).

Programme Committee

Izzat Alsmadi	North Dakota State University, USA
Dines Bjørner	IIMM Institute, Denmark
Eerke Boiten	University of Kent, UK
Raymond Boute	Ghent University, Belgium
Andrew Butterfield	Trinity College Dublin, Ireland
Jim Davies	University of Oxford, UK
David Duce	Oxford Brookes University, UK
John Fitzgerald	University of Newcastle upon Tyne, UK
Jeremy Gibbons	University of Oxford, UK
Randolph Johnson	National Security Agency, USA
Mícheál Mac an Airchinnigh	Trinity College Dublin, Ireland
Dino Mandrioli	Politecnico di Milano, Italy
José Oliveira	University of Minho, Portugal
Kees Pronk	Delft University of Technology, The Netherlands
Bernhard Schätz	Technische Universität München, Germany
Wolfgang Schreiner	Johannes Kepler University Linz, Austria
Simão Sousa	University of Beira Interior, Portugal
Kenji Taguchi	National Institute of Informatics, Japan
Jeannette Wing	Carnegie-Mellon University, USA

Referees

Besides the members of the Programme Committee, the following external referees contributed to the paper reviewing process:

André Brito Passos	David Pereira	Ian Bayley
Clare Martin	Fuyuki Ishikawa	Radu Calinescu
Cyrille Artho	Henrique Costa	

Sponsoring Institutions

Formal Methods Europe Association (FME)
Software Improvement Group (SIG), Amsterdam, The Netherlands

Table of Contents

Abstraction and Modelling: A Complementary Partnership

Jeffrey Kramer

Department of Computing
Imperial College London
Huxley Building
180 Queen's Gate
London SW7 2AZ, U.K.
j.kramer@imperial.ac.uk

Abstract. Why is it that some software engineers are able to produce clear, elegant designs and programs, while others cannot? Is it purely a matter of intelligence? What is the problem? One hypothesis is that the answer lies in abstraction: the ability to exhibit abstraction skills and perform abstract thinking and reasoning. Abstraction is a cognitive means by which engineers, mathematicians and others deal with complexity. It covers both aspects of removing detail as well as the identification of generalisations or common features, and has been identified as a crucial skill for software engineering professionals. Is it possible to improve the skills and abilities of those less able through further education and training? Are there any means by which we can measure the abstraction skills of an individual?

In this talk, we explore these questions, and argue that abstraction and modelling are complementary partners: that abstraction is the key skill for modelling and that modelling provides a sound means for practising and improving abstraction skills.

J. Gibbons and J.N. Oliveira (Eds.): TFM 2009, LNCS 5846, p. 1, 2009.
© Springer-Verlag Berlin Heidelberg 2009

Teaching Formal Methods for the Unconquered Territory

Nestor Catano[1] and Camilo Rueda[2]

[1] University of Madeira,
Department of Mathematics and Engineering Funchal, Portugal
ncatano@uma.pt
[2] Pontificia Universidad Javeriana,
Department of Computer Science Cali, Colombia
crueda@cic.puj.edu.co

Abstract. We summarise our experiences in teaching two formal methods courses at Pontificia Universidad Javeriana. The first course is a JML-based software engineering course. The second course is a model-driven software engineering course realised in the B method for software development. We explain how formal methods are promoted in Pontificia Universidad Javeriana, how we motivate students to embrace formal methods techniques, and how they are promoted through the presentation of motivating examples.

1 Introduction

In this paper we summarise our experiences in teaching formal methods to undergraduate students of Pontificia Universidad Javeriana. We strive to help students to build skills on formal methods, and to master formal tools they might use in their future IT software engineering jobs in the "unconquered territory", *e.g.*, traditional software engineering companies that are reluctant to adopt formal methods as part of their software development practices. The first author has been lecturing a JML [7] (short for Java Modeling Language) software engineering course during the past 3 years, and the second author a model-based formal software development course during the past 5 years. Prior to these courses, students must attend a standard software engineering course, as well as discrete mathematics related courses. That is, our students have enough software engineering and mathematical background to appreciate the benefits of using formal methods tools and techniques during software development. Formal methods offer a practical alternative to constructing correct software, and can be implanted along the several steps of the software development cycle, from requirements capture, by employing formal specification language notations such as B [1] or Z [16,17], through the design and coding of systems [14]. But, formal methods will only be widely popular in software companies if formal methods tools exist that provide support for software engineering practices, and software engineers are properly trained in related techniques in universities and education centres.

J. Gibbons and J.N. Oliveira (Eds.): TFM 2009, LNCS 5846, pp. 2–19, 2009.
© Springer-Verlag Berlin Heidelberg 2009

We explain here two approaches undertaken to the teaching of formal methods in Pontificia Universidad Javeriana. (*i.*) Model-driven software engineering is realised in the B method for software development [1]. That is, from a B model of a system, obtained from the software requirements of an application, a code-level model in B is attained while applying successive *refinement steps*. Mathematical techniques are used along the way so as to guarantee the correctness of the refinement process. The code-level model is a concrete model of the more abstract initial model of the system. Section 5 presents the model-driven B-based course taught at Pontificia Universidad Javeriana, Cali. The course is illustrated with the formal software development of the control system structure of the recently inaugurated Cali mass transportation system (MIO). (*ii.*) Section 4 presents the structure of our JML-based formal software development course. The JML-based course places between the rather heavy, fully mathematically based, formal software development course in B, and the rather mathematically informal previous software engineering course, introduced in Section 3. The formal methods courses in JML and B are two complementary courses that give students two different insights of the use of formal methods in software development. In the following, Section 2 gives an overview on JML and B based formal methods techniques and tools.

2 Preliminaries on Formal Methods

2.1 The Java Modeling Language (JML)

JML is a specification language for Java that provides support for B. Meyer's design-by-contract principles [11]. The idea behind the design-by-contract methodology is that a contract between a class and its clients exists. The client must guarantee certain conditions, called pre-conditions, to be able to call a method of the class. In return, the class must guarantee certain conditions, called post-conditions, that will hold after the method is called.

JML specifications use Java syntax, and are embedded in Java code within special marked comments `/*@ ... @*/` or after `//@`. A simple JML specification for a Java class consists of pre- and post-conditions added to its methods, and class invariants restricting the possible states of class instances. Specifications for method pre- and post-conditions are embedded as comments immediately before method declarations. JML predicates are first-order logic predicates formed of side-effect free Java boolean expressions and several specification-only JML constructs. Because of this side-effect restriction, Java operators like `++` and `--` are not allowed in JML specifications. JML provides notations for forward and backward logical implications, `==>` and `<==`, for non-equivalence `<=!=>`, and for logical *or* and logical *and*, `||` and `&&`. The JML notations for the standard universal and existential quantifiers are `(\forall T x; E)` and `(\exists T x; E)`, where `T x;` declares a variable `x` of type `T`, and `E` is the expression that must hold for every (some) value of type `T`. The predicates `(\forall T x; P; Q)` and `(\exists T x; P; Q)` are equivalent to `(\forall T x; P ==> Q)` and `(\exists T x; P && Q)` respectively.

JML supports the use of several mathematical types such as sets, sequences, functions and relations, in specifications. JML specifications are inherited by subclasses – subclass objects must satisfy super-class invariants, and subclass methods must obey the specifications of all super-class methods that they override. This ensures behavioural sub-typing. That is, a subclass object can always be used (correctly) where a super-class object is expected. In the following, we briefly review JML specification constructs, and the JML common tools. The reader is invited to consult [9] for a full introduction to JML.

2.2 The JML Common Tools

The JML common tools [7,6] is a suite of tools providing support to run-time assertion checking of JML-specified Java programs. The suite includes `jmlc`, `jmlunit` and `jmlrac`. The `jmlc` tool compiles JML-specified Java programs into a Java byte-code that includes instructions for checking JML specifications at run-time. The `jmlunit` tool generates JUnit [10] unit tests code from JML specifications and uses JML specifications processed by `jmlc` to determine whether the code being tested is correct or not. Test drivers are run by using the `jmlrac` tool, a modified version of the `java` command that refers to appropriate run-time assertion checking libraries.

The JML common tools make possible the automation of regression testing from the precise, and correct JML characterisation of a software system. The quality and the coverage of the testing carried out by JML depend on the quality of the JML specifications. The run-time assertion checking with JML is sound, *i.e.*, no false reports are generated. The checking is however incomplete cause users can write informal descriptions in JML specifications, e.g., `(* x is positive *)`. The completeness of the checking performed by JML depends on the quality of the specifications and the test data provided.

2.3 Refinement Calculus and the B Method

In the *refinement* calculus strategy for software development, the process of going from a system specification to its implementation in a machine goes through a series of stages. Each stage adds more details to the description of a system. Each stage can thus be seen as a model of the system at a particular level of abstraction. Models at each level serve different purposes. At higher levels models are used to state and verify key system properties. At lower levels models are used to simulate the system behaviour. It is crucial that models at each stage are coherent with the system specification, *i.e.*, that the simulation obeys the specification properties. A model M_{i+1} at stage $i+1$ is said to be a refinement of a model M_i at stage i when the states computed by M_i and M_{i+1} at each given step obey a so-called "gluing invariant" stating properties for the joint behaviour of both models. A refinement step generates *proof obligations* that must be formally verified in order to assert that a model M_{i+1} is indeed a refinement of a model M_i. These are necessary and sufficient conditions to guarantee that, although at different levels of abstraction, both are models of the same system. Correctness of the whole development process is thus ensured ([4]).

In the B method ([1], [15]) models are so-called *machines* composed of a static part defining observations (variables, constants, parameters, etc) of the system and their invariant properties, and a dynamic part defining operations changing the state of the system. Each operation must maintain the invariant property. In B, the language for stating properties (essentially predicate logic plus set theory) and the language for specifying dynamic behaviour (*i.e.* programs) are seamlessly integrated.

A significant feature of the B system modelling approach is the availability of automatic verification tools such as B-tools (http://www.b-core.-com/btool.html), Atelier-B (http://www.atelierb.eu/index_en.html) o r Rodin (http://www.event-b.org/platform.html), and model-checking simulators such as ProB (http://users.ecs.soton.ac.uk/mal/systems/prob.html).

2.4 Event B Models

We introduce a derivative of the B method called event B [3]. Event B models are complete developments of discrete transition systems. They are composed of *machines* and *contexts*. These correspond, roughly, to a B method *machine* whose static part (except variables and their invariants) is transferred to a different module (the context). B method operations are replaced in event B machines by *events*. In B method machines, operations are *invoked*, either by a user or by another machine, whereas in event B, an event *occurs* when some condition (its *guard*) holds. Three basic relations are used to structure a model. A machine *sees* a context and can *refine* another machine. A context can *extend* another context. Events have two forms, as shown in table 1.

Table 1. Events

any x **where** $G_2(x, v)$ **then** $A_2(x, v)$ **end**	**when** $G_1(v)$ **then** $A_1(v)$ **end**

The "when" form of event executes the action A_1 when the current value of the system variables v satisfies the guard G_1. The "any" form of event executes action A_2 when there exists some value of x satisfying the guard G_2. Proofs obligations require invariants to hold after executing the actions. A simple example, modelling a parking lot, is shown in table 2. Variable C keeps track of the cars in the parking lot. Set $CARS$ defines a type. Constant n is the maximum capacity of the parking lot. Events model entrance and exit from the parking.

Table 2. A parking lot model

machine
Parking
sees
ctx
variables
C
invariants
$C \subseteq CARS \wedge card(C) \leq n$
events
arrives =
any *car*
where
$car \in CARS \wedge car \notin C \wedge \ card(C) < n$
then
$C := C \cup \{car\}$
end
leaves =
any *car* **where** $car \in C$
then
$C := C \backslash \{car\}$
end
end

context
ctx
sets
CARS
constants
n
axioms
$n \in \mathbb{N} \wedge n > 0$
end

3 The Software Engineering Programme at Pontificia Universidad Javeriana

Engineering in computer science at Pontificia Universidad Javeriana (called systems engineering for historical reasons) is a 5 year program, structured per semesters, comprising a 2 years' common trunk in physics and mathematics, shared by all other engineering programs, including 2 courses in discrete mathematics and logic. This is followed by a 3 years' program in computer science, with about a 30% course charge in economics and humanities. The discrete mathematics, as well as all the basic computer related courses comply with the ACM/IEEE undergraduate computer science curriculum (`http://sites.computer.org/ccse/`). There are two basic courses in software engineering:

1. Software processes dealing with fundamental design principles, API design, tool design, software life cycle and capability models.
2. Software engineering and management, dealing with software requirements specifications (SRS), project management, validation and verification, and software evolution.

These courses are scheduled for sixth and seventh semester undergraduate students respectively. A formal software development course is given in the sixth

semester, at the same time as the software processes course. Additionally, there is a case-based software engineering workshop scheduled for eighth semester students. Topics for this workshop vary, yet it has often been the case that the workshop has been dedicated to software verification with JML.

Formal model courses are not part of the core in the ACM/IEEE curriculum. We decided to include formal model courses in Pontificia Universidad Javeriana engineering program mostly for two reasons: (*i.*) a survey among recruiting executives of companies in Cali revealed they consider the ability to clearly reason about a software design as a key (usually lacking) competence in young professionals, and (*ii.*) the economic development plan of Cali pointed at software production as a key strategy and increasing software quality as the most pressing need in this realm.

Pontificia Universidad Javeriana computer science department keeps a close relations with ParqueSoft, the biggest Colombian technological cluster, with more than 200 software companies and 800 software developers (`http://www.-parquesoft.com`). ParqueSoft software companies are typically launched by students from the three biggest universities in Cali. ParqueSoft's total sales are about US $47 million a year. About half of these companies have achieved standard quality assurance certifications. Most computer science students at Pontificia Universidad Javeriana conduct internships in these and other companies based in Cali during their third year of studies. Pontificia Universidad Javeriana Computer Science department encourages students to initiate software start-ups at ParqueSoft through an entrepreneurship joint educational program. Students in this program substitute their engineering degree final thesis report with a technical report on their proposed software venture.

4 The JML-Based Software Engineering Course

This course covers conceptual underpinnings of formal software development and software verification with JML [7,6]. We evolve our course through several software development examples that illustrate our methodology. We employ The JML common tools, and the Prototype Verification System (PVS) [13] to validate the developments (see Section 2). We start this course with an overview of first order logic and proof systems. Then, program correctness using Dijkstra's weakest pre-condition calculus is introduced, followed by the design-by-contract, and software verification using the JML common tools. In the end of the course, the PVS specification and verification system is presented with the aid of the electronic phone book example that comes with the PVS standard documentation. The course outline follows.

1. An Introduction to Formal Methods **(3 hours)**
2. First Order Logic **(6 hours)**
   (a) Syntax and Semantics
   (b) Expressiveness, Models and Tautologies
3. The Java Modelling Language (JML) **(24 hours)**

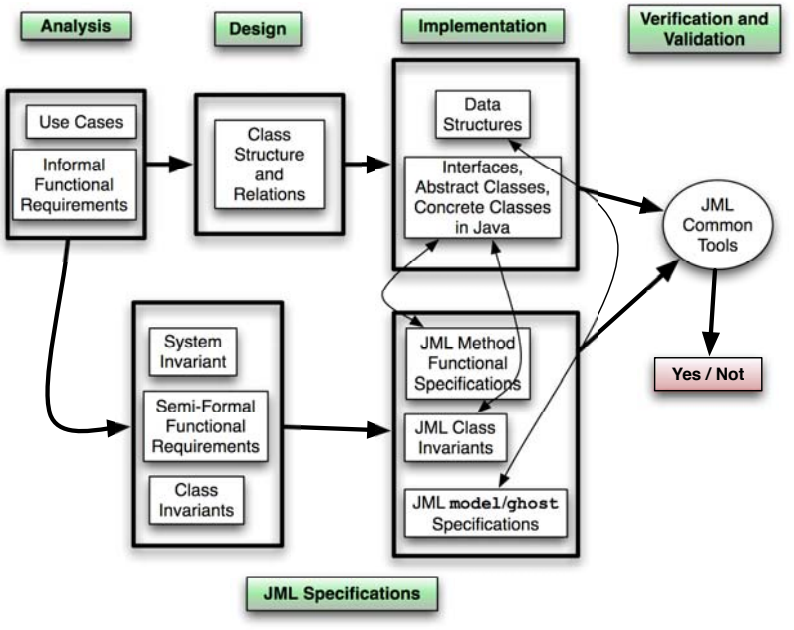


Fig. 1. JML-based Software Engineering

(a) Program Correctness and Weakest Pre-condition Calculus
(b) Design-by-Contract
(c) JML and The Design-by-Contract
(d) JML's Advanced Features
(e) The JML Common Tools
(f) Formal Development Examples

4. Proof Systems **(3 hours)**
   (a) Decidability, Soundness, Completeness
5. The PVS Specification and Verification System **(15 hours)**
   (a) The Logic of PVS
   (b) An Electronic Phone Book Example in PVS
   (c) Types, Declarations, Induction, Recursion
   (d) Encoding Abstract State Machines in PVS
   (e) JML-Checked Java Implementation of a PVS Specification of B-Trees

The JML-based software development methodology introduced in the course is based on Meyer's object oriented methodology presented in [12] (see Chapter 28), while using the JML common tools for validating the developments. The methodology is illustrated in Figure 1. The software development cycle consists of four steps, namely, analysis, design, implementation, and verification. In the same spirit of the methodology introduced by Meyer, we do not restrict any step of the software development cycle to occur before or after any other step (as opposed to the

Waterfall model introduced in 1970's), so that the arrows in Figure 1 convey information on usage rather than on precedence in time. JML specifications (lower part of the figure) are written in parallel to the Java application itself (upper part of the figure). The informal functional requirements (functional requirements written in English) of the step of analysis originate three documents that will later serve to write the JML specifications describing the application, namely, the semi-formal functional requirements, the invariant of each class, and the invariant of the whole system. The semi-formal functional requirements, though written in English, are expressed in a more mathematical style, suitable to be then ported into JML specifications. Class invariants and the system invariant are naturally expressed as JML invariants. And the semi-formal functional requirements are expressed as JML method pre- and post-conditions. At the same time, JML invariants and the JML method specifications reflect the class structure of the Java code.

Additionally, in order to have a higher level of abstraction in specifications, JML allows one to declare so-called `model` variables. These are variables that exist only at the level of the specifications. Model variables can be related to concrete variables (*i.e.*, variables declared in Java code) by the use of `represents` and `depends` clauses, specifying how the value of a model variable can be calculated from the values of the concrete variables. JML model variables allow one to describe in full detail the data structures used in a Java program, and how these structures evolve through class inheritance.

4.1 Formal Software Development of Ax-LIMS with JML

In the following, we explain the JML-based software development methodology described in Figure 1 by presenting key aspects in the development of Ax-LIMS [8], a project management Java plug-in wrapping up in Java most services and features provided by LIMS (Laboratory Information Management System), a project management application, developed in PHP, specially designed for the planning, organisation and resource management of biotechnology projects (see Figure 2). Biotechnology projects manage laboratory processes and tasks. A process manages laboratory experiments. Tasks are administrative tasks that need to be carried out between the project start and completion dates. Information about projects is stored in a PostGreSQL database. Although LIMS business logic is written in PHP, a REST (Representation State Transfer) API interface exists that supports communication between Ax-LIMS and LIMS. Ax-LIMS implementation relies on the project and process features provided by LIMS. The Ax-LIMS example is adapted from a formal software development project involving ParqueSoft (`http://www.parquesoft.com`), the International Centre for Tropical Agriculture (CIAT, `http://www.ciat.cgiar.org/`), and Pontificia Universidad Javeriana faculty.

Students in our course are asked to formally develop the project manager plug-in that directly connects to LIMS. Students are given a software requirements document, describing the functional requirements of the Ax-LIMS plug-in, as well as a set of use cases describing the interaction of the user with the plug-in.

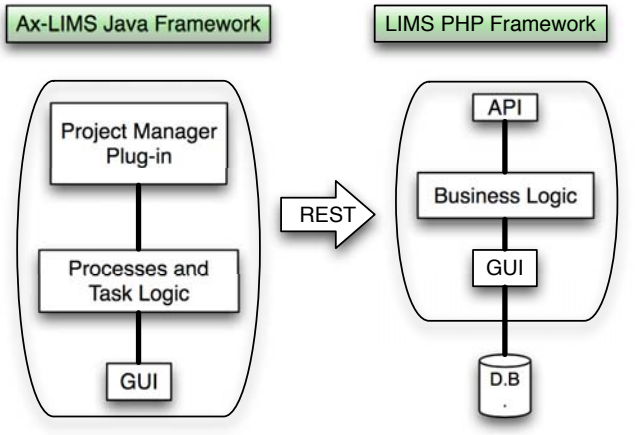


Fig. 2. Architecture of Ax-LIMS

We call these requirements informal software requirements, since they are written in plain English. Students are asked to turn this informal requirements document into a semi-formal document, that is, one that expresses informal functional requirements in terms of pre-conditions, post-conditions and invariants, suitable to later be ported into JML formal specifications. Modelling these invariants in JML guarantees that any class method implementing any functionality of the Ax-LIMS manager plug-in must adhere to Ax-LIMS' software requirements. This is checked with the JML common tools.

Additionally, students are asked to derive a class structure for the plug-in from the use cases and the informal functional requirements. A typical class structure might include classes `Project`, `Task`, and `State`. The `Project` class might declare a field `tasks` of type `List`[1] for managing project tasks. The `State` might define all possible states for projects and tasks. In classes `Task` and `Project`, fields `psd`, `pcd`, `asd` and `acd`, representing planned start date, planned completion date, actual start date and actual completion date are declared.

From this basic implementation, semi-formal requirements such as "The planned start date for a task is earlier than its planned completion date", and "If a task has an actual completion date, then it also has an actual start date" can be specified in JML as two invariants in class `Task` (see below). The method `compareTo` in the Java standard class `Date` returns a positive number whenever the first date is bigger than the second one, a negative number when the second one is bigger than the first one, otherwise returns `0`.

```
//@ invariant pcd.compareTo(psd) > 0;
//@ invariant acd != null ==> asd != null;
```

[1] Field `tasks` ought to be declared of type `ArrayList<Task>` indeed, but the Java version under which the latest version of JML runs does not allow parametric types.

Finally, students are taught to write and evolve Java code that adheres to the JML invariant properties obtained from the functional requirements. Students are shown how to iteratively specify invariants in JML, write Java code that respects these specifications, check the code against the JML specifications using the JML common tools, and evolve the specifications or the code (or both) accordingly. Once code that complies with all the class invariants is written, no error is issued by the JML common tools. Checking that written code complies with JML-specified invariants is automatically done by JML. It eliminates students' responsibility of keeping track of how properties a program must respect are affected by changes in the program code.

4.2 Experience with the Course

We want to emphasise here the importance of thinking of invariant properties when developing software. Thinking about invariants prior to writing code is a practice to which students do not easily adhere. Having a previous formal specification of the application and systematically using a tool, *i.e.*, the JML common tools, for checking the correctness of the code as it is written forces students to think about how the written code affects the consistency and the correctness of the whole program.

It is our experience that invariants are the key notion in formal software development that makes a difference with respect to traditional (non formal methods based) software engineering courses; whereas students attending traditional software engineering courses are not used to think of invariants properties leading their software developments. In general, students feel intimidated by the idea of coming up with an invariant. Often, they design code that can make their programs be in an inconsistent state. We endeavour to help students to surmount this fear so as to write their programs in a more mathematically correct way. We strongly believe JML already helps students in this sense, from furnishing a friendly Java-like syntax, to making it possible to use first-order logic predicates in JML specifications naturally.

5 Model-Driven Software Engineering in B

This course illustrates the advantages of using formal techniques for modelling systems and developing software. The course puts forward the idea of using these techniques as guiding principles and methodologies for thinking a system model from scratch in a disciplined way. The event B approach is developed in the first part of the course and the B method in the second (see Sections 2.4 and 2.3). Both are based on the analysis of a collection of case studies.

The course follows what Abrial calls "the parachute strategy", in which systems are first considered from a very abstract and simple point of view, with broad fundamental observations. This view usually comprises few simple invariant properties that students can easily grasp, *e.g.*, defining what can reasonably be expected from the operation of such a system. Liveness properties are also

introduced at this level. Once the system is completely understood at this level, students are required to develop small variants and to prove all obligations. The second step is to consider a refinement by adding a viewpoint observation in such a way that the new model keeps a palpable behavioural relation with its abstraction. Students are then encouraged to find by themselves precise relationships between abstract and concrete variables that guarantee coherent behaviour between both models.

As more experience is gained in the modelling framework, development of more complete systems comprising several refinements is attempted. In these refinements, the way how invariant properties can be used as a tool for tuning events is stressed. Systems that are very familiar to students, such as a soccer match, are used at this stage for them to be able to develop an intuitive, yet precise, idea of the use of refinement conditions. In the final stage of the course, a model of a more real-life system is constructed, alternating designs discussed in class with refinements supplied by students in their homework.

Although a broad review of first order logic and set theory is given at the beginning of the course, our strategy is to introduce event B modelling language constructs, such as relations, functions, sets and their operations as they are needed to express some specific property of the system at hand. Even the review of logic is done in a very "instrumental" way, stressing its use in modelling. Students are never required to do by-hand formal proofs of any logic formula, but they are constantly encouraged to argue about its validity. The idea behind this is to help students build a more "friendly" attitude towards mathematical formalisms, by showing them how a machine can do the tedious job of proving a formula in their place.

Our course uses the Rodin platform intensively (`http://wiki.event-b.org/index.php/Rodin_Platform`). This platform comprises an intuitive graphical interface based on Eclipse (`http://www.eclipse.org`) in which users can edit, prove and animate an event B model. In the beginning of the course, students are required to edit and type-check models in Rodin, though, instead of proving them, the animator plug-in is used to understand the model behaviour. In any case, initial models are chosen so that Rodin can automatically prove them. As familiarity with the tool increases, the models considered have a few proof obligations that Rodin cannot prove automatically. At this point, the "Mathematical Language" chapter of the Rodin manual [2] is presented. This provides a intuitive, yet rigorous, introduction to propositional and predicate logic from the vantage point of carrying out automatic proofs in those systems. Students can then understand Rodin proof strategies, and also relate this knowledge to how they can better guide Rodin's interactive prover. From this point on, students are required to fully prove their developments with Rodin.

A second part of the course uses event B to construct programs. This is done in event B by considering an abstract system that just specifies preconditions (context part and invariants) having an event computing in one step the result of the program (i.e. specifying its post-condition). Refinements of this initial model proceed just as for system models. The last refinement is such that each

event could easily be coded in some imperative programming language. Rules to automatically translate this last refinement into code can be easily devised but, unfortunately, as of this writing they have not been integrated into Rodin so that students must do this by hand.

The final part of the course deals with software systems. These, in principle, could be modelled in event B just as any other system. In our view, however, the lack of constructs to relate machines in different ways makes its use problematic in a pedagogical sense. The point is that students are, in other courses and in their initial job experiences, used to build software out of software pieces already available. In some versions of the course, we decided to use the B model explicit module integration constructs (e.g. using available machines within a system being developed). B machine constructs are introduced as needed for the model at hand. Specification of systems such as (a simplification of) the project management plug-in (see 4.1) are typical projects students consider for the *Software Processes* course.

5.1 Experience with the Course

As said before, the course is scheduled in the sixth semester, when students have already taken courses in programming, object oriented design and database models. They thus come with a background about how software is built that is very different from what they find in the event B course. This causes a sceptical attitude from the beginning that makes it difficult to keep alive the necessary motivation. We use two strategies to tackle this problem. First, systems considered at the beginning of the course pertain to situations that are familiar to students, yet not trivial. Second, in the software development part, students are required to constantly assess advantages and disadvantages of the same problem modelled with B and with the traditional methodologies used in other courses. Nevertheless, we have found that for this purpose a transition going from traditional ways of thinking systems to JML and then to event B greatly diminishes the initial scepticism and, at the same time, helps students to place each methodology in the right context of application.

A second issue is the use of mathematical formalisms. Students are evermore demanding to see clear relationships between their future professional activity and the mathematics they are being taught. In the first offerings of this course, mathematical formalisms were given a central role from the outset and thus all logic and mathematics needed to express event B proof obligations and constructs was presented at the beginning of the course. This only aggravated the motivational problem. Current versions of the course, as stated above, develops mathematical constructs along the course, always with a clear view of what precisely they are intended for in the problem at hand. Moreover, students are required to express in their own words those fundamental properties of a system, prior to their formalisation using the mathematical language of event B.

Using two slightly different models and their corresponding tools in the course poses problems. In principle, the whole course could be based on the B method, using the Atelier B (`http://www.atelierb.eu/index-en.php`) tool. However,

students have many problems with the interactive prover of Atelier B because of its black-box feel, with a command line approach that makes it hard to have a glimpse of exactly what hypotheses are being tried, what are available and therefore what strategies could better guide the proof. Work is underway by Atelier B maintainers to devise a more intuitive interactive prover, so this might change in the future.

5.2 Formal Software Development of MIO in B

The MIO mass transportation system was recently inaugurated in Cali. The system comprises a series of articulated buses following the main corridor routes of Cali, complemented with feeding buses connecting Cali with its outskirts. Besides serving as a palliative of the chaotic current transport service of Cali, the MIO system has renewed local people's sense of belonging to Cali. Our students thus feel excited about formally modelling the MIO and somehow contributing to the progress of Cali.

The MIO system is partially modelled in class by us; the rest is left to students as home-work. The MIO transport system is composed of articulated buses travelling along dedicated lanes of city avenues. Bus stations are constructions where users enter by validating a magnetic card in a reader. A turnstile unblocks when the card is validated so that the card owner can pass through. When a bus arrives, sliding doors in the station open in synchronisation with the bus doors, and passengers can enter or exit. Sensors at station doors identify when a bus door is opening. The station doors remain open during some fixed number of seconds. At any time, only one bus can be parked in a station. Users top-up their magnetic cards at machines in the stations.

There are two further motivations for introducing and modelling the MIO system in our course. Firstly, the system is already part of students daily lives so that they know very well how it works, and, secondly, a convincing complete

Table 3. Static part abstraction &refinement 1(left), refinement 2 (right)

constants	
std	door in station
axioms	
	$std \in DOOR \rightarrowtail\!\!\!\!\!\!\rightarrow ST$
variables	
$binst$	bus in station
opd	open station doors
$authb$	buses authorised to depart
invariant	
	$binst \in BUS \rightarrowtail ST$
	$opd \subseteq dom(std)$
	$std[opd] \subseteq ran(binst)$
	$authb \subseteq dom(binst)$
	$binst[authb] \cap std[opd] = \varnothing$

variables	
$authp$	persons authorised to enter
$perst$	persons inside a station
$perb$	persons in a bus
invariant	
	$perst \in PERSON \nrightarrow ST$
	$perb \in PERSON \nrightarrow BUS$
	$dom(perst) \cap dom(perb) = \varnothing$
	$authp \subseteq PERSON$
	$dom(perst) \subseteq authp$
	$dom(perb) \subseteq authp$

Table 4. Refinement 2: some events

depart =
any b
where
$b \in authb$
then
$binst := binst \setminus \{b \mapsto binst(b)\}$
$authb := authb \setminus \{b\}$
end

open_door =
any d
where
$d \in DOOR \wedge\ d \notin opd$
$std(d) \in ran(binst)$
$\wedge\ binst^{-1}(std(d)) \notin authb$
then
$opd := opd \cup \{d\}$
end

close_door =
any d
where
$d \in opd$
then
$authb := authb \cup \{binst^{-1}(std(d))\}$
$opd := opd \setminus \{d\}$
end

enter =
any p, s
where
$p \in authp \wedge\ p \notin dom(perst)$
$\wedge\ p \notin dom(perb) \wedge s \in ST$
$\wedge\ s \notin ran(perst)$
then
$perst := perst \cup \{p \mapsto s\}$
end

authorise =
any p
where
$p \in PERSON \wedge p \notin authp$
then
$authp := authp \cup \{p\}$
end

get_in_bus =
any p
where
$p \in dom(perst)$
$std^{-1}(perst(p)) \in opd$
then
$perbus := perbus \cup \{p \mapsto binst^{-1}(perst(p))\}$
$perst := perst \setminus \{p \mapsto perst(p)\}$
end

exit_from_bus =
any p
where
$p \in dom(perb)$
$perb(p) \in dom(binest)$
$std^{-1}(perst(p)) \in opd$
then
$perbus := perbus \setminus \{p \mapsto perb(p)\}$
$perst := perst \cup \{p \mapsto binest(perb(p))\}$
end

simplified model can be constructed that illustrates most of the features of the event B modelling technique. The whole model comprises an abstract machine, five contexts and seven refinement machines. A summary of all these is shown in table 5.

To give a broad idea of the development path of the system we discuss next the second refinement in table 5. The abstract model and the first refinement define the observations in the left of table 3. The second refinement includes those in

Table 5. Components of the MIO system

Component	Observations	Feature learnt
Abstraction	what bus is parked in what station	Modelling with partial functions
Refinement 1	station doors, set of parked buses allowed to leave	Linking refinement variables to abstract ones
Refinement 2	card holders in a station, card holders authorised to enter a station	Maintaining invariant on two linked functional variables, dealing with relational inverse
Refinement 3	Entrance control (abstract): card reader light, turnstile	Modelling equipment, tune events from invariants, model with relational range/domain restriction
Refinement 4	Equipment controller: card reader, controller, message channels	Interplay between physical and controller events, when/how to introduce software
refinement 5	passing through physical turnstile	Modelling message synchronisation
Refinement 6	Controller software: Modelling a message channel	Representing data structures as refinements
Refinement 7	sensors at station doors	Modelling weak and strong synchronisation patterns

the right. Refinement 2 is rather "orthogonal" with respect to refinement 1. The idea is to show students how new observation viewpoints lead to new events and, possibly, to stronger conditions in old events. The refinement models by set inclusion the fact that only authorised card holders can be in a station at a given time. It also gives the opportunity to discuss possible future enhancements to the current operation of the MIO system by keeping track, for example, of bus occupancy.

Students are encouraged to discuss what fundamental aspects of the operation of the system they experience each day should be present at this level and what could be deferred for further refinements. This discussion is always directed towards ways to express their contributions as part of the invariant, rather than thinking about possible new events. Property $dom(perst) \subseteq authp$, for instance, expressing the fact that only authorised card holders can be inside a station, was proposed by the majority of the students as pertaining to this level.

Some events of refinement 2 are shown in table 4. Events are rather simple. The message that is constantly given to students is that the construction of refinements should always be guided by the invariant. Students must always express the need of an event guard they propose in terms of an invariant property that must be maintained, such as, for example, the first guard of the "enter" event. Skill in using event B language constructs like the relational inverse is motivated by giving simple data structure-like intuitions for it. Students easily come up with, for example, the guard in the third line of the "open_door" event stating that, in order to open a station door, the bus currently parked at the station where the given door is, cannot have already been authorised to leave. These types of event guards, aiming at avoiding "ping-pong" loops between two

events are discovered by the students in a "critics game". Students are divided in groups of designers and critics. Credit is given to the former for coming up with correct events and to the latter for finding loopholes, particularly with respect to invariant properties. Later in the design such synchronisation patterns, in particular when material equipment is involved, are analysed in light of the "design patterns" proposed by Abrial in one of his event B examples.

6 Conclusion and Future Work

We believe formal methods tools have attained a level of maturity today that we should move into their use in practical settings, *e.g.*, making the word "formal" in "formal software development" the rule and not the exception. People in academia have a say on this. We can contribute to making this happen by helping students to build skills in formal software development, initiating students in the use of formal methods tools, and guiding them in the process of discovering the close embracing relation between software models and mathematical formalisms. That is, we can contribute to this by helping students to build skills to be put in practice in the "unconquered territory". This is an ambitious, yet feasible purpose.

Pontificia Universidad Javeriana has committed to the purpose of making formal methods and formal methods tools more popular in software development and in software industry. Firstly, Pontificia Universidad Javeriana computer science department keeps a close relation with ParqueSoft, the biggest Colombian technological centre. This relation encompasses the running and submission of R&D projects to Colciencias (the Colombian national science foundation), and projects are under their way to be submitted to international funding. Pontificia Universidad Javeriana further organises a student entrepreneurship program with ParqueSoft. Undergraduate students in the seventh semester can join a software venture formation program under the leadership of ParqueSoft. There they get hands-on experience in software development and marketing by integrating software development groups in young companies. In the last semester they propose software development ideas backed by more realistic knowledge of market needs. ParqueSoft can then choose to launch start-ups based on some of the most innovative ideas presented.

Secondly, the teaching of formal methods is key part of the software engineering undergraduate courses in the computer science department. As an example of formal methods courses taught in the science department, we have presented in this paper a JML-based formal development course, and a model driven course in B. The JML course allows students to have a first contact with formal specification of programs. Program correctness is introduced gradually. In our course, students enjoy evolving JML specifications and Java code, and checking the code for flaws. Our explanation for this positive attitude is that students regard the whole process of specification and verification as a different sort of programming. That is, that given a specification, a correct program must be constructed that respects the specification. The essence of "assigning programs to meanings".

It has been our experience in our formal methods courses that invariant is a key central notion in formal development courses, either in the form of class invariants as in JML, or as a refinement gluing invariant in the B method for software development, that leads software developments. This is students' first face-off with invariants. We acknowledge students often win this battle.

In these experiences, we have found important to develop in students a standpoint of complementarity with regard to methodologies and techniques in software modelling and construction. We are careful not to present formal methods as "better" methodologies that should replace other strategies in all situations. On the contrary, we encourage students to think on ways the knowledge gained in formal method courses may help improving their approach in traditional software engineering methods.

The authors are currently lecturing together a "formal modelling of systems using discrete mathematics" formal methods course for a recently opened master in engineering offered by Pontificia Universidad Javeriana computer science department. This year is our first year of lecturing this course. The course is organised half about the JML part, and half part on software development in B. It has been our intention not only to provide students with two complementary approaches to software construction, but also to communicate the idea that the part in B is a continuation of the part in JML. That is, we want to find common ways to relate the teaching of formal methods in JML and formal methods in B.

In [5] F. Bouquet *et al.* present the JML2B tool that checks JML specifications for inconsistencies by porting the specifications into B machines. All the available B tools can then be used for checking the B code. The translation from JML to B machine goes straightforward. JML invariants are expressed as B invariants universally quantified over all the instances of the class. Method preconditions translation includes conditions on the types of the method parameters. JML `assignable` classes, restricting which variable may be changed by a method call, are translated into the "any" form of events (see Section 2.4 for an introduction to events). We envision defining a formal methods teaching methodology that considers going from informal software requirements to JML specifications and Java code, straight down to B machines in near future.

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Abrial, J.-R.: Rodin deliverable D7, Event-B mathematical Language. In: Information Society Technologies, ch. V (2005)
3. Abrial, J.-R., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: Application to Event-B. Fundam. Inf. 77(1-2), 1–28 (2007)
4. Abrial, J.R., Hallerstede, S.: Refinement, decomposition and instantiation of discrete models: Application to Event-B. Fundamentae Informatica 77(1,2), 1–24 (2007)
5. Bouquet, F., Dadeau, F., Julien, J.: JML2B: Checking JML specifications with B machines. In: The 7th International B Conference, pp. 285–288 (2007)

6. Breunesse, C., Catano, N., Huisman, M., Jacobs, B.: Formal methods for Smart Cards: An experience report. Science of Computer Programming 55(1–3), 53–80 (2005)
7. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer (STTT) 7(3), 212–232 (2005)
8. Catano, N., Barraza, F., García, D., Ortega, P., Rueda, C.: A case study in JML-assisted software development. In: Proceedings of the Eleventh Brazilian Symposium on Formal Methods (SBMF 2008). ENTCS, July 2009, vol. 240, pp. 5–21 (2009)
9. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P.: JML reference manual (2008), `http://www.eecs.ucf.edu/~leavens/-JML/jmlrefman/jmlrefman_toc.html`
10. Link, J.: Unit Testing in Java. Morgan Kaufmann, San Francisco (2003)
11. Meyer, B.: Applying "design by contract". Computer 25(10), 40–51 (1992)
12. Meyer, B.: Object Oriented Software Construction. Prentice Hall PTR, Englewood Cliffs (1997)
13. Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS Language Reference. Computer Science Laboratory, SRI (November 2006)
14. Robinson, A., Voronkov, A.: Handbook of Automated Reasoning. MIT Press, Cambridge (2001)
15. Schneider, S.: The B-Method: An Introduction. Palgrave, Oxford (2001)
16. Spivey, J.M.: An introduction to Z and formal specifications. Software Engineering Journal 4(1), 40–50 (1989)
17. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. International Series in Computer Science. Prentice-Hall, Inc., Englewood Cliffs (1996)

Teaching Formal Methods Based on Rewriting Logic and Maude

Peter Csaba Ölveczky

Department of Informatics, University of Oslo

Abstract. This paper advocates teaching formal methods based on rewriting logic and the Maude tool for the purpose of widening access to formal methods. On the one hand, Maude's elegant, intuitive, and expressive programming/modeling language, its high-performance analysis methods, and some of its academic and industrial applications should make it appealing to a wide range of computer science students. On the other hand, teaching rewriting logic allows us to naturally incorporate substantial formal methods theory, such as equational logic and inductive theorem proving, TRS theory, and model checking. This paper also gives an overview of the content of – and the student feedback to – an introductory formal methods course based on rewriting logic that has been given at the University of Oslo since 2002.

1 Introduction

The slogan of TFM'09 is *widening the access to formal methods*. Key challenges that must be addressed by formal methods courses aiming at introducing formal methods to students with limited awareness and/or interest in formal methods include:

1. The need for formal methods must be well motivated to possibly skeptical students.
2. Applying formal methods should be reasonably easy, fun, and elegant.
3. The selected formalisms must appear to be relevant, both w.r.t. the student's future specialization – where we hope (s)he will apply formal methods – and elsewhere outside academia. No matter how nice a formalism and a tool are, if they are not used outside academia, the students will not be motivated to use the tool.
4. At the same time, the course should provide a serious introduction to substantial aspects of formal methods theory, and should exhibit some of the success stories of formal methods.

This paper advocates the use of *rewriting logic* [1, 2] and its associated high-performance tool Maude [3] as a basis for introductory courses in formal methods that interest students while giving a good introduction to formal methods.

Rewriting logic extends equational logic and term rewriting to model dynamic systems. In rewriting logic, the static parts of the system are modeled as

J. Gibbons and J.N. Oliveira (Eds.): TFM 2009, LNCS 5846, pp. 20–38, 2009.
© Springer-Verlag Berlin Heidelberg 2009

an algebraic equational specification, and dynamic state changes are modeled as rewrite rules on the equivalence classes of terms induced by the equational theory. There is by now ample evidence that, despite the simplicity of the formalism, rewriting logic is quite expressive and general, and can be used to model a wide range of distributed systems. In particular, it provides a nice and simple model for concurrent objects [4]. Maude [3] is a freely available high-performance state-of-the-art formal tool based on rewriting logic. Maude supports the simulation/rewriting, reachability analysis, and linear temporal logic model checking of rewrite theories.

The challenges (1) to (4) above are (or can be) addressed by a rewriting-logic-based course as follows:

1. There is a wide range of critical distributed systems where formal analysis has proved indispensable. The model checking effort of Lowe to find an attack on the NSPK cryptographic protocol after 17 years is but one example that should appeal to students.
2. The functional programming style of Maude is fairly elegant – as I try to convey in Section 4 – and is typically enjoyed by students who like programming. Likewise, the object-oriented and rule-based way of modeling distributed systems has been shown to be intuitive and easily understandable also for people without formal methods background [5].
3. Since rewriting logic is fairly expressive and can be applied to a wide range of distributed systems, it should be relevant to students who will pursue other fields than formal methods. Furthermore, Maude is increasingly being used outside academia, with some "sexy" applications, such as its use at Microsoft to find previously unknown security flaws in web browsers [6], the use in the Japanese car industry to find bugs in embedded automotive software, etc.
4. At the same time, a rewriting-logic-based course naturally includes a fair amount of formal theory, including formal proofs and deduction systems, classic term rewrite system theory, algebra, and (linear temporal logic) model checking, often mentioned as one of the success stories of formal methods.

To test the hypothesis about the suitability of widening access to formal methods through rewriting logic and Maude, I have developed an introductory course at the Department of Informatics, University of Oslo, that has been given since 2002. University of Oslo should be a suitable candidate for this experiment, since formal methods typically do not attract many students, and since many students at the department are somewhat weak and uninterested in mathematics. This paper gives an overview of the contents of our course, and on our experiences and student feedback on this course since 2002.

The course consists of two parts: (i) equational specifications and their analysis, and (ii) rewrite specifications and their analysis. The first part introduces classic (order-sorted) equational logic and term rewrite system (TRS) theory. In particular, we study: the definition of the usual data types (lists, sets, binary trees, ...) in Maude; classic TRS theory including proving termination of and confluence of an equational specification; equational logic; inductive theorems; and some small examples such as quick-sort and merge-sort. Part (ii) introduces:

rewriting logic and its proof theory; modeling distributed concurrent objects in rewriting logic; modeling a wide range of communication forms; temporal logic; reachability analysis and LTL model checking in Maude; and a set of larger examples, such as the two-phase commit protocol for distributed databases, the TCP, alternating bit, and sliding windows communication protocols, and the NSPK cryptographic protocol.

By giving a range of larger examples, I have tried to convey the difficulty of designing distributed systems, and how such systems can be modeled. Most importantly, the NSPK crypto-protocol case study serves as a very nice motivating example for formal model checking in today's iBanking society: a three-line protocol which is so hard to understand and get right that its flaws went undiscovered for 17 years, and whose critical flaw was discovered by formal model checking techniques similar to those presented in the course.

As further explained in Section 6, our experiences are mostly positive. The course consistently gets very positive student evaluation, but has one substantial weakness. Furthermore, although the course does not attract as many students as we would have liked, the course does attract significantly more students than our previous formal methods course, and is mostly taken by students who major in other fields, such as computer networks, computational linguistics, mathematical logic, and so on.

Section 2 gives some background on rewriting logic and Maude. Section 4 gives an overview of our course, and gives some samples of the Maude specifications to allow the reader to form a first impression about the suitability of basing a formal methods course on Maude. The paper also discusses experiences and student feedback on the course (Section 6), course material (Section 5), and related courses (Section 7). Section 9 gives some concluding remarks.

2 Rewriting Logic and Maude

Rewriting logic [1,2] is a logic of change that was developed by José Meseguer in the early 1990-ies. A rewriting logic specification is a rewrite theory (Σ, E, R), where (Σ, E) is an algebraic equational specification – that may be unsorted, many-sorted, order-sorted, or a membership equational logic [7] specification – with Σ an algebraic signature and E a set of equations (and possibly membership axioms), and where R is a set of labeled conditional rewrite rules of the form

$$l : \ [t]_E \longrightarrow [t']_E \ \textbf{if} \ cond,$$

with l a *label* and t and t' Σ-terms. Such rules specify the system's local transition patterns. The state space and functions of a system are thus specified by an equational specification, whereas the dynamic state changes are modeled by rewrite rules. Despite its simplicity, rewriting logic has been shown to be an expressive model of concurrency in which many other models of concurrency and communication can be naturally represented [1,8].

Maude [3] is a mature high-performance language and tool supporting the specification and analysis of rewrite theories. Maude assumes that the equations

are terminating and confluent. Maude executes rewrite rules by reducing the state to its equational normal form before applying a rewrite rule. We briefly summarize the syntax of Maude. Operators are introduced with the `op` keyword. They can have user-definable syntax, with underbars '`_`' marking the argument positions, and are declared with the sorts of their arguments and the sort of their result. Some operators can have equational *attributes*, such as `assoc`, `comm`, and `id`, stating, for example, that the operator is associative and commutative and has a certain identity element. Such attributes are then used by the Maude engine to match terms *modulo* the declared axioms. There are three kinds of logical statements, namely, *equations*—introduced with the keywords `eq`, or, for conditional equations, `ceq`—*memberships*—declaring that a term has a certain sort and introduced with the keywords `mb` and `cmb`—and *rewrite rules*—introduced with the keywords `rl` and `crl`. The mathematical variables in such statements are declared with the keywords `var` and `vars`.

Full Maude [3] is a prototype extension of Maude – implemented in Maude – that provides convenient syntax for object-oriented specification. In object-oriented Full Maude modules one can declare *classes* and *subclasses*. A class declaration

`class` C `|` att_1 `:` s_1 `, ... ,` att_n `:` s_n `.`

declares an object class C with attributes att_1 to att_n of sorts s_1 to s_n. An *object* of class C in a given state is represented as a term

`<` O `:` C `|` att_1 `:` val_1 `, ...,` att_n `:` val_n `>`

where O is the object's name, and where val_1 to val_n are the current values of the attributes att_1 to att_n. In an object-oriented system, the state, which is usually called a *configuration*, is a term of the built-in sort `Configuration`. It has typically the structure of a *multiset* made up of objects and messages. Multiset union for configurations is denoted by a juxtaposition operator (empty syntax) that is declared associative and commutative and having the `none` multiset as its identity element, so that order and parentheses do not matter, and so that rewriting is *multiset rewriting* supported directly in Maude. The dynamic behavior of concurrent object systems is axiomatized by specifying each of its concurrent transition patterns by a rewrite rule. For example, the rule

```
rl [l] : m(O,w)  < O : C | a1 : x, a2 : O', a3 : z >   =>
                 < O : C | a1 : x + w, a2 : O', a3 : z >  m'(O') .
```

defines a family of transitions in which a message `m`, with parameters `O` and `w`, is read and consumed by an object `O` of class `C`. The transitions have the effect of altering the attribute `a1` of the object `O` and of sending a new message. "Irrelevant" attributes (such as `a3`, and the *right-hand side occurrence* of `a2`) need not be mentioned in a rule.

As mentioned, rewrite theories are executable under fairly mild conditions. Maude supports a wide range of analysis strategies for rewrite theories, including simulation, reachability analysis, and linear temporal logic model checking.

Maude's *rewrite* command simulates *one* behavior of the system, possibly up to a certain number of rewrite steps. It is written with syntax

`rew` [[*n*]] *t* `.`

where *t* is the initial state, and *n* is the (optional) bound of the number of rewrite steps to execute.

Maude's *search* command uses explicit-state breadth-first search to search for states that are reachable from a given initial state *t*, and that match a *search pattern*, and satisfy a *search condition*. The command which searches for *one* state satisfying the search criteria has syntax

`search [1]` *t* `=>*` *pattern* `such that` *cond* `.`

Maude caches the visited states, so that the search command terminates if the state space reachable from *t* is finite, or if the desired number of (un)desired states are reachable from the initial state.

Maude's *linear temporal logic model checker* [9] analyzes whether each behavior from an initial state satisfies a temporal logic formula. *State propositions* are terms of sort `Prop`, and their semantics should be given by (possibly conditional) equations of the form

statePattern `|=` *prop* `=` *b*

for *b* a term of sort `Bool`, which defines the state proposition *prop* to hold in all states *t* where *t* `|=` *prop* evaluates to `true`. A temporal logic *formula* is constructed by state propositions and temporal logic operators such as `True`, `False`, `~` (negation), `/\`, `\/`, `->` (implication), `[]` ("always"), `<>` ("eventually"), `O` ("next"), and `U` ("until"). The model checking command has syntax

`red modelCheck(`*t*`,` *formula*`)` `.`

for *t* the initial state and *formula* the temporal logic formula. The model checking terminates if the state space reachable from *t* is finite.

Rewriting logic is an active area of research, in which more than a thousand research papers have been published. Some of the applications of rewriting logic and Maude include: defining the formal semantics for a wide range of programming and modeling languages [10]; work at Microsoft that discovered previously unknown address and status bar spoofing attacks in web browsers [6]; developing analysis tools for programming languages, such as the JavaFAN [11] tool for efficiently analyzing multi-threaded Java programs; analysis of advanced security, communication, and wireless sensor network protocols (see e.g. [12, 13, 5, 14, 15, 16]); modeling of cell biology to simulate and analyze biological reactions [17, 18]; finding several bugs in embedded software used by major car makers; implementing the latest version of the NRL PA crypto-protocol analysis tool [19]; implementing extensions of rewriting logic, such as Real-Time Maude tool [20, 21] for real-time systems and the PMaude tool for probabilistic systems [22]. The paper [23] provides an early bibliography and roadmap of the use of Maude around the world.

3 Prerequisites and Course Duration

Our course is aimed at third-year students at the Department of Informatics at the University of Oslo. The course basically starts "from scratch", as it is not assumed that students have significant mathematical background; they may for example never have seen a proof system before. It is also *not* assumed that the students have any experience with functional or logic programming.

The course consists of 14 lectures, each lasting between two and three 45-minute "hours".

4 Course Overview and Sampler

This section gives an overview of the course content, and some samples to get a first impression of the course. The course is divided into two roughly equal-sized parts: modeling and analyzing, respectively, the *static* and the *dynamic* parts of the systems. The static part corresponds to "classical" term rewrite system (TRS) theory and specification of data types in Maude.

4.1 The Static Part

Defining Data Types in Maude. In Maude, data types are defined by an algebraic *equational specification*, where a *signature* declares a set of sorts and function symbols (or *operators*); the operators are divided into *constructors* (`ctor`) that define the carrier of the sort, and *defined functions*. The first such specification given in the course is the following module, that defines the natural numbers, together with an addition function, in a Peano style:

```
fmod NAT-ADD is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .

  vars M N : Nat .

  eq 0 + M = M .
  eq s(M) + N = s(M + N) .
endfm
```

The underscore in the function declaration gives the place of the arguments in a "mix-fix" notation. It should be noted that, for convenience and efficiency, common data types, such as integers, floating-point numbers, Booleans, strings, etc., are built-in in Maude.

As mentioned in Section 2, the equational specification may be order-sorted (that is, include subsort declarations), and operators can be declared to be *associative* (`assoc`), *commutative* (`comm`), and/or to have an *identity element* (`id`). Matching is then performed *modulo* such axioms. Using these features, lists of integers can be defined as follows, where the sort `NeList` defines *non-empty* lists:

```
sorts List NeList .
subsorts Int < NeList < List .

op nil : -> List [ctor] .
op __ : List List -> List [assoc id: nil ctor] .
op __ : NeList NeList -> NeList [assoc id: nil ctor] .
```

List concatenation is defined by the juxtaposition operator (`__`). Since `Int` is a subsort of `List`, the number `6` is also a list. Therefore, `6 32` is a two-element list, and `(6 32) 17` is a list. Since `__` is declared to be associative, `(6 32) 17` and `6 (32 17)` are the same list, and can be written `6 32 17`. Since list concatenation is declared to have `nil` as the identity element, any list is either of the form `nil` or $n\ l$ (or $l\ n$), for n an integer and l a `List`.

The definition of the usual list functions seems to be more elegant than in most languages:

```
op length : List -> Nat .
ops first last : NeList -> Int .
op rest : NeList -> List .
op reverse : List -> List .

vars I J K : Int .    vars L L' : List .    vars NEL NEL' : NeList .

eq length(nil) = 0 .                    eq reverse(nil) = nil .
eq length(I L) = 1 + length(L) .        eq reverse(L I) = I reverse(L).

eq first(I L) = I .                     eq rest(I L) = L .

eq last(L I) = I .
```

Using this representation of lists, the well known merge-sort function can be specified fairly elegantly in Maude:

```
op mergeSort : List -> List .
op merge : List List -> List [comm] .

eq mergeSort(nil) = nil .
eq mergeSort(I) = I .
ceq mergeSort(NEL NEL') = merge(mergeSort(NEL), mergeSort(NEL'))
      if length(NEL) == length(NEL')  or length(NEL) == s length(NEL') .

eq merge(nil, L) = L .
ceq merge(I L, J L') = I merge(L, J L') if I <= J .
```

In the same way, we have defined the usual data types, such as binary trees, and multisets. Indeed, any computable data type can be defined as a confluent and terminating equational theory [24].

Confluence and Termination. Maude's rewrite engine executes equational specifications by reducing a term to its normal form. Maude therefore assumes

that a specification is *confluent* and *terminating*, modulo associativity and commutativity of the function symbols so declared. This gives us the motivation to study these properties, which in my introductory course are studied only for unsorted systems without associativity, commutativity, and conditional equations. After defining formally what it means to perform a simplification step, confluence is studied in the usual way, assuming termination, and using the critical pair's lemma to check local confluence.

The course deals a fair amount with theoretical and practical aspects of proving termination. We study both "weight function" mappings of ground terms onto well-founded domains, and the elegant theory of Dershowitz' simplification orderings [25, 26] that lead to the multiset and lexicographic path orderings as well as other termination orderings. Although these techniques apply to term rewrite systems, it is the hope and motivation that the students are able to adapt their understanding of termination to also analyze termination of imperative programs such as the Euclidean algorithm for finding the greatest common divisor of two natural numbers `m` and `n`:

```
int gcd(int m, int n) {   // m,n > 0
  int x := m;   int y := n;
  int r := x % y;
  while (r>0) {
    x := y;
    y := r;
    r := x % y;
  }
  return y;
}
```

Equational Logic. The course introduces equational logic (again, in its simplest, unsorted case). For many of our students, this is the first time they see a formal deduction system. The usual undecidability and decidability results are given. Finally, we study *inductive theorems* and induction on data types, and prove basic inductive theorems on toy problems, such as that reversing a binary tree twice yields the original tree.

Although the described course does not use it, Maude has an associated *inductive theorem prover* that can assist in the proof of inductive theorems [27].

4.2 Modeling and Analyzing Dynamic Systems in Maude

Part II of the course deals with the formal modeling of *dynamic* systems as *rewriting logic* theories, and their formal analysis in the Maude tool. Since two of the main goals of the course is to (i) give the students some intuition about the difficulty of designing distributed systems, and (ii) teach the students to formally model designs of such systems, we focus on modeling and analyzing a range of examples from different domains: transport protocols, transaction protocols for distributed database systems, and cryptographic protocols.

Rewriting Logic. We introduce *rewriting logic* and its proof theory; in particular, this proof system allows us to reason about what actions can be performed concurrently. We show that if the state has a *multiset* structure, then each element in the multiset could be involved in one rewrite "action" in a concurrent rewrite step; that is, distributed objects can performs actions concurrently.

We illustrate such modeling with simple examples, such as modeling the life (the age and marital status) of a collection of persons, various kinds of games, etc. For example, (the scores of) a never-ending soccer game is modeled by the following module, where a typical state could be the term `"PSV" - "Ajax" 3 : 2`:

```
mod GAME is protecting NAT + STRING .
  sort Game .
  op _-_ _:_ : String String Nat Nat -> Game [ctor] .

  vars HOME AWAY : String .     vars M N : Nat .

  rl [home-goal] :
     HOME - AWAY M : N  =>  HOME - AWAY M + 1 : N .

  rl [away-goal] :
     HOME - AWAY M : N  =>  HOME - AWAY M : N + 1 .
endm
```

Formal Analysis in Maude. Since, in contrast to equational theories, the rewrite rules need not be confluent or terminating, the Maude tool offers different formal analyses. We cover rewriting for simulation and search for reachability analysis. For example, in the soccer game, the following search command checks whether it is possible to reach a state in which the away team leads by at least three goals[1]:

```
Maude> search [1] "Ajax" - "PSV" 0 : 0
                =>*
                "Ajax" - "PSV" M:Nat : N:Nat such that N:Nat >= M:Nat + 3 .
```

Concurrent Objects. We then introduce concurrent objects by the simple example of modeling the lives of a collection of persons, where the class `Person` is declared as follows:

```
  class Person | age : Nat, status : Status .

  sort Status .
  op single : -> Status [ctor] .
  ops engaged married separated : Oid -> Status [ctor] .
```

[1] Variables declared on the fly have the form *var*:*sort*.

The following conditional rewrite rule, involving two objects, models the engagement of two single persons who are both older than 15:

```
vars N N' : Nat .   vars X X' : String .

crl [engagement] :  < X : Person | age : N, status : single >
                    < X' : Person | age : N', status : single >
                   =>
                    < X : Person | status : engaged(X') >
                    < X' : Person | status : engaged(X) >
                   if N > 15 /\ N' > 15 .
```

A married person can initiate a separation, for example by sending a message to his/her spouse. The following declares a `separate` message, and shows the rules for sending, respectively receiving, such a message[2]:

```
msg separate : Oid -> Msg .

rl [sendSep] :  < X : Person | status : married(X') >
               =>
                < X : Person | status : separated(X') >
                separate(X') .

rl [recvSep] :  separate(X)
                < X : Person | status : married(X') >
               =>
                < X : Person | status : separated(X') > .
```

The course also presents an object-oriented version of the mandatory *dining philosophers* example.

Communication Protocols. After defining ways of modeling a wide range of communication models (unordered/ordered transmission; reliable/unreliable; unicast/multicast/broadcast, ...), we specify a set of transport protocols, such as TCP-like sequence number based protocols, the alternating bit protocol, and (as a student homework) different versions of the sliding window protocol. Maude search is used to analyze the protocols. From an educational perspective, the sliding window protocol is a very good example for illustrating that model checking distributed systems takes a long time in a highly nondeterministic setting where any message may get lost (or duplicated).

The Two-Phase Commit Protocol for Distributed Databases. We model and analyze the *two-phase commit protocol* for transactions in distributed database systems with replicated data, as the protocol is described in the textbook [28]. Again, we analyze our model by searching for *final* states of protocol runs, and automatically find the well known facts that the multi-database is consistent after a run if and only if messages cannot get lost.

[2] As we show in the course, separation is not this simple.

The NSPK Cryptographic Protocol. The *Needham-Schroeder public key* (NSPK) authentication protocol is a frequently used and cited protocol from 1978. NSPK is, for example, cited in *Handbook of Applied Cryptography* [29] from 1996, without any error in the protocol being mentioned. In crypto-protocol notation, NSPK is described as follows:

$$\begin{array}{lll} \text{Message 1.} & A \rightarrow B : & A.B.\{N_a.A\}_{PK(B)} \\ \text{Message 2.} & B \rightarrow A : & B.A.\{N_a.N_b\}_{PK(A)} \\ \text{Message 3.} & A \rightarrow B : & A.B.\{N_b\}_{PK(B)} \end{array}$$

We define an object-oriented Full Maude model of the protocol for multiple agents and protocol runs. An agent which can *initiate* a run of the protocol is modeled as an object of the following class `Initiator`:

```
class Initiator | initSessions : InitSessions, nonceCtr : Nat .
```

The initiator needs to know the nonce it sent to the responder in Message 1, so that it can check whether this is the same nonce that it receives in Message 2. In our setting, where an initiator may be simultaneously involved in many runs of the protocol with different responders, the initiator must store information about the nonces of all these runs. In the attribute `initSessions` an initiator A stores such information in a multiset of elements of the following three kinds:

- `notInitiated`(B) indicates that A can/will initiate contact with B but has not yet done so;
- `initiated`(B, N) indicates that A has sent Message 1 to B with nonce N and is waiting for Message 2 from B; and
- `trustedConnection`(B) indicates that A has established (what she thinks is) an authenticated connection with B.

The data type representing this kind of information is defined as follows:

```
sorts Sessions InitSessions .          subsort Sessions < InitSessions .
op emptySession : -> Sessions [ctor] .
op __ : InitSessions InitSessions -> InitSessions
                                       [ctor assoc comm id: emptySession] .
op __ : Sessions Sessions -> Sessions [ctor assoc comm id: emptySession] .

op notInitiated : Oid -> InitSessions [ctor] .
op initiated : Oid Nonce -> InitSessions [ctor] .
op trustedConnection : Oid -> Sessions [ctor] .
```

The dynamics of the protocol is described by four rewrite rules, two for the initiator and two for the responder. The rules for the initiator are described next.

The first rule models the sending of Message 1. The agent `A` has `notInitiated(B)` in its `initSessions` attribute, which indicates that it is interested in establishing an authenticated connection with `B`. The initiator generates a fresh nonce `nonce(A, N)`, encrypts this nonce together with its identifier with

the public key of B (`encrypt ... with pubKey(B)`), and adds its own and B's name (`msg ... from A to B`) to this message and sends it out into the configuration. Agent A must remember that it has `initiated` contact with B with nonce `nonce(A, N)` and must also increase its nonce counter. All this happens in the following rule:

```
vars A B : Oid .    vars M N : Nat .    vars NONCE NONCE' : Nonce .
var IS : InitSessions .

 rl [start-send-1] :
    < A : Initiator | initSessions : notInitiated(B) IS, nonceCtr : N >
   =>
    < A : Initiator | initSessions : initiated(B, nonce(A, N)) IS,
                      nonceCtr : N + 1 >
    msg (encrypt (nonce(A, N) ; A) with pubKey(B)) from A to B .
```

The next rule models the reception of Message 2 from, and the sending of Message 3 to, an agent B who sent a pair of nonces encrypted with A's public key. If the first nonce (`NONCE`) in the message received (and decrypted) by A is the same as the nonce stored in A's `initSessions` attribute for B, the agent A figures out that it has established an authenticated connection with B, and sends Message 3 (B's nonce (`NONCE'`) encrypted with B's public key) to B:

```
rl [read-2-send-3] :
   (msg (encrypt (NONCE ; NONCE') with pubKey(A)) from B to A)
   < A : Initiator | initSessions : initiated(B, NONCE) IS >
  =>
   < A : Initiator | initSessions : trustedConnection(B) IS >
   msg (encrypt NONCE' with pubKey(B)) from A to B .
```

The Dolev-Yao intruder is modeled as a class `Intruder` with 14 simple rewrite rules. For example, the following rule models the case when the intruder intercepts a message that it cannot decrypt. In that case, the intruder just stores the message content in its `encrMsgsSeen` attribute, and stores the new names it learns in its `agentsSeen` attribute:

```
vars I O O' : Oid .   var OS : OidSet .   var MSGC : MsgContent .
var ENCRMSGS : EncryptedMsgContentSet .

crl [intercept-but-not-understand] :
    (msg (encrypt MSGC with pubKey(O)) from O' to O)
    < I : Intruder | agentsSeen : OS, encrMsgsSeen : ENCRMSGS >
   =>
    < I : Intruder | agentsSeen : OS ; O ; O',
                     encrMsgsSeen : (encrypt MSGC with pubKey(O)) ENCRMSGS >
   if O =/= I .
```

Another rule can then spontaneously send *any* fake message among the messages the intruder has seen out into the configuration, using any agent `A` it has seen[3] as sender:

```
crl [send-encrypted] :
    < I : Intruder | encrMsgsSeen : (encrypt MSGC with pubKey(B)) ENCRMSGS,
                     agentsSeen : A ; OS >
   =>
    < I : Intruder | >
    (msg (encrypt MSGC with pubKey(B)) from A to B)
   if A =/= B .
```

The following search command finds the well known attack on NSPK. In the initial state, `"Scrooge"` does *not* want to have a contact with the `"Bank"`. In the search command we check whether from such a state, it is possible to reach a state where the `"Bank"` thinks it has an authenticated connection with `"Scrooge"`:

```
Maude> (search [1]
         < "Scrooge" : Initiator |
                       initSessions : notInitiated("Beagle Boys"), ... >
         < "Bank" : Responder | respSessions : emptySession, nonceCtr : 1 >
         < "Beagle Boys" : Intruder |
                           initSessions : notInitiated("Bank"), ... >
       =>*
         C:Configuration
         < "Bank" : Responder | respSessions : trustedConnection("Scrooge")
                                               RS:RespSessions > .)
```

Maude does find a behavior leading to the bad state, and can exhibit this behavior as explained in [30].

This example is excellent for motivating the students, for illustrating the difficulties of designing distributed systems, and for showing the usefulness of formal model checking. The protocol is described in *three* lines of specification. Yet, due to concurrent runs, it is so hard to understand that it took 17 years to find the error, which was found using exactly the same kind of analysis we are doing: exhaustive model checking of a formal model of the protocol [31].

Temporal Properties and LTL Model Checking. Finally, we explain different classes of requirements, and show how invariants can be validated by Maude's search command. More complex temporal system properties can be formalized in linear temporal logic (LTL), and Maude's LTL model checker can be used to analyze whether a system satisfies its requirements. However, to avoid introducing yet another logic to my students, I typically postpone teaching temporal logic to the follow-up course.

[3] Both `encrMsgsSeen` and `agentsSeen` are declared to hold *multisets*. Therefore, the rule can nondeterministically select *any* of the agents and *any* of the messages the intruder has stored.

5 Teaching Material

Fairly mature lecture notes (340 pages) for my course are available on the web at `http://peterol.at.ifi.uio.no/inf3230-lecturenotes.html`. These lecture notes start from scratch and contain many examples and exercises, and should be accessible for people without any formal methods experience. These notes are also suggested reading for the introductory formal methods course CS 476 at the University of Illinois at Urbana-Champaign (UIUC), and seem to have been read by a fair amount of people at UIUC. These notes are also one of the main sources that are recommended to people who want to get a gentle introduction to Maude.

The Maude book [3] is a very useful reference material on the Maude system and on rewriting logic, but, in my view, requires some previous knowledge about term rewriting or similar formal methods. Although no course book exists for the course CS 476 at UIUC, the slides for that much more advanced introductory course are quite comprehensive and can almost be studied as a course book.

6 Evaluation and Impact

This section briefly summarizes anonymous student feedback, my own impression of the students' experiences, and some of the impact this course has had in Oslo.

6.1 Student Feedback

The comments and evaluations from university's anonymous web-based feedback system have been overwhelmingly positive. Unfortunately, most comments are non-constructive comments of the form "Very interesting and exciting". Some write that that it is good that "theory and practice are well interleaved," and other thought programming in Maude was fun. One person told me that he did not really understand the sliding window protocol in the network course, but understood it very well in this course, where implementation details are abstracted away to focus on the essence of the protocol.

The negative feedback overwhelmingly concerns Full Maude, the prototype extension of Maude that is used to model and analyze object-oriented Maude specifications. Unfortunately, Full Maude's slight lack of robustness and, in particular, its cryptic, non-existing, and/or misplaced error messages make it a frustrating experience for some students to get larger specifications, such as the sliding window protocol, right.

As for the difficulty of the course, 5% of my students in 2003 found the course "difficult," 55% found the course "somewhat difficult," and the remaining 40% found it "neither easy nor difficult."

Despite the very positive student feedback each year, the course still does not attract as many students as I would like. The reason *may* lie in the way the students have to select courses, and in the freedom they have to select courses outside their specialization. Typically, 20 to 30 students take the exam each year.

6.2 Impact in Oslo

I am not very much aware of what former students of the course are doing. Two cases that are worth mentioning are:

1. A former student and teaching assistant in my course started a company five years ago, selling a product/service that is implemented in Maude. The company is still doing well, and has also employed another former student and TA of mine to program in Maude.
2. Inspired by the use of Maude to find the attack on NSPK, a former student went on to do a Ph.D. in crypto-analysis using Maude, including defining his own protocol analyzer on top of Maude. The person is currently analyzing critical infrastructure in the Norwegian banking sector.

In addition, it is worth remarking that Maude is now frequently used in neighboring research groups at the department. For example, Maude is used to implement proof search strategies in the logics group, the Creol object-oriented language [32] developed at the department is interpreted in Maude, and a student in the linguistics research group (!) has analyzed an IETF-developed multicast protocol using Maude [14].

7 Related Courses

I am only aware of one other "introductory" formal methods course based on rewriting logic. It is given at UIUC by José Meseguer, who developed rewriting logic. That course is significantly more theoretically challenging and comprehensive than the one in Oslo. In addition to treating the theory of rewriting logic and its analysis methods in depth, it also focuses on the verification of sequential imperative programs. There is less focus on larger examples, although the NSPK case study is covered. Another difference is that the UIUC course does not have a course book (but an extensive set of slides). Indeed, my lecture notes are recommended supplementary reading in the UIUC course.

Classic term rewriting theory has been taught many places for years. One difference between many of those and the the first part of the Oslo course is our focus on defining functions in an executable language such as Maude.

Likewise, analyzing dynamic systems using process algebras, model checkers like Spin, various kinds of transition systems, has been much taught. One of the differences of using Maude is the *modeling convenience* and *expressiveness* of the Maude formalism, and the ability to perform both simulation, reachability analysis, and LTL model checking. Another attractive feature of Maude is its very simple and intuitive functional programming style language, which is typically appealing to students, and which makes it far easier to model a system system that, say, in Promela/Spin. Another difference is our focus on case studies from different domains.

8 Follow-Up Courses

Due to the large amount of interesting research being performed using rewriting logic and Maude, there are plenty of appealing subjects to choose from for an advanced follow-up course based on rewriting logic. The follow-up course at the University of Oslo teaches the following topics:

- A student project formalizes and analyzes a published communication protocol which is claimed to be correct, but where a simple Maude search finds an unexpected deadlock.
- Linear temporal logic and its model checking in Maude.
- Meta-programming in Maude.
- Specification and analysis of real-time systems using Real-Time Maude [20].
- Other analysis methods, including narrowing analyses and the use of Maude's inductive theorem prover (ITP) [27].
- Modeling cell biology and and analyzing biological cell reactions [17, 18].
- Study the work in [6] on finding the attacks in web browsers.

Other topics of general interest include:

- Grigore Roşu at UIUC teaches a course on how a wide range of programming languages can be given a rewriting logic semantics and can be analyzed using Maude.
- Probabilistic rewrite theories [33] and their analysis using PMaude [22].
- Theory on the algebraic denotational semantics of equational and rewrite theories.

This is but a small sample of topics that could be covered by an advanced course.

9 Concluding Remarks

This paper has advocated the use of rewriting logic and its associated high-quality tool Maude as a basis for teaching formal methods with the aim of *widening the access to formal methods*. The reasons for believing that a Maude-based formal methods course may interest people who would not normally consider studying formal methods include:

- The logic and programming language are simple and intuitive: they consist of an algebraic signature, equations, and rewrite rules. That's all. Furthermore, the object-oriented rules are very intuitive and easy to understand also for people without formal methods or Maude knowledge (see, e.g., [5]).
- The functional and fairly elegant programming possible in Maude, that this paper tried to convey with the merge-sort example, should be appealing to people who like to program.
- It is fairly easy – in particular compared to traditional formal languages for concurrency – to model a wide range of distributed systems in Maude.

- The NSPK security protocol analysis provides compelling motivation for the use of formal model checking. Furthermore, it is easy to specify NSPK in Maude, and to find the attack using search in Maude.
- No matter its elegance or nice features, no language will motivate students if it is perceived to be a purely academic language not used in industry. Maude has some "sexy" industrial applications, most notably the work at Microsoft that uncovered previous unknown security flaws in web browsers.

From a formal methods teaching perspective, a fair amount of formal methods theory, including classic TRS theory, proof systems and inductive proofs, as well as different forms of model checking – one of the success stories of formal methods – can naturally be integrated and motivated by the use of Maude for system modeling and analysis.

This paper has also presented an introductory Maude-based formal methods course given at the University of Oslo since 2002. This course, also aimed at – and taken by – students who will not necessarily pursue formal methods further, has consistently received positive student feedback, and comes with a fairly mature course book that is freely available and should be suitable starting point for studying formal methods.

Although I believe that the expressive and intuitive formalism of Maude makes it better suited than other model checking systems, such as SMV [34] and Spin [35], for teaching modeling and analysis of complex distributed systems with advanced data types and communication features, much more work comparing Maude-based formal methods teaching with other approaches is needed before significant conclusions can be drawn.

Acknowledgments. I am grateful to Olaf Owe for supporting the development of the described course at the University of Oslo, and to the anonymous reviewers for helpful comments on an earlier version of this paper.

References

1. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96, 73–155 (1992)
2. Bruni, R., Meseguer, J.: Semantic foundations for generalized rewrite theories. Theoretical Computer Science 360, 386–414 (2006)
3. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
4. Meseguer, J.: A logical theory of concurrent objects and its realization in the Maude language. In: Agha, G., Wegner, P., Yonezawa, A. (eds.) Research Directions in Concurrent Object-Oriented Programming, pp. 314–390. MIT Press, Cambridge (1993)
5. Ölveczky, P.C., Meseguer, J., Talcott, C.L.: Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. Formal Methods in System Design 29, 253–293 (2006)

6. Chen, S., Meseguer, J., Sasse, R., Wang, H.J., Wang, Y.M.: A systematic approach to uncover security flaws in GUI logic. In: IEEE Symposium on Security and Privacy, pp. 71–85. IEEE Computer Society, Los Alamitos (2007)
7. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Parisi-Presicce, F. (ed.) WADT 1997. LNCS, vol. 1376, pp. 18–61. Springer, Heidelberg (1998)
8. Meseguer, J.: Rewriting logic as a semantic framework for concurrency: a progress report. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 331–372. Springer, Heidelberg (1996)
9. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude Manual, Version 2.3 (2007), `http://maude.cs.uiuc.edu`
10. Meseguer, J., Rosu, G.: The rewriting logic semantics project. Theoretical Computer Science 373, 213–237 (2007)
11. Farzan, A., Chen, F., Meseguer, J., Rosu, G.: Formal analysis of Java programs in JavaFAN. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 501–505. Springer, Heidelberg (2004)
12. Denker, G., Meseguer, J., Talcott, C.: Protocol Specification and Analysis in Maude. In: Heintze, N., Wing, J. (eds.) Workshop on Formal Methods and Security Protocols, Indianapolis, Indiana, June 25 (1998)
13. Denker, G., García-Luna-Aceves, J.J., Meseguer, J., Ölveczky, P.C., Raju, Y., Smith, B., Talcott, C.: Specification and analysis of a reliable broadcasting protocol in Maude. In: Hajek, B., Sreenivas, R.S. (eds.) 37th Annual Allerton Conference on Communication, Control, and Computation. University of Illinois, Urbana-Champaign (1999)
14. Lien, E.: Formal modelling and analysis of the NORM multicast protocol using Real-Time Maude. Master's thesis, Department of Linguistics, University of Oslo (2004)
15. Goodloe, A., Gunter, C.A., Stehr, M.O.: Formal prototyping in early stages of protocol design. In: WITS 2005. ACM Press, New York (2005)
16. Ölveczky, P.C., Thorvaldsen, S.: Formal modeling, performance estimation, and model checking of wireless sensor network algorithms in Real-Time Maude. Theoretical Computer Science 410, 254–280 (2009)
17. Eker, S., Knapp, M., Laderoute, K., Lincoln, P., Talcott, C.: Pathway logic: Executable models of biological networks. Electronic Notes in Theoretical Computer Science 71 (2002)
18. Eker, S., Knapp, M., Laderoute, K., Lincoln, P., Meseguer, J., Sonmez, K.: Pathway logic: Symbolic analysis of biological signaling. In: Pacific Symposium on Biocomputing, Hawaii, pp. 400–412 (2002)
19. Escobar, S., Meadows, C., Meseguer, J.: State space reduction in the Maude-NRL Protocol Analyzer. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 548–562. Springer, Heidelberg (2008)
20. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation 20, 161–196 (2007)
21. Ölveczky, P.C., Meseguer, J.: The Real-Time Maude tool. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 332–336. Springer, Heidelberg (2008)
22. Agha, G., Meseguer, J., Sen, K.: PMaude: Rewrite-based specification language for probabilistic object systems. Electronic Notes in Theoretical Computer Science 153, 213–239 (2006)
23. Martí-Oliet, N., Meseguer, J.: Rewriting logic: Roadmap and bibliography. Theoretical Computer Science 285 (2002)

24. Bergstra, J.A., Tucker, J.V.: Initial and final algebra semantics for data type specificartions: Two characterization theorems. SIAM Journal on Computing 12, 366–387 (1983)
25. Dershowitz, N.: Orderings for term-rewriting systems. Theoretical Computer Science 17, 279–301 (1982)
26. Dershowitz, N.: Termination of rewriting. Journal of Symbolic Computation 3, 69–116 (1987)
27. Clavel, M.: The ITP tool home page, `http://maude.sip.ucm.es/itp/`
28. Elmasri, R., Navathe, S.B.: Fundamentals of Database Systems, 5th edn. Addison Wesley, Reading (2007)
29. Menezes, A., van Oorschot, P., Vanstone, S.: Handbook of Applied Cryptography. CRC Press, Boca Raton (1996), `http://www.cacr.math.uwaterloo.ca/hac`
30. Ölveczky, P.C.: Formal modeling and analysis of distributed systems in Maude. Course book for INF3230, Dept. of Informatics, University of Oslo (2009)
31. Lowe, G.: An attack on the Needham-Schroeder public-key authentication protocol. Information Processing Letters 56, 131–133 (1995)
32. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Software and Systems Modeling 6, 35–58 (2007)
33. Kumar, N., Sen, K., Meseguer, J., Agha, G.: A rewriting based model of probabilistic distributed object systems. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 32–46. Springer, Heidelberg (2003)
34. Clarke, E., Grumberg, O., Long, D.: Verification tools for finite-state concurrent systems. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) REX 1993. LNCS, vol. 803. Springer, Heidelberg (1994)
35. Holzmann, G.J.: The model checker SPIN. IEEE Trans. on Software Engineering 23, 279–295 (1997)

Which Mathematics for the Information Society?

João F. Ferreira[1], Alexandra Mendes[1], Roland Backhouse[1], and Luís S. Barbosa[2]

[1] School of Computer Science, University of Nottingham, Nottingham, England
{joao@joaoff.com, afm@cs.nott.ac.uk, rcb@cs.nott.ac.uk}
[2] CCTC & Dep. Informatics, Minho University, Braga, Portugal
lsb@di.uminho.pt

Abstract. MathIS is a new project that aims to reinvigorate secondary-school mathematics by exploiting insights of the dynamics of algorithmic problem solving. This paper describes the main ideas that underpin the project. In summary, we propose a central role for formal logic, the development of a calculational style of reasoning, the emphasis on the algorithmic nature of mathematics, and the promotion of self-discovery by the students. These ideas are discussed and the case is made, through a number of examples that show the teaching style that we want to introduce, for their relevance in shaping mathematics training for the years to come. In our opinion, the education of software engineers that work effectively with formal methods and mathematical abstractions should start before university and would benefit from the ideas discussed here.

We are all shaped by the tools we use,
in particular the formalisms we use shape our thinking habits,
for better or for worse, and that means we have to be very careful in
the choice of what we learn and teach, for unlearning is really not possible.
— E. W. DIJKSTRA in [15]

1 Introduction

Modern IT-driven societies demand highly skilled professionals who can successfully design complex systems at ever-increasing levels of reliability and security. Such a demand requires from these professionals a high degree of *mathematical fluency*, that is, the ability to resort to the mathematical language and method to build models of problems, and reason effectively within them.

However, there is little hope that such demands be met by current standards in school mathematics education. Students are not being adequately trained in formal reasoning and proof, and, as a result, they have difficulties in employing mathematics to solve new problems.

This paper describes our ideas on how to reinvigorate mathematics education. In summary, we propose a central role for formal logic, the development of a calculational style of reasoning, the emphasis on the algorithmic nature of mathematics, and the promotion of self-discovery by the students. This work is being done in the context

J. Gibbons and J.N. Oliveira (Eds.): TFM 2009, LNCS 5846, pp. 39–56, 2009.
© Springer-Verlag Berlin Heidelberg 2009

of the MathIS project[1], whose goal is to exploit the dynamics of algorithmic problem solving and calculational reasoning in both mathematics education and the practice of software engineering.

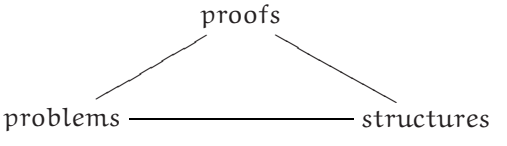
We start in section 2 by providing some background on the existing problems of the current standards in mathematics education and by explaining in detail how we think they can be improved.

We exemplify our ideas in section 3, where we present extracts of educational material that illustrate how we would rewrite and teach mathematics. The first example is a recreational problem, inspired by chess, that shows the effectiveness of a calculational formal logic. We often use recreational examples, since they can make serious concepts more palatable to students. The second example is on integer division. Elementary number theory is inherently algorithmic and we believe that exploiting this attribute can improve the way we teach it. We conclude the section by explaining how we think the material should be introduced. Our approach is based on teaching scenarios, which are detailed guidelines on how to solve specific problems. These scenarios are primarily written for teachers and they are designed to promote self-discovery by the students.

The paper is concluded by a discussion on future work, assessment, and tool support.

2 Mathematics as *the Art of Effective Reasoning*

The triangle



provides a perspective on the dynamics of mathematics as the interplay of its three vertices. Mathematics starts from *problems* which are modelled, abstracted, and generalised into precise (and hopefully, simple) *structures* upon which it becomes possible to reason formally about them, characterise possible solutions, and establish their properties. Reasoning, *i.e.*, *proofs*, raise new problems which, again, can be abstracted and generalised... The adjective *formal* not only qualifies mathematical *proofs* as rigorous, but also suggests an underlying discipline for their production and communication.

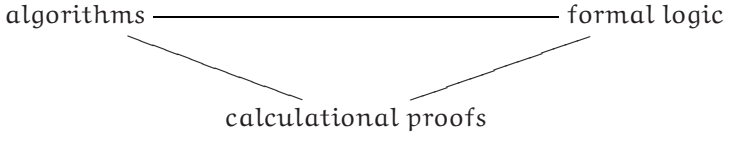
Such a discipline is often absent, or disguised, or simply left implicit, in classical mathematical texts and sometimes even regarded with suspicion. However, the presence of a discipline would have tremendous benefit because it enables students to acquire mental tools which empower their reasoning skills and make them more rigorous and productive.

Also, the notion of *proof* is almost swept under the carpet in most school manuals and in teaching practice. Teachers usually associate proofs with intricate, heavily semantic arguments, often presented in a non-systematic, pseudo-intuitive way, which

[1] MathIS is a 3-year project that has started in January 2009. More details are available from the project portal at `www.di.uminho.pt/mathis`

they (rightly) suppose an average student is not able to grasp, let alone to master, reproduce or adapt to new contexts. If proofs, as usually taught, are inadequate, a mathematical discipline relegating proofs to a minority of highly crafted students is misleading and useless. Actually, not only proving skills, i.e. the ability of formally explaining and justifying an argument, lie at the very heart of what mathematics is about, but also in modern IT societies, proofs (of functional correctness, of security compliance, etc.) have achieved the status of a core business. This is why training software engineers in formal development methods has become so important as to justify holding international conferences on the subject. We believe, however, the issue goes far deeper into the educational system and should also be addressed at such levels.

But how can the contributions of computing science improve current standards in mathematical education? As a contribution to a wider debate, we would like to single out in this paper three main topics: the emphasis on the central role of *formal logic*, the development of a *calculational* style of reasoning, and the emphasis on the *algorithmic character* common to a great number of mathematical problems. In a sense, they form the vertices of another triangle mirroring the one depicted above:



(1)

A central role for formal logic. The emphasis on the central role of *formal logic*, not only in foundations, but also in applied mathematics and as part of a number of standard engineering curricula, is perhaps the main consequence to mathematics of computing science development. An indicator of this move is the almost universal presence of a course on formal logic in every computing science or mathematics undergraduate curriculum.

Proficiency in mathematics, however, would benefit from an earlier introduction and explicit use of logic in high school. Note this is usually not the case in most European countries; the justification for such an omission is that *logic is implicit in mathematics and therefore does not need to be taught as an independent issue*. Such an argument was used in Portugal to eliminate logic from the high-school curriculum in the nineties. The damage it caused is still to be assessed, but it is certainly not alien to the appalling indicators and statistics in what concerns the country's overall ranking in mathematics education [27].

We believe that logic should be introduced using simple problems that emphasize formalization and calculation (this belief is also supported in [13]). Recreational problems like the ones we show in section 3.2, for example, are easy to understand and usually grab students' attention. Also, logic puzzles, where the goal is to solve simultaneous equations on booleans, can be introduced by analogy with simultaneous equations on numbers. High-school students already learn how to solve simultaneous equations on numbers; going from the reals to the simpler boolean domain, where each variable is either true or false, should be no problem. In fact, experiences done in Finland confirm that it is feasible and advantageous to introduce formal logic at secondary-school level [2,4,3]. It is important to note, however, that in these experiments there was no

introduction of new mathematics; this means that the use of problems which are outside the curriculum, as the ones shown in sections 3.2 and 3.3, still has to be assessed.

A calculational reasoning style. Two decades of research on *correct-by-construction* program design have created a new discipline of algorithmic problem solving and shed light on the underlying mathematical structures, modelling, and reasoning principles. Starting with the pioneering work of Dijkstra and Gries [17,20], and in particular, through the development of the so-called *algebra of programming* [12,7], a *calculational style* [5,29,16] emerged, emphasising the use of systematic mathematical calculation in the design of algorithms. This was not new, but routinely done in algebra and analysis, albeit subconsciously and not always in a systematic fashion. The realization that such a style is equally applicable to logical arguments [17,20] and that it can greatly improve on traditional verbose proofs in natural language has led to a systematization that can, in return, also improve exposition in the more classical branches of mathematics. In particular, lengthy and verbose proofs (full of *dot-dot* notation, case analyses, and natural language explanations for "obvious" steps) are replaced by easy-to-follow calculations presented in a standard layout which replaces classical implication-first logic by variable-free algebraic reasoning [29,19].

The systematization of a calculational style of reasoning, proceeding in a formal, essentially syntactic way, can greatly improve on the way proofs are presented. In particular it may help to overcome the typical justification for omitting proofs in school mathematics: that they are difficult to follow for all but exceptional students. Moreover, in school mathematics, there are many examples which show how the formalization of topics arising in different contexts results in formulae with the same *flavour*, which can be manipulated thereafter by the same rules of the predicate calculus, without reference to a 'domain specific' interpretation of such formulae in their original area of discourse. This is the essence of formal manipulation, and yields proofs that are shorter, explicit, independent of hidden assumptions, easy to re-construct, check and generalise.

That such a syntax-driven approach is extremely effective in practice cannot be understated. For example, it was the formal manipulation of Maxwell's equations that led to conjecturing the existence of electromagnetic waves, confirmed experimentally shortly afterwards. As noted by Dijkstra, presenting calculation, i.e. the manipulation of uninterpreted formulae, as an alternative to traditional, informal mathematical reasoning, accomplishes Leibniz's dream of reducing reasoning to some kind of calculation: *Traditional mathematics did not provide the most hospitable environment for its realization; this, therefore, had to take place in a separate discipline, which is now known as Computing Science* [14]. Several authors [21,28] point out that in mathematics, formal calculation is both a convenience that people, contrary to popular opinion, naturally adopt, and an asset for discovery and development. The only obstacle that keeps it from universal and systematic use throughout science (and thereby feeds prejudice) is the calculationally deficient notation that still prevails in many disciplines.

Making explicit the algorithmic content of mathematics. Another contribution of computing science to mathematical education is in the systematic identification of the algorithmic content of a large part of mathematics. Recall that, algorithmic problems are the ones where the solution involves, possibly implicitly, the design of an algorithm, i.e., a sequence of instructions that can be mechanically executed to solve it.

Algorithms have been studied since the beginning of civilization. However, the advent of the digital age has revolutionized the nature, the pace and the importance of algorithm development. The unprecedented demands on precision and concision that this entails have brought about massive improvements in our problem-solving skills [6]. Their potential for reshaping mathematical teaching, introducing powerful behaviour abstractions (such as *invariants* or *contracts*), problem decomposition techniques or goal-directed derivations, is just beginning to loom.

We can say that our work fits with what is now usually called "computational thinking" [32]; we, too, want to transfer skills created and developed within computing science and we want to illustrate the value of computational thinking to everyone interested in problem solving. In particular, we believe that mathematics education can be reinvigorated by exploring the algorithmic nature of much of its contents. Research in computational thinking is being led by the Center of Computational Thinking at Carnegie Mellon where their major activity is conducting PROBEs or PROBlem-oriented Explorations. These PROBEs are experiments that apply novel computing concepts to problems to show the value of computational thinking.

In the next section, we show two examples that can be used to introduce algorithmic skills. First, in subsection 3.3, we show how the emphasis on a calculational approach can also lead to a constructive and precise derivation of the integer division algorithm. In subsection 3.4, we present an algorithmic problem whose goal-oriented solution is based on problem decomposition and invariants.

3 An Educational Programme

How can such an 'inheritance' of computing science development be carried back to high-school mathematics and effectively improve current teaching standards?

As mentioned in the introduction, the authors are currently involved in planning and implementing a pilot educational programme, targeting high-school students, to address this question.

Our first observation is that the simple reasoning style introduced in the first year of high-school algebra is calculational. A deeper analysis of the structure of these calculations (formerly taught by examples) establishes a basis for consolidation and for extension to other, more advanced mathematical subjects. On the other hand, exploring the dynamics of algorithmic problem solving, working from concrete problems through goal-directed constructions, will strengthen their logical skills and ability to reason in an efficient way. We believe that exploring the vertices of triangle (1) has a potential to make students proficient in structuring formal arguments and aware of the central role of *proofs* in the mathematical practice. This programme will help to validate such hypothesis.

3.1 The Programme

The main component of this programme is the development of specific educational material supporting the use of a calculational approach and algorithmic problem solving strategies in the *practice of mathematics*. This material, in the form of example-driven

teaching scenarios whose structure is detailed in subsection 3.4, is designed for use with teams of up to 20 volunteer high school students in the context of extra-curricular "Maths' Clubs".

These clubs are aimed at students between 15 and 17 years old and do not require any extra-curricular prerequisite knowledge. Since the students participation is on a voluntary basis, we expect them to be above-average students. The clubs are run by volunteer secondary-school teachers, but members of the MathIS project will attend them frequently. We expect to have a close collaboration with the teachers so that we can refine the teaching scenarios and assess if the material being taught is suitable. The first club experience will start in October 2009 at a secondary school in Braga, Portugal. Note that the project aims at conducting a *pilot experience* to provide some empirical evidence relevant to our claims. Whether general recommendations on the implementation of the standard secondary school mathematics curriculum will emerge from this experiment, although desirable, is still not clear at this preliminary stage.

Our focus is placed on two domains which are both understood as strategic by the secondary school teachers collaborating with the project (mainly the first one) and attractive to students (mainly the second):

- the *refactoring* of specific areas of the high-school mathematics curriculum, to build an alternative to their usual presentation in the classroom,
- *recreational mathematics*, in the form of logic puzzles and combinatorial games, which, lying outside standard mathematics curricula, are an attractive source of non trivial examples for the envisaged techniques.

In both cases the emphasis is placed on the judicious use of formal logic in mathematical reasoning and the intertwined development of calculational proofs and algorithms, making the, often hidden, algorithmic content explicit. Both domains are illustrated in detail in the next two sub-sections. Finally, sub-section 3.4 discusses the structure of a *teaching scenario* by means of a concrete example.

3.2 Recreational Mathematics

Les hommes ne sont jamais plus ingénieux que dans l'invention des jeux.
(Men are never more ingenious than in inventing games.)
—GOTTFRIED W. LEIBNIZ to De Montmort
(29 July 1715)

Recreational mathematics is a type of mathematics that usually appeals to students and inspires them to study further. Recreational problems are often based on real-life situations, and thus, are easily understood and do not generally require an advanced knowledge of mathematics to be solved. Here we present an example inspired by chess and a logic puzzle, whose solutions can be easily obtained by calculation.

Problem (chess moves). In chess, a bishop moves along the diagonal. That is, starting from a position $(\mathtt{i},\mathtt{j})$, a bishop can move a (positive or negative) distance $\mathtt{k}$ to the position $(\mathtt{i}+\mathtt{k}\,,\ \mathtt{j}+\mathtt{k})$ or to the position $(\mathtt{i}+\mathtt{k}\,,\ \mathtt{j}-\mathtt{k})$. (This is provided, of course, that the bishop stays within the boundary of the board. See figure 1; the bishop is in position $(2,2)$.)

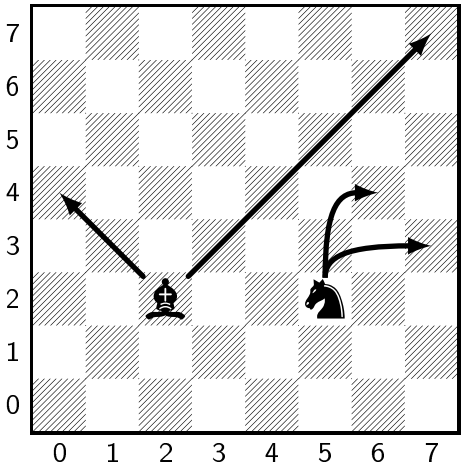


Fig. 1. Examples of chess moves: bishop and knight

Show that a move from the position (i,j) to the position $(\mathrm{i}{+}\mathrm{k}\,,\ \mathrm{j}{+}\mathrm{k})$ does not change the colour of the square. **Hint:** The definition

$$\mathit{black}.(\mathrm{i},\mathrm{j}) \ \equiv\ \mathit{even}.\mathrm{i} \ \equiv\ \mathit{even}.\mathrm{j}$$

can be useful.

A calculational solution. The goal is to prove the following two equalities:

$$\mathit{black}.(\mathrm{i},\mathrm{j}) \ \equiv\ \mathit{black}.(\mathrm{i}{+}\mathrm{k}\,,\ \mathrm{j}{+}\mathrm{k}) \tag{2}$$

$$\mathit{black}.(\mathrm{i},\mathrm{j}) \ \equiv\ \mathit{black}.(\mathrm{i}{+}\mathrm{k}\,,\ \mathrm{j}{-}\mathrm{k}) \tag{3}$$

A calculational and annotated proof of (2) is as follows:

$$\begin{array}{ll}
 & \mathit{black}.(\mathrm{i}{+}\mathrm{k}\,,\ \mathrm{j}{+}\mathrm{k}) \\
= & \quad \{ \quad \text{definition of } \mathit{black} \quad \} \\
 & \mathit{even}.(\mathrm{i}{+}\mathrm{k}) \equiv \mathit{even}.(\mathrm{j}{+}\mathrm{k}) \\
= & \quad \{ \quad \mathit{even} \text{ distributes over addition, i.e.,} \\
 & \qquad\quad [\ \mathit{even}.(\mathrm{a}{+}\mathrm{b}) \ \equiv\ \mathit{even}.\mathrm{a} \ \equiv\ \mathit{even}.\mathrm{b}\ ] \quad \} \\
 & \mathit{even}.\mathrm{i} \equiv \mathit{even}.\mathrm{k} \equiv \mathit{even}.\mathrm{j} \equiv \mathit{even}.\mathrm{k} \\
= & \quad \{ \quad \text{associativity and symmetry of } \equiv \quad \} \\
 & \mathit{even}.\mathrm{i} \equiv \mathit{even}.\mathrm{j} \equiv \mathit{even}.\mathrm{k} \equiv \mathit{even}.\mathrm{k} \\
= & \quad \{ \quad \text{associativity and reflexivity of } \equiv \quad \}
\end{array}$$

$$
\begin{array}{ll}
 & \mathit{even}.\mathtt{i} \equiv \mathit{even}.\mathtt{j} \\
= & \quad \{ \quad \text{definition of } \mathit{black} \quad \} \\
 & \mathit{black}.(\mathtt{i},\mathtt{j}) \ .
\end{array}
$$

The proof of (3) is similar and left to the reader. Note that the solution constitutes a unified interface for reasoning about how the colour of the squares change regardless of the chess piece. For example, we can use the same proof structure to prove that the move of a knight always changes the colour of the square (see figure 1; in this case, the key property is that the numbers 1 and 2 have different parities). Furthermore, the teacher can easily create new exercises just by choosing different moves.

Standard solutions to parity problems are usually done within the familiar domain of numbers. In this particular example, a standard solution would claim that the parities of $\mathtt{i}+\mathtt{k}+\mathtt{j}+\mathtt{k}$ and $\mathtt{i}+\mathtt{j}$ are the same. However, reasoning within the boolean domain can be more effective: the algebraic manipulations may be less familiar than ordinary arithmetic, but they are easier because the domain is much simpler. Another way of familiarizing the students with calculational logic is via logic puzzles. The next example shows a puzzle, where the goal is to solve simultaneous equations on booleans.

Logic Puzzle: Portia's Casket. In Shakespeare's *Merchant of Venice*, Portia had three caskets: gold, silver, and lead. Inside one of these caskets Portia had put her portrait and on each was an inscription. Portia explained to her suitor that each inscription could be either true or false but on the basis of the inscriptions he was to choose the casket containing the portrait. If he succeeded he could marry her. The exercise presented here is a simpler version (just two caskets, instead of three).

Suppose there are two caskets, gold and silver, into one of which Portia placed her portrait. The inscriptions are:

Gold: Exactly one of these inscriptions is true.
Silver: This inscription is true if the portrait is in here.

Which casket contains the portrait? What can you deduce about the inscriptions?

A calculational solution. Let G stand for "the portrait is in the gold casket", let S stand for "the portrait is in the silver casket", g stand for "the inscription in the gold casket is true" and s for "the inscription in the silver casket is true". Then, we are given:

$$(\mathsf{g} \equiv (\mathsf{g} \equiv \neg \mathsf{s})) \wedge (\mathsf{s} \equiv \mathsf{s} \Leftarrow \mathsf{S}) \wedge (\mathsf{G} \equiv \neg \mathsf{S}) \ .$$

We can simplify this expression as follows:

$$
\begin{array}{ll}
 & (\mathsf{g} \equiv (\mathsf{g} \equiv \neg \mathsf{s})) \wedge (\mathsf{s} \equiv \mathsf{s} \Leftarrow \mathsf{S}) \wedge (\mathsf{G} \equiv \neg \mathsf{S}) \\
= & \quad \{ \quad \text{definition of if} \quad \} \\
 & (\mathsf{g} \equiv (\mathsf{g} \equiv \neg \mathsf{s})) \wedge (\mathsf{s} \equiv (\mathsf{s} \equiv \mathsf{s} \vee \mathsf{S})) \wedge (\mathsf{G} \equiv \neg \mathsf{S}) \\
= & \quad \{ \quad \text{associativity and reflexivity} \quad \} \\
 & \neg \mathsf{s} \wedge (\mathsf{s} \vee \mathsf{S}) \wedge (\mathsf{G} \equiv \neg \mathsf{S})
\end{array}
$$

$$
\begin{array}{ll}
= & \{ \quad \text{negation} \quad \} \\
& (s \equiv \mathit{false}) \wedge (s \vee S) \wedge (G \equiv \neg S) \\
= & \{ \quad \text{substitution of equals for equals} \quad \} \\
& (s \equiv \mathit{false}) \wedge (\mathit{false} \vee S) \wedge (G \equiv \neg S) \\
= & \{ \quad \mathit{false} \text{ is the unit of disjunction} \quad \} \\
& (s \equiv \mathit{false}) \wedge S \wedge (G \equiv \neg S) \\
= & \{ \quad \text{reflexivity} \quad \} \\
& (s \equiv \mathit{false}) \wedge (S \equiv \mathit{true}) \wedge (G \equiv \neg S) \\
= & \{ \quad \text{substitution of equals for equals} \quad \} \\
& (s \equiv \mathit{false}) \wedge (S \equiv \mathit{true}) \wedge (G \equiv \mathit{false}) \\
= & \{ \quad \text{negation rule and reflexivity} \quad \} \\
& \neg s \wedge S \wedge \neg G \quad .
\end{array}
$$

The conclusion is that the inscription in the silver casket is false and the portrait is in the silver casket. We can't conclude anything about the inscription in the gold casket.

Note that the calculation above could be made shorter by combining several steps in one. However, we have presented it as we usually present it to our students.

3.3 Refactoring School Mathematics

The elementary theory of numbers should be one of the very best subjects for early mathematical instruction. It demands very little previous knowledge, its subject matter is tangible and familiar; the processes of reasoning which it employs are simple, general and few; and it is unique among the mathematical sciences in its appeal to natural human curiosity.
—G. H. HARDY in the sixth Josiah Willard Gibbs Lecture (New York, 1928)

We now present an example taken from elementary number theory, an area in which many important concepts are of algorithmic nature [9] [10]. In particular, we show how the specification of integer division as a Galois connection can lead to effective proofs and how to use it as a specification of the division algorithm.

Integer Division as a Galois Connection. The integer division of P by Q, here denoted by $P \div Q$, is usually introduced as the integer x such that

$$P = x \times Q + r \quad \wedge \quad 0 \leq r < |Q| \quad .$$

This formulation is usually accompanied by many examples that convey the concept of division, dividend, and remainder. However, we believe that the students do not learn how to reason effectively about division. Properties like the following

$$\langle \forall a, b, c :: (a \div b) \div c = a \div (c \times b) \rangle \tag{4}$$

are rarely discussed, and even when they are, their justification is typically informal and imprecise. Note, however, that this sort of property is often given as a *rule of thumb* in connection to exercises. Properly understanding them becomes relevant to build the correct underlying mathematical intuitions. Therefore, we propose the introduction of the integer division of P by Q as the Galois connection:

$$\langle \forall k ::\ k \times Q \leq P \ \equiv\ k \leq P \div Q \rangle \quad . \tag{5}$$

We can use this definition to effectively prove properties of integer division. For instance, replacing k by $P \div Q$, we establish the property:

$$(P \div Q) \times Q \leq P \quad .$$

We can also conclude that $0 \leq P$ is equivalent to $0 \leq P \div Q$, by replacing k by 0. Also, using indirect equality, definition (5) can be used to prove property (4) in just three steps:

$$\begin{array}{ll} & k \leq (a \div b) \div c \\ = & \quad \{ \quad \text{definition (5)} \quad \} \\ & k \times c \leq a \div b \\ = & \quad \{ \quad \text{definition (5) and associativity} \quad \} \\ & k \times (c \times b) \leq a \\ = & \quad \{ \quad \text{definition (5)} \quad \} \\ & k \leq a \div (c \times b) \quad . \end{array}$$

Moreover, definition (5) is a suitable specification for an algorithm that computes $P \div Q$. Note that the goal of such an algorithm is to compute a solution to the equation

$$x ::\ \langle \forall k ::\ k \times Q \leq P \ \equiv\ k \leq x \rangle \quad .$$

(The notation "$x ::\ E$" means that E is an equation on x.) If a solution to this equation exists, then it is unique (because the relation $\leq$ is reflexive and anti-symmetric). Furthermore, an important property of the solution x is that it is the largest integer that satisfies

$$x \times Q \leq P \quad . \tag{6}$$

As a consequence, it also satisfies

$$\neg((x+1) \times Q \leq P) \quad . \tag{7}$$

Properties (6) and (7) are the only ingredients we need to specify the division algorithm:

$$\begin{array}{l} S \\ \{\ x \times Q \leq P \ \ \wedge \ \ \neg((x+1) \times Q \leq P)\ \}\ . \end{array}$$

We now apply a common technique in algorithm development: we take the first conjunct as the invariant, since it is easy to initialise ($x := 0$), and we take the negation of the second conjunct as the loop guard. The first version of the algorithm becomes:

$$
\begin{array}{l}
\{\ 0 \le P\ \} \\
x := 0; \\
\{\ \textbf{Invariant:}\quad x \times Q \le P\ \} \\
\mathbf{do}\ \ (x+1) \times Q \le P \rightarrow x := A \\
\mathbf{od} \\
\{\ x \times Q \le P\ \ \wedge\ \ \neg((x+1) \times Q \le P)\ \}\ .
\end{array}
$$

The precondition $0 \le P$ is necessary to make the invariant initially valid. Now, calculating the assignment to x, so that the invariant is preserved, is the same as calculating A in a way that the following requirement is satisfied:

$$A \times Q \le P \ \Leftarrow\ x \times Q \le P\ \ \wedge\ \ (x+1) \times Q \le P\ \ .$$

Clearly, we can choose A to be $x+1$ and we get the next version of the algorithm:

$$
\begin{array}{l}
\{\ 0 \le P\ \} \\
x := 0; \\
\{\ \textbf{Invariant:}\quad x \times Q \le P\ \} \\
\mathbf{do}\ \ (x+1) \times Q \le P \rightarrow x := x+1 \\
\mathbf{od} \\
\{\ x \times Q \le P\ \ \wedge\ \ \neg((x+1) \times Q \le P)\ \}\ .
\end{array}
$$

For brevity, we do not show the full derivation. Instead, we would like to stress that the derivation of the division algorithm is an educational example that can be used to teach algorithmic techniques such as loop formation, using the invariant to calculate assignments, and proving progress.

Another technique that can be taught is the introduction of extra variables and computations to produce more efficient versions. In this example, the algorithm above can be further optimized by introducing the computation of the remainder.

Also, the calculational approach allows us to be more constructive because the requirements emerge from the calculations. As an example, we do not need to assume that the divisor Q is positive; it emerges as a necessary condition in the proof that the bound-function is bounded below.

Recall that a *bound function* is a natural-number-valued function of the program variables that measures the size of the problem to be solved. A guarantee that the value of such a bound function is always decreased at each iteration is a guarantee that the number of times the loop body is executed is at most the initial value of the bound function. In this example, a possible bound function is $P-x$, and the proof that it is bounded below is as follows:

$$
\begin{array}{ll}
 & 0 \leq P - x \\
= & \quad \{ \quad \text{cancellation} \quad \} \\
 & x \leq P \\
= & \quad \{ \quad \text{we know from the invariant that } x \times Q \leq P; \\
 & \qquad \textbf{assuming that } 0 < \mathbf{Q}, \text{ we have } x \leq x \times Q; \\
 & \qquad \text{because } \leq \text{ is transitive, we also have } x \leq P \quad \} \\
 & \mathsf{true} \ .
\end{array}
$$

Note that the assumption $0 < Q$, highlighted in bold in the calculation, emerges naturally from the shape of the invariant. As a result, the next version of the algorithm would include $0 < Q$ as a precondition to guarantee that the algorithm terminates.

3.4 Teaching Scenarios

What is teaching?
In my opinion, teaching is giving opportunity
to the students to discover things by themselves.
— GEORGE PÓLYA in TEACHING US A LESSON
(MAA Video Classics, Number 1)

The success of teaching depends on the amount of discovery that is left for the students: if the teacher discloses all the information needed to solve a problem, students act only as spectators and become discouraged; if the teacher leaves all the work to the students, they may find the problem too difficult and become discouraged too. It is thus important to find a balance between these two extremes.

We propose the introduction of educational material in the form of teaching scenarios, which are fully worked out solutions to algorithmic problems together with "method sheets"— detailed guidelines on the principles captured by the problem, how the problem is tackled, and how it is solved. Although they can be used directly by the student, they are primarily written for the teacher and they are designed to maintain a balance between the two extremes mentioned above. In other words, they are designed to promote self-discovery. In general, each scenario is divided into the following sections:

- **Brief description and goals.** This section provides a summary of the scenario, allowing the teacher to determine if it is adequate for the students.
- **Problem statement.** This section states the problem (or problems) discussed in the scenario.
- **Students should know.** This section lists pre-requisites that should be met by the students. The teacher can use it to determine if the scenario is adequate for the students.
- **Resolution.** This section presents a possible solution for the problem in the style advocated here.

- **Notes for the teacher.** In this section, the solution presented above is decomposed into its main parts and each part is discussed in detail. To maintain the balance mentioned in the first paragraph, we also recommend how the teacher should present the material, including questions that the teacher should or should not ask and important concepts that should be introduced.
- **Extensions and exercises.** This section can be used for homework or project assignments. All the exercises are accompanied by their solutions.
- **Further reading.** Recommended reading for the teacher and the students. It may include discussions and comparisons between conventional solutions and the one presented in the scenario.

The concept of teaching scenarios is also used in the project "Computer Science Unplugged" [11], whose goal is to teach principles of computing science through games and puzzles. They provide a series of activity worksheets that can be used in the classroom. These worksheets are similar to the scenarios we are developing, but the goals are slightly different: whilst they want to convey general principles and ideas of computing, we want to focus on calculational and algorithmic principles that can be used to reinvigorate mathematics. Also, their project is aimed at primary-aged children and our pilot programme is aimed at pre-university level students.

We now present some extracts from a scenario that generalizes the problem "The Chameleons of Camelot", found in [23, p. 140] (a more recent and acessible reference is [33]). Its goal is to help students recognizing, modelling, and solving algorithmic problems. The solution is goal-oriented and explores an invariant of the underlying non-deterministic algorithm. It is also an example of problem decomposition and it can be used to convey the notions of loop, guard, postcondition, and non-determinism. To obtain the full version, please visit the website `http://joaoff.com/aps/scenarios`.

The Chameleons of Camelot. On the island of Camelot there are three different types of chameleons: grey chameleons, brown chameleons, and crimson chameleons. Whenever two chameleons of different colours meet, they both change colour to the third colour.

For which numbers of grey, brown, and crimson chameleons is it possible to arrange a succession of meetings that results in all the chameleons displaying the same colour?

For example, if the number of the three different types of chameleons is 4, 7, and 19 (irrespective of the colour), we can arrange a succession of meetings that results in all the chameleons displaying the same colour. An example is:

$$(4\,,7\,,19)\rightarrow(6\,,6\,,18)\rightarrow(0\,,0\,,30)\quad .$$

On the other hand, if the number of chameleons is 1, 2, and 3, it is impossible to make them all display the same colour.

(Note that this problem is more general than in conventional presentations, since we are interested in characterizing the initial numbers of chameleons for which there is a solution, rather than working with particular values.)

Notes for the teacher. As said before, teaching scenarios are primarily written for the teacher and are designed to promote self-discovery. The following extract illustrates the general tone of the section *Notes for the teacher*; in particular, it suggests a formalization of the goal and how to decompose the problem:

– **Determine the postcondition and decompose the problem.** Now that we have modelled the underlying algorithm, we have to express our goal. The answer comes directly from the problem statement: "For which numbers of grey, brown, and crimson chameleons is it possible to arrange a succession of meetings that results in all the chameleons displaying the same colour?". So we need to express formally that all the chameleons display the same colour. One alternative is[2]:

$$g = b = 0 \vee b = c = 0 \vee c = g = 0 \quad . \tag{8}$$

An informal description can be useful (e.g. "Provided that there is at least one chameleon, the first disjunct means that there are only crimson chameleons, the second means that there are only green chameleons, and the third means that there are only brown chameleons. Their disjunction means that at least one of these statements is true."). At this stage, the teacher should note that instead of working directly with the final goal (8), we can think of how to get to intermediate states that simplify the problem. The teacher should lead the students to the observation that if any two types of chameleons are equally numbered, we can arrange a meeting between all the chameleons of these two types. We suggest the teacher start with some concrete examples until the students get there (e.g. $(0\,,0\,,3)$, $(171\,,10\,,10)$, and $(5\,,3\,,5)$). Formally, we can express these states as:

$$g = b \;\vee\; b = c \;\vee\; c = g \quad . \tag{9}$$

If the algorithm reaches a state that satisfies this expression, it remains to arrange a meeting between all the chameleons of two equally numbered classes.
(...)

The two following extracts show how to determine an invariant and how to use it to solve the problem:

– **Determine appropriate invariants.** Now that we have formalized our algorithm and goal, there is not much left to do other than to investigate how (9) behaves under the three loop assignments[3]. A standard technique is to work from the postcondition, using the assignment axiom:

$$\{\, Q[v := e] \,\}\, v := e \,\{\, Q \,\} \quad ,$$

[2] Variables g, b, and c have been introduced early in the scenario as the number of green, brown, and crimson chameleons, respectively.

[3] The introduction of the loop assignments is not shown here. Please see the full scenario for details.

where $v := e$ represents an assignment and Q is the postcondition. Taking the first disjunct of (9), we calculate how it behaves under the three assignments. We start with the first assignment:

$$
\begin{array}{ll}
 & (g = b)[g, b, c := g - 1, b - 1, c + 2] \\
= & \quad \{ \text{ substitution } \} \\
 & g - 1 = b - 1 \\
= & \quad \{ \text{ cancellation } \} \\
 & g = b \quad ;
\end{array}
$$

Now, the second assignment:

$$
\begin{array}{ll}
 & (g = b)[g, b, c := g - 1, b + 2, c - 1] \\
= & \quad \{ \text{ substitution } \} \\
 & g - 1 = b + 2 \\
= & \quad \{ \text{ cancellation } \} \\
 & g = b + 3 \quad ;
\end{array}
$$

Finally, the third assignment:

$$
\begin{array}{ll}
 & (g = b)[g, b, c := g + 2, b - 1, c - 1] \\
= & \quad \{ \text{ substitution } \} \\
 & g + 2 = b - 1 \\
= & \quad \{ \text{ cancellation } \} \\
 & g + 3 = b \quad .
\end{array}
$$

The teacher should ask the students if they see any pattern in the calculations. The discussion should lead to the fact that the number of grey and brown chameleons after any meeting is either the same, or it differs by 3. A concise way of expressing this fact is by using congruences:

$$g \cong b \pmod 3 \quad .$$

Using the same reasoning for the other two disjuncts, we conclude that an invariant of the loop is

$$g \cong b \pmod 3 \ \vee \ g \cong c \pmod 3 \ \vee \ b \cong c \pmod 3 \quad .$$

– **Discuss initial values.** There is one final step that needs to be done: we have to guarantee that the invariant found in the previous section is satisfied initially. If the invariant is valid initially, it will remain valid after each iteration, and it is possible to attain a state satisfying (9). (It is important that the students understand what an invariant is and how it is being used here to solve the problem.) In conclusion, any initial values g, b and c that satisfy the invariant allow a succession of meetings that results in all the chameleons displaying the same colour. We suggest the teacher to go through the initial examples once again to see which ones satisfy the invariant.

The section also includes questions that we recommend the teacher to ask, together with a justification for its importance:

- *What is our goal? What do we want to prove? How do we express it formally?*

  Every time we are working in a goal-oriented fashion, this question should be asked explicitly. The teacher may need to help the students formalizing the states where there are chameleons of only one colour; if that is the case, we suggest they help with the first disjunct and let the students do the other two.

To emphasize the self-discovery nature of the scenarios, we also include obtrusive questions that the teacher should not ask:

- *Can you see that if two different types of chameleons are equally numbered, the problem is easy to solve?*

  The teacher should start by asking the students which states make the problem easy to solve. With the help of some examples, we believe that most students will get to the fact that if two different types of chameleons are equally numbered, the problem is easy to solve.

Self-discovery is also promoted by the sections *Extensions and exercises* and *Further reading*, which are both designed to encourage further work by the students.

4 Conclusions and Future Work

Our own experience in teaching formal methods at the university tells us (and employers of our students confirm so) that good thinking habits rooted in sound principles add to overall effectiveness and aptness to face adversity and unexpected challenges. If explicit examples need to be found on the advantages of a sound mathematical basis in software engineering, the 2004 collapse of the Portuguese school teacher allocation system [24] serves as a typical illustration: the problem was later solved [1] by a small software house which claims to use formal methods in their normal practice. This situation drove the country's attention to the need for better trained software engineers.

Our claims in this paper are that there is a need to act at lower levels of the educational system and that a number of issues arising from computing science research may have a decisive impact on reinvigorating mathematics education to meet the challenges of modern IT-driven societies. Stressing the algorithmic content of mathematics, for example, already led us to novel results in number theory [9] [10].

It is important to note that there is more educational material than the examples shown in this paper. We have recently used algorithmic techniques to rewrite some material on elementary number theory [10] and we are currently developing a package of teaching-scenarios to use in our pilot experience. In fact, we believe that the project can only succeed if there is an abundance of material and guides ready for the teachers to use. In our opinion, providing resources and assistance to the teachers is the best way to overcome the challenge of convincing them to use the approach we propose.

One problem we foresee is the difficulty in assessing the impact of our project. The use of test and control groups, randomized trials, and assessment based on lectures to students in Math's Clubs have serious flaws. (Some of the difficulties involved in the assessment are pointed out by Herbert Wilf in his essay [31].) Nevertheless, the novel results mentioned above and preliminary results on the didactical suitability of the calculation format obtained within the group (see [18]) encourage us to continue our efforts. Also, the success claimed by related work like [2] and [26], makes us believe that we can have a positive impact. The possibility of adopting qualitative research procedures, based on case-studies and collaborative action-research projects [22], to assess the proposed methodologies is under consideration if a suitable cooperation with education researchers is achieved.

A topic we have omitted in this paper but which is central to the MATHIS project concerns the development of tool support. Actually, this capitalizes on recent developments and increased flexibility in human-computer interaction technology, which we believe is mature enough to provide an infra-structure for the envisaged methodological shift. A collection of tools to enable on-screen calculation with mathematical formulae in a blackboard-like style, resorting to tablet PCs and e-learning platforms, is under development [25]. A direct inspiration for such tools is Math/pad [8,30], a general-purpose structure editor for on-screen algebraic calculation.

Acknowledgements. Long-term collaboration with J. N. Oliveira on calculational approaches to mathematics is deeply acknowledged. We are also grateful to the anonymous referees for their valuable comments. This research was supported by FCT (the Portuguese Foundation for Science and Technology), in the context of the MATHIS Project under contract PTDC/EIA/73252/2006. The work of João F. Ferreira and Alexandra Mendes was further supported by FCT grants SFRH/BD/24269/2005 and SFRH/BD/29553/2006, respectively.

References

1. ATX. Algoritmo de Colocação de professores (in Portuguese). ATX Software press release (November 2004)
2. Back, R.-J., Mannila, L., Peltomaki, M., Sibelius, P.: Structured derivations: A logic based approach to teaching mathematics. In: FORMED 2008: Formal Methods in Computer Science Education, Budapest (2008)
3. Back, R.-J., von Wright, J.: Mathematics with a little bit of logic: Structured derivations in high-school mathematics
4. Back, R.-J., von Wright, J.: Doing high school mathematics carefully. Technical report (1997)
5. Backhouse, R.C.: Mathematics and programming. A revolution in the art of effective reasoning. Inaugural Lecture, School of Computer Science and IT, University of Nottingham (2001)
6. Backhouse, R.C.: Program Construction. John Wiley and Sons, Inc., Chichester (2003)
7. Backhouse, R.C., Hoogendijk, P.F.: Elements of a relational theory of datatypes. In: Möller, B., Schuman, S., Partsch, H. (eds.) Formal Program Development. LNCS, vol. 755, pp. 7–42. Springer, Heidelberg (1993)
8. Backhouse, R.C., Verhoeven, R.: Mathspad: A system for on-line preparation of mathematical documents. Software - Concepts and Tools 18, 80–89 (1997)

9. Backhouse, R., Ferreira, J.F.: Recounting the rationals: Twice! In: Audebaud, P., Paulin-Mohring, C. (eds.) MPC 2008. LNCS, vol. 5133, pp. 79–91. Springer, Heidelberg (2008)
10. Backhouse, R., Ferreira, J.F.: On Euclid's algorithm and elementary number theory (2009) (submitted for publication), http://joaoff.com/publications/2009/euclid-alg/
11. Bell, T., Witten, I.H., Fellows, M.: Computer Science Unplugged: An enrichment and extension programme for primary-aged children (December 2006), http://csunplugged.org/index.php/en/books
12. Bird, R., Moor, O.: The Algebra of Programming. Series in Computer Science. Prentice-Hall International, Englewood Cliffs (1997)
13. Boute, R.: Using Domain-Independent Problems for Introducing Formal Methods. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 316–331. Springer, Heidelberg (2006)
14. Dijkstra, E.W.: A new science, from birth to maturity. note EWD1024 (1988)
15. Dijkstra, E.W.: On the cruelty of really teaching computing science. note EWD1036 (1988)
16. Dijkstra, E.W.: On the economy of doing mathematics. note EWD1130 (1992)
17. Dijkstra, E.W., Scholten, C.S.: Predicate Calculus and Program Semantics. Springer, Heidelberg (1990)
18. Ferreira, J.F., Mendes, A.: Student's feedback on teaching mathematics through the calculational method. In: 39th ASEE/IEEE Frontiers in Education Conference. IEEE, Los Alamitos (2009)
19. Gries, D., Feijen, W.H.J., van Gasteren, A.J.M., Misra, J.: Beauty is our Business. Springer, Heidelberg (1990)
20. Gries, D., Schneider, F.: A Logical Approach to Discrete Mathematics. Springer, Heidelberg (1993)
21. Gries, D.: Improving the curriculum through the teaching of calculation and discrimination. Communications of the ACM 34(3), 45–55 (1991)
22. Guba, E., Lincoln, Y.: Competing paradigms in qualitative research. In: Denzin, N., Lincoln, Y. (eds.) Handbook of qualitative research, pp. 105–117. Sage, London (1994)
23. Honsberger, R.: In Polya's Footsteps: Miscellaneous Problems and Essays (Dolciani Mathematical Expositions). The Mathematical Association of America (October 1997)
24. ME. Declaração sobre o processo de colocação de professores para o ano lectivo 2004-05 (in Portuguese). Government press release (September 2004)
25. Mendes, A.: Work in progress: Structure editing of handwritten mathematics. In: 38th ASEE/IEEE Frontiers in Education Conference. IEEE, Los Alamitos (2008)
26. Michalewicz, Z., Michalewicz, M.: Puzzle-based Learning: Introduction to Critical Thinking, Mathematics, and Problem Solving, 1st edn. Hybrid Publishers (2008)
27. OCDE Report. Education at a clance: OCDE indicators 2006. OCDE Publishing, Paris (2006)
28. DIMACS Symposium. Teaching logic and reasoning in an illogical world. Technical report, Rutgers University (1996)
29. van Gasteren, A.J.M.: On the Shape of Mathematical Arguments. LNCS, vol. 445. Springer, Heidelberg (1990)
30. Verhoeven, R., Backhouse, R.C.: Towards tool support for program verification and construction. In: Wing, J.W.J., Davies, J. (eds.) FM 1999 - Int. Formal Methods Symposium. LNCS, vol. 1709, pp. 1128–1146. Springer, Heidelberg (1999)
31. Wilf, H.S.: Can there be research in mathematical education? http://www.math.upenn.edu/~wilf/website/PSUTalk.pdf
32. Wing, J.M.: Computational thinking. Commun. ACM 49(3), 33–35 (2006)
33. Winkler, P.: Puzzled: Understanding relationships among numbers. Commun. ACM 52(5), 112 (2009)

What Top-Level Software Engineers Tackle after Learning Formal Methods: Experiences from the Top SE Project

Fuyuki Ishikawa[1], Kenji Taguchi[1], Nobukazu Yoshioka[1], and Shinichi Honiden[1,2]

[1] GRACE Center,
National Institute of Informatics,
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, Japan
[2] Graduate School of Information Science and Technology,
The University of Tokyo,
7-3-1 Hongo, Bunkyo-ku, Tokyo, Japan

Abstract. In order to make practical use of formal methods, it is not sufficient for engineers to obtain general, fundamental knowledge of the methods and tools. Actually, it is also necessary for them to carefully consider their own contexts and determine adequate approaches to their own problems. Specifically, engineers need to choose adequate methods and tools, determine their usage strategies, and even customize or extend them for their effective and efficient use. Regarding the point, this paper reports and discusses experiences on education of formal methods in the Top SE program targeting software engineers in the industry. The program involves education of a variety of scientific methods and tools with group exercises on practical problems, allowing students to compare different approaches while understanding common principles. In addition, the program involves graduation studies where each student identifies and tackles their own problems. Statistics on problem settings in the graduation studies provide interesting insights into what top-level engineers tackles *after* learning formal methods.

1 Introduction

Formal methods are attracting increasing attentions from the software industry, as software is getting more and more complex while efficiency and reliability of software development is getting more and more significant. On the other hand, there has been a gap between knowledge and techniques that software engineers in the industry generally have and those required for use of formal methods. It is thus essential to provide a place where engineers in the industry can learn formal methods, expecting their practical application.

When engineers apply formal methods to problems in practical system development, they need to tackle the difficulty in determining adequate approaches to their own problems, carefully considering their own contexts. Specifically, they need to choose adequate methods and tools, determine their usage strategies, and

J. Gibbons and J.N. Oliveira (Eds.): TFM 2009, LNCS 5846, pp. 57–71, 2009.
© Springer-Verlag Berlin Heidelberg 2009

even customize or extend them for their effective and efficient use. Therefore, education of general, fundamental knowledge and techniques of formal methods and tools is not sufficient by itself.

Regarding this point, the Top SE program in Japan has provided a unique place to produce "superarchitects" who can promote practical use of advanced, scientific methods and tools, including formal methods, for tackling problems in software engineering [1,2]. The Top SE program primarily targets engineers in the industry and provides education on a variety of methods and tools by combining lecturers from the academia and the industry. Considering the points discussed above, the program provides more than education of general, fundamental knowledge and techniques on methods and tools, in the following two forms of activities.

Lecture Courses. In the Top SE project, lecture courses are organized to involve different methods and tools so that students can compare different approaches while understanding common principles. Each course involves group exercises where students jointly tackle practical problems while exchanging ideas on different approaches to the problems. The problems are jointly developed by lecturers from the academia and the industry.

Graduation Studies. The Top SE program also includes graduation studies where students identify and tackle their own problems using scientific approaches they have learned. The studies are finally evaluated in terms of validity of problem setting, validity of approach (method/tool) selection, and problem-solving ability (e.g., adequate abstraction).

The Top SE project successfully completed its setup phase of five years with government sponsorship, through which 61 students have graduated and 21 lecture courses have been developed. In April 2009, it started a renewed phase based on the established education system, and welcomed 31 new students, out of which 29 are engineers from the industry.

This paper reports and discusses the experiences on formal methods education based on the principles described above in the Top SE project. Besides design of lecture courses, this paper focuses on the graduation studies. The statistics on graduation studies will clarify what kinds of issues the students (actually top-level engineers in the industry) find significant and also solvable to some extent by themselves. It will give interesting suggestions regarding roles of the academia and the industry for promotion of widespread, practical use of formal methods.

The remainder of the paper is organized as follows. Section 2 describes the principles and current status of the Top SE program. Section 3 describes lecture course designs on formal methods in the program. Section 4 reports actual topics of graduation studies on formal methods tackled by the students. Section 5 gives and discusses statistics obtained in the experiences, and Section 6 finally concludes the paper.

2 Top SE Program

2.1 Principles

In the software engineering area, there has been a gap between what are taught in the academia and what are required by the industry [3]. The Top SE program in Japan is organized to bridge the industry-academia gap by providing a place where the academic and the industry jointly deliver knowledge and techniques for practical use of advanced scientific methods and tools [1,2]. Figure 1 illustrates the principles of the Top SE program, which involves digital home appliances as one of the target practical areas.

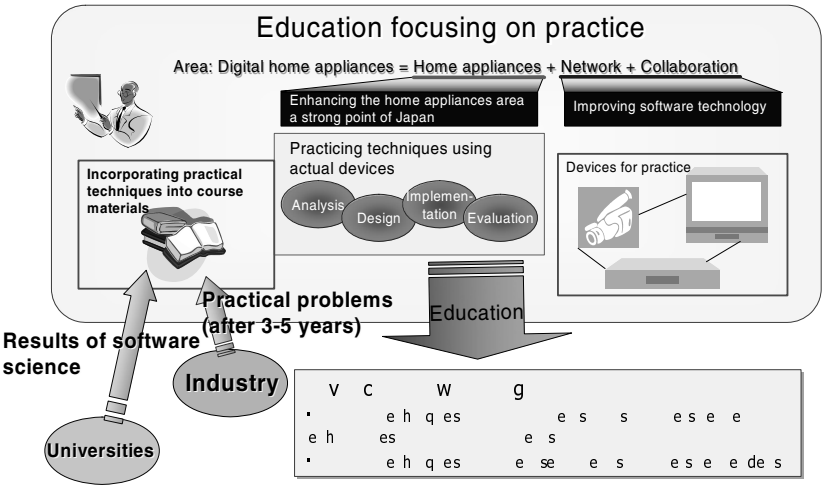


Fig. 1. Approach to Educate SuperArchitects, with the Example of Digital Home Appliances

Below summarizes principles of the Top SE program.

Target Topics. Topics cover software engineering, especially focusing upper stream processes where scientific methods and tools work effectively and efficiently. As essential goals in software engineering, efficiency, reliability, and changeability are investigated. In addition, security is also focused on as it is now an essential aspect in a variety of systems. Either hardware technologies or social matters are not considered as primary topics.

Target Students. As the original motivation is to deliver scientific approaches to the industry, the primary target students are software engineers from the industry. They know issues in software development well but do not know so much about use of scientific methods and tools. Especially, engineers around their 30's are targeted as they have identified issues in software engineering and are leading next-generation development processes. The program also

accepts graduation students of universities, who know scientific aspects well and are eager to learn their application in practical software development. For education of university students, the Top SE program also provides lecture courses at universities, which puts more focuses on delivering practical aspects to graduate students. This is out of the scope of this paper.

Produced Graduates. The project defines "superarchitects" to produce, who have knowledge and techniques of the following.

- Abstraction of practical problems into (semi) formal models, e.g., UML, automata, formal specifications, and goal-oriented requirements models.
- Application of tools to concrete problems, e.g., digital, networked home appliances.
- Adaptability to new technology and tools, e.g., identifying essential differences and common principles in similar but different approaches.
- Ability to promote the technologies and tools, e.g., instructing development teams.

Lecturers and Lecture Courses. Figure 2.1 illustrates the approach of the Top SE program to lecture course organization. Lecturers from the academia and the industry are grouped and develop lecture course materials, which involve results of software science from the academia combined with practical problem (exercise) settings from the industry. Lecture courses are organized to involve different methods and tools so that students can compare different approaches while understanding common principles. In addition, each course involves group exercises where students jointly tackle practical problems while exchanging ideas on different approaches to the problems even if using the same method and tool.

Graduation Studies. The students finally tackle graduation studies, for a few months, where they identify and tackle their own problems using scientific approaches they have learned. Students often determine to choose and apply some methods and tools for their own problem areas, e.g., workflow management system. Some students develop extension of existing methods and tools so that the methods and tools are used more effectively and efficiently for a certain class of problems, e.g., development of a model generator from a specific format. Graduation studies are evaluated in terms of validity of problem setting, validity of approach (method/tool) selection, and problem-solving ability (e.g., adequate abstraction).

Detailed description and discussion on the principles of the Top SE project are found in [1].

2.2 Current Status

The Top SE project had been fully sponsored by the Japanese government in its setup phase of 5 years until March 2009. During the phase, it accepted students three times (in 2005, 2006, and 2007) and provided them an education program of one year and half (for free). The number of the students and the number of the lecture courses were gradually increased. The third-term students (from

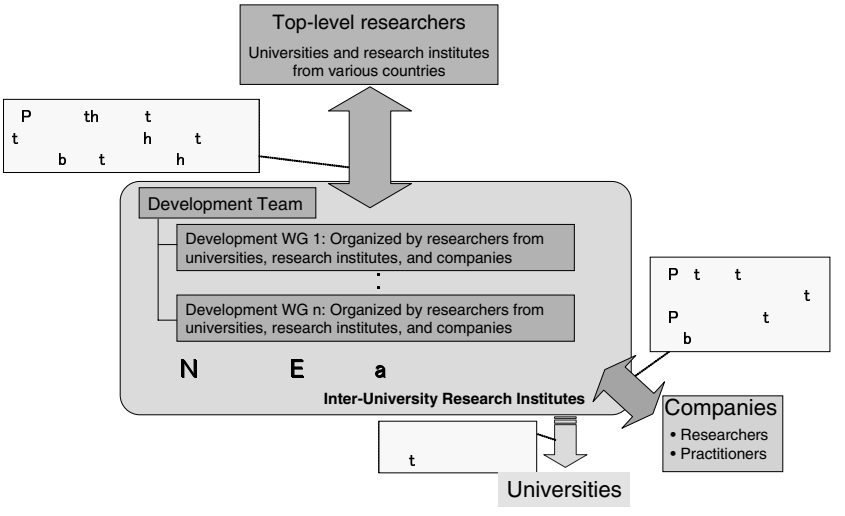


Fig. 2. Approach to Organize and Develop Lecture Courses

September 2007 to March 2009) enjoyed the educational system developed so far, which is planned to be maintained and refined continuously.

Below describes the status of the Top SE project at the end of the setup phase (March 2009).

Students. In March 2009, 30 students graduated. 4 of them were graduate students (of universities) and the others were software engineers from the industry. 61 students graduated in total in the setup phase.

Lectures. 21 lecture courses, classified in 6 series plus introductory courses, were finally developed and provided for the third-term students as shown in Table 1. Each lecture course involved 12 classes, each of which was held for one hour and a half. Classes were held in the evening (16:30-18:00, 18:15-19:45) on weekdays. Lecture courses were developed and operated by 25 lecturers, 15 from the academia (universities or research institutes) and 10 from the industry (companies). Each student must get through at least 8 courses with credit for graduation.

In addition, postgraduate support has been established on the basis of a partnership with a graduate university. A graduate of the Top SE program can proceed to doctoral course with advanced standing as well as continuous support of the supervisor of the graduation study. In April 2009, 7 of the graduates entered the doctoral course of the partner graduate university.

After successful completion of the setup phase fully sponsored by the government, the renewed Top SE project started in April 2009. In response to feedbacks obtained in the setup phase, the educational system was renewed, e.g., the start time of the lectures was moved to later in the evening. Although it became a

Table 1. Lecture Courses in the Top SE Program

Series	Course
Foundations	Fundamental Theories
	Practice of Software Engineering
Architecture	Component-based Development
	Software Patterns
	Aspect-Oriented Development
Formal Specification	Formal Specifications (Foundations)
	Formal Specifications (Applications)
	Formal Specifications (Security)
Model Checking	Verification of Design Models (Foundations)
	Verification of Design Models (Applications)
	Verification of Performance Models
	Modeling and Verification of Concurrent Systems
Requirements Analysis	Goal-Oriented Requirements Analysis
	Requirements Elicitation and Identification
	Security Requirements Analysis
	Early Requirements Analysis
Implementation Techniques	Testing
	Program Analysis
	Verification of Implementation Models
Management	Software Metrics
	Software Development Management

fare-paying education with some scholarship, 31 students joined and started to study (almost the same number as that of the third-term students).

This paper discusses formal methods education in the Top SE program, on the basis of experiences in the setup phase. Section 3 describes design of lecture courses. Specifically, among the courses enumerated in Table 1, the Formal Specification series and the Model-based Verification series are discussed as they primarily deal with formal methods. In addition, part of the Implementation Techniques series is discussed, which includes formal methods targeting source codes. Section 4 describes graduation studies on formal methods. Section 5 discusses detailed statistics on lecture courses and graduation studies.

3 Lecture Courses on FM

3.1 Model Checking Series

The Model Checking series focuses on mathematical modeling of software behaviors and their efficient verification with automated tools. According to the principles described in 2.1, The series consists of four lecture courses as illustrated in Figure 3.

MC1: Verification of Design Models (Foundations). This course delivers fundamental knowledge and techniques for use of model checking methods.

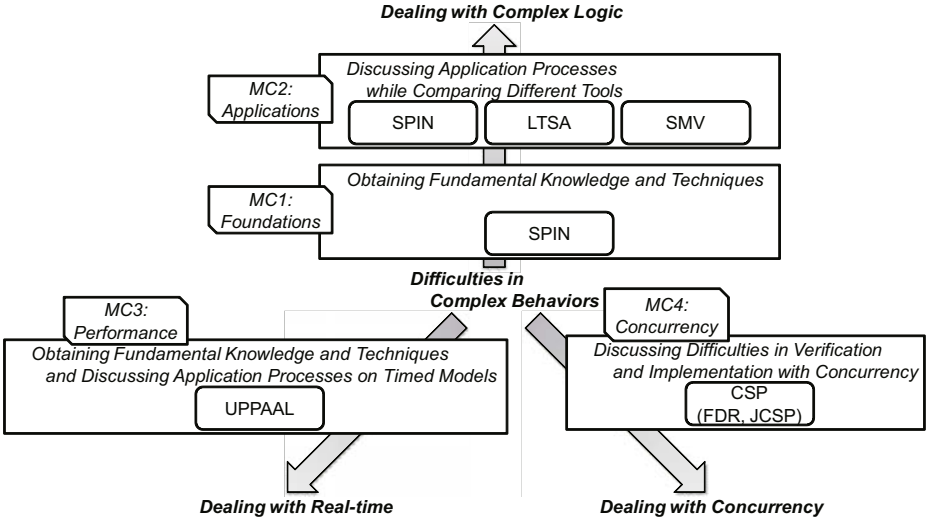


Fig. 3. Model Checking Series

SPIN [4] is used as one of the most sophisticated tools. Besides basic notations in Promela and usage of SPIN, the course explicitly focuses on the verification process, including construction of models from the design, identification of property descriptions, and validation of the verification activity itself. This process is organized systematically, involving UML modeling and tool-independent model design. For group exercise on a practical problem, coordination of networked home appliances is investigated with incorporation of actual devices. The problem setting is based on emerging practical situation to be tackled actively by Japanese companies.

MC2: Verification of Design Models (Applications). This course delivers advanced knowledge and techniques for application of model checking methods. SPIN [4], LTSA [5] and SMV/NuSMV [6] are used to tackle the same problem so that students can discuss what are common and what are different in those tools. The course also puts more focus on the problems of adequate abstraction considering the verification purpose as well as careful modeling of external environments that can lead to unexpected behaviors. For group exercise, each group defines a problem such as cellular phone control, and investigates it by discussing which tool(s) to use.

MC3: Verification of Performance Models. This course delivers knowledge and techniques for use of model checking methods for real-time systems, considering the performance aspect. UPPAAL [7] is used as one of the most sophisticated tools. The course also discusses guidelines and pitfalls in modeling timing aspects. As a practical problem, an audio-control protocol with bus collision is investigated where timing of voltage changes is essential for correct communication.

MC4: Modeling and Verification of Concurrent Systems. This course delivers knowledge and techniques for using formal models for verification and implementation of concurrent systems. CSP is used as a formalism to model concurrency, with a tool for verification of equivalence and refinement relationships, FDR2 [8], as well as a tool for deriving implementation, JCSP [9]. As difficulties in concurrent systems are so common, students define their own practical problem settings for the group exercise in this course, such as cellular phone protocols and voting protocols.

3.2 Formal Specification Series

The Formal Specification series focuses on mathematical modeling of system states and their changes through operation invocations. According to the principles described in 2.1, the series consists of three lecture courses as illustrated in Figure 4.

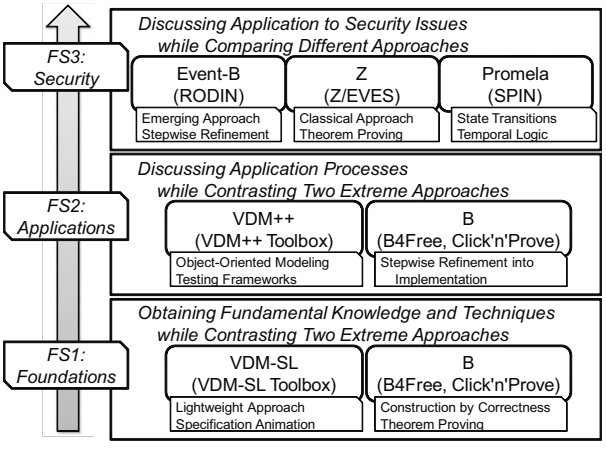


Fig. 4. Formal Specification Series

FS1: Formal Specification (Foundations). This course delivers fundamental knowledge and techniques for use of formal specification. Two different methods are discussed to deliver common principles and different decisions: B-Method aiming at correct derivation of implementation based on theorem proving and VDM aiming at lightweight validation based on specification animation and testing. As tools, B4Free and Click'n'Prove [10] are used for B-Method and VDM-SL Toolbox [11] for VDM. As a practical problem, a standard routing protocol for ad-hoc networks is investigated, which is an emerging problem with clear specifications in natural languages. Here B-Method is used to gradually introduce complexity by modeling distribution of data among nodes through stepwise refinement, while VDM is used to model behaviors of each node.

FS2: Formal Specification (Applications). This course delivers advanced knowledge and techniques for application of formal specification. B-Method and VDM are used similarly to **FS1**, but more focus is put on different levels of abstraction well as stepwise refinement through them. As tools, B4Free and Click'n'Prove are used for B-Method and VDM++ Toolbox [11] for VDM. As a practical problem, other aspects of the routing protocol for ad-hoc networks is investigated.

FS3: Formal Specification (Security). This course delivers knowledge and techniques for using formal specifications in development of secure systems. Three different methods and tools are discussed with the same practical problem of access control in a hospital. The first is Event-B [12] with the RODIN tool, where event-based modeling and stepwise refinement are investigated. The second is the Z/EVES tool [13], where theorem proving is investigated. The last is SPIN [4], where state transition-based modeling and temporal properties are investigated.

3.3 Implementation Techniques Series

The Implementation Techniques series complement focuses on verification tasks during and after implementation. Regarding formal methods, it complement with the above two series by providing lecture courses on methods and tools targeting source codes.

IM1: Program Analysis. This course delivers fundamental knowledge and techniques for analysis of programs. JML [14] is used and discussed from the view points of a means for Design by Contract, a means for unit testing, and a means for static verification of source codes. As practical problems, an access control system at a hospital and an on-line shopping system are investigated.

IM2: Verification of Implementation Models. This course delivers advanced knowledge and techniques for application of model checking to verification of source codes. Java PathFinder [15] is used and discussed. As a practical problem, network protocols are investigated. More detailed description and discussion on the course is found in [16].

4 Graduation Studies on FM

As described in 2.1, students identify and tackle their own problems in their graduation studies. Many students have chosen topics on formal methods. This paper classifies the topics as follows.

Case Study. In this type of study, problems are identified in development of some target systems, such as workflow management system and web application. Solutions are defined by choosing adequate formal methods/tools as well as defining application strategies. Below describes a few examples of this type of graduation studies.

- One of the studies used VDM for formal modeling and validation in an experimental project involving a team at the company with which the student works. It evaluated additional costs (man-hours) as well as specification items additionally identified through the modeling and validation, which would lead to much more cost if found in later phases.
- Another study used SPIN and NuSMV, respectively, for verification of access control policies on shared file operations, and detected situations where a high-level policy of confidentiality or availability is broken (not correctly implemented) by low-level policies.

Domain-Specific Finer-Grained Support. In this type of study, problems are identified in application of formal methods/tools to development of systems in some domains, such as workflow management system and web application. Solutions are defined by developing domain-specific methods/tools that provide finer-grained support for application of general formal methods/tools. Below describes a few examples of this type of graduation studies.

- One of the studies investigated translation rules from BPMN [17], a standard notation of business processes, to the notation of timed automata in UPPAAL. After adding timing constraints to a BPMN process, UPPAAL automata are obtained according to the translation rules. The study had a case study based on a simplified version of examples in the BPMN specification, where introduction of timing constraints makes it difficult to achieve consistency.
- Another study investigated generation of models from configurations in Web applications, namely, configuration files in Struts and JSP files [18].

Bridging Gaps between Different Methods/Tools. In this type of study, problems are identified in bridging between a task where formal methods/tools are used and another task where other (formal or not formal) methods/tools are used. Solutions are defined by developing methods/tools for combining formal methods/tools with other (formal or not formal) methods/tools. Below describes a few examples of this type of graduation studies.

- One of the studies investigated a method and tool that supports deriving specifications in VDM from requirements obtained by using KAOS [19,20].
- Another study investigated a method that derives properties to be verified by model checking from goal-oriented requirements models [21].

Extension of Methods/Tools. In this type of study, problems are identified in capabilities of existing formal methods/tools themselves regarding their own purposes (e.g., verification). Solutions are defined by developing extension of formal methods/tools. Below describes a few examples of this type of graduation studies.

- One of the studies defined refinement relationships in VDM++ as well as proof obligations so that specific kinds of refinement relationships, which have been recently dealt with in Event-B, can be explicitly modeled in VDM.
- Another study developed an Eclipse plug-in for SPIN, including an editor with auto completion, a wizard to specify LTL formula and invocation of the model checker.

5 Statistics and Discussion

In the following, attendance and completion at lecture courses on formal methods is first discussed. Problem settings in the graduation studies on formal methods are then discussed.

5.1 Attendance and Completion at Lecture Courses on FM

Table 5.1 shows how many students successfully got through each course with credit. It means the number of students who registered to the course, attended the classes, and completed reports on personal exercises and group exercises. In the table, the number of students who attended each course is also shown between parentheses (students who registered but not completed or who audited). Here only 30 students who graduated in March 2009 are counted, as previous graduates did not have the full lineup of 21 lecture courses described in Table 1.

Table 2. Completion and Attendance at Lecture Courses on Formal Methods

Series	Course	Students: completed (attended)
Model Checking	MC1	17 (21)
	MC2	12 (15)
	MC3	5 (10)
	MC4	8 (10)
Formal Specification	FS1	20 (27)
	FS2	14 (20)
	FS3	4 (5)
Implementation Techniques	IM1	6 (14)
	IM2	5 (6)

(Students in total: 30)

It would be difficult to discuss valid meanings of the figures, because there may be different kinds of reasons why students did not attend courses and why they did not complete. For example, it is often the case that a student wants to attend a course, but it is held on the day of the week when he/she has to stay at his/her company. In such a case, students often just download the slides and other materials and try to study by themselves (asking the lecturers questions if necessary). Another aspect is that **MC4** and **FS3** were held later in the program and most of the students had completed sufficient number of courses and wanted to focus on their graduation studies.

Anyway, it seems that most of the students were interested in **MC1** and **FS1**, that is, introductory courses on formal methods.

5.2 Selection of FM in Graduate Studies

28 studies were related to formal methods, out of 61 studies. Here 2 studies are not counted that mentioned some formal methods while discussing whole

pictures of development processes or educational programs to be used in the students' companies. The result shows that about half of the graduation studies used formal methods. In addition, 5 students determined to continue investigation of formal methods by getting into doctoral course of the partner graduate university. These results suggest formal methods are somewhat popular among educated top-level engineers in Japan.

5.3 Tool Selection in Graduate Studies on FM

Table 5.3 shows the number of graduation studies, on formal methods, that use each tool.

Table 3. Tools in Graduate Studies on Formal Methods

Series	Tool	Number of Studies
Model Checking	SPIN	8
	UPPAAL	2
	CSP (FDR/JCSP)	3
	Tool-Independent	1
Formal Specification	VDM	5
	Event-B	3
Implementation Techniques	JML (ESC/Java2)	1
	Java PathFinder	1
Combination	SPIN and SMV/NuSMV	1
	SPIN and Java PathFinder	1
	VDM and SPIN	1
	VDM and Event-B	1

(28 Studies on FM, out of 61)

The one classified as "Tool-Independent" is investigation of the application process of model checking. It defined roles of involved engineers and inputs/outputs exchanged between them, considering expertise required to use model checking.

Model checking was quite popular in the graduate studies. Especially, SPIN was used by many studies. It would be because students were much more familiar with SPIN as they had used it primarily (in the lecture course **MC1**).

On the other hand, VDM was also popular, which would be somewhat unique in Japan. It would be because VDM Toolbox is maintained by a Japanese company and Japanese interface has been provided. It would be also because of the well-known application case of VDM for an IC card system, which is used extensively in Japanese people's life [22]. Students who chose VDM tended to claim that they would not be able to introduce formal methods heavier than VDM, the lightweight one, to their colleagues and companies.

5.4 Topics in Graduation Studies on FM

Table 5.4 shows the number of graduation studies, on formal methods, that belong to each classification described in Section 4.

Table 4. Topics in Graduate Studies on Formal Methods

Classification	Number of Studies
Case Study	6
Domain-Specific Finer-Grained Support	11
Bridging Gaps between Different Methods/Tools	7
Extension of Methods/Tools	4

(28 Studies on FM, out of 61)

Among the studies on formal methods, Domain-Specific Support was investigated most actively. Motivations discussed in the studies suggest two points of the following.

- When students discusses with themselves how they can solve their problems in software development in their domains, by using formal methods, they often find other problems appear in applying formal methods.
- When students discusses with themselves how they can solve the problems found in applying formal methods, they often reach some initial ideas to provide methods/tools to support application of formal methods to their own domains. Although the studies are done in a few months and the initial ideas are somewhat naive, students have been able to present a core part of the ideas as well as simple case studies to show their effectiveness.

In other words, for widespread use of formal methods, domain-specific methods and tools providing finer-grained support are inevitable. Top-level engineers in the industry have sufficient abilities and motivations to provide them when they are given opportunities to do so.

Bridging Gaps between Different Methods/Tools suggests similar points. Methods and tools are inevitable that manage input to formal methods and/or output from formal methods, e.g., connecting requirements and model checking, connecting UML and formal specification, and so on.

On the other hand, it seems difficult for general engineers to extend existing methods and tools in terms of their own purposes, e.g., improving efficiency of model checkers. Actually, two of this kind of studies were simple, dealing with syntax issues (auto-completion and syntax sugars) while one essential extension was investigated by a research-oriented graduate student. The other one is investigation of the application process of model checking discussed in the previous section, which is more meaningful when deployed in the industry.

6 Conclusion

The Top SE project has been established to deliver scientific approaches in software engineering, including formal methods, to engineers in the industry. It provides lecture courses developed jointly by lecturers from the academia and from the industry, as well as opportunities of graduation studies where students identify and tackle their own problems.

This paper has reported and discussed experiences on formal methods education in the Top SE project. Students, mostly engineers in the industry, were eager to learn formal methods, both model checking and formal specification, which are used effectively and efficiently in early phases of development.

For practical use of formal methods, it is not sufficient for engineers to obtain general, fundamental knowledge of the methods and tools. It is also necessary for them to carefully consider their own contexts and determine adequate approaches to their own problems. Specifically, engineers need to choose adequate methods and tools, determine their usage strategies, and even customize or extend them for their effective and efficient use.

The Top SE project has provided educational experiences to polish this kind of abilities of engineers, through group exercises and graduation studies. Especially, experiences in graduation studies are unique, where engineers identify and tackle problems by themselves. This paper has discussed what the experiences have pointed out regarding engineers' points of view on formal methods. They consider formal methods as what are somewhat incomplete by themselves and what SHOULD be and CAN be complemented by providing domain-specific methods/tools as well as methods/tools bridging gaps between different methods/tools.

The Top SE program has established its education system that accepts about 30 students per year. After discussion given in this paper, we believe the approach of the program will promote practical use of formal methods, where not only the academia but also the industry develop practical support for effective and efficient use of formal methods.

Acknowledgments

The Top SE program is fully sponsored by the Special Coordination Fund for Promoting Science and Technology, for fostering talent in emerging research fields by MEXT, the Ministry of Education, Culture, Sports, Science and Technology, Japan. We would like to thank all the lecturers and students who have worked very hard in the Top SE program, providing the unique, interesting experiences discussed in this paper.

References

1. Honiden, S., Tahara, Y., Yoshioka, N., Taguchi, K., Washizaki, H.: Top SE: Educating Superarchitects Who Can Apply Software Engineering Tools to Practical Development in Japan. In: The 29th International Conference on Software Engineering, pp. 708–718 (2007)
2. Top SE project (NII), `http://www.topse.jp/`
3. Beckman, K., Coulter, N., Khajenoori, S., Mead, N.R.: Collaborations: Closing the industry-academia gap. IEEE Software 14(6), 49–57 (1997)
4. SPIN - formal verification, `http://spinroot.com/`
5. LTSA - Labelled Transition System Analyser, `http://www.doc.ic.ac.uk/ltsa/`

6. The SMV System, http://www.cs.cmu.edu/~modelcheck/smv.html
7. UPPAAL, http://www.uppaal.com/
8. Formal Systems (Europe) Ltd., http://www.fsel.com/
9. Communicating Sequential Processes for Java (JCSP), http://www.cs.kent.ac.uk/projects/ofa/jcsp/
10. B Method - Presentation of B Method, B Language, and formal methods, http://www.bmethod.com/
11. VDM information web site, http://www.vdmtools.jp/
12. Rodin - rigorous open development environment for complex systems, http://rodin.cs.ncl.ac.uk/
13. Saaltink, M.: The Z/EVES System. In: Till, D., Bowen, J.P., Hinchey, M.G. (eds.) ZUM 1997. LNCS, vol. 1212, pp. 72–85. Springer, Heidelberg (1997)
14. The Java Modeling Language (JML), http://www.cs.ucf.edu/~leavens/JML/
15. Java PathFinder, http://javapathfinder.sourceforge.net/ (last Access, April 2008)
16. Artho, C., Taguchi, K., Tahara, Y., Honiden, S., Tanabe, Y.: Teaching software model checking. In: Formal Methods in Computer Science Education, FORMED 2008 (2008)
17. BPMN information home, http://www.bpmn.org/
18. Kubo, A., Washizaki, H., Fukazawa, Y.: Automatic extraction and verification of page transitions in a web application. In: The 14th Asia-Pacific Software Engineering Conference, ASPEC 2007 (2007)
19. Goal-Driven Requirements Engineering: The KAOS Approach, http://www.info.ucl.ac.be/~avl/ReqEng.html
20. Nakagawa, H., Taguchi, K., Honiden, S.: Formal specification generator for KAOS: Model transformation approach to generate formal specifications from KAOS requirements models. In: The 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), pp. 531–532 (2007)
21. Ogawa, H., Kumeno, F., Honiden, S.: Model checking process with goal oriented requirements analysis. In: The 15th Asia-Pacific Software Engineering Conference (ASPEC 2008), pp. 377–384 (2008)
22. Kurita, T., Chiba, M., Nakatsugawa, Y.: Application of a Formal Specification Language in the Development of the "Mobile FeliCa" IC Chip Firmware for Embedding in Mobile Phone. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 425–429. Springer, Heidelberg (2008)

Chief Chefs of Z to Alloy: Using a Kitchen Example to Teach Alloy with Z

Sureyya Tarkan and Vibha Sazawal*

University of Maryland, Department of Computer Science,
College Park, Maryland, 20742, USA
{sureyya,vibha}@cs.umd.edu
http://www.cs.umd.edu/users/sureyya/

Abstract. Z is a well-defined and well-known specification language. Unfortunately, it takes significant expertise to use existing tools (such as theorem provers) to automatically check properties of Z specifications. Because Alloy is substantially similar to Z and the Alloy Analyzer offers a relatively simple method of model checking, we believe that Alloy should be largely employed in classes that teach Z. To this end, we present an online tutorial especially designed to help students transition from Z to Alloy. The tutorial includes both the classic Birthday Book example and a large real-world scenario based on a Kitchen Environment. Our experiences with novices studying the tutorial suggest that the tutorial helps students learn both Z and Alloy. In addition, novices can answer questions correctly about the approximately 500-line Kitchen Environment model after only a few hours of study.

Keywords: Formal Methods, Formal Specification, Z, Model Checking, Alloy.

1 Introduction

When teaching Formal Methods, it is important to choose a widely understood notation as the mathematics inherent in concepts should be unambiguous. Z [22] is a well-defined and well-known specification language which emphasizes the mathematical parts of formal definitions. A number of successful textbooks have been written using Z [9,27] suggesting that educators prefer this specification language over others available to them.

Although teaching the necessary mathematics to define software systems formally is helpful in software engineering education, mathematics alone is not enough. Verification and model checking tools are becoming increasingly popular [14]. Specifications are large and cannot be verified by manual inspection alone; thus, automatic checking of specification properties is essential. Educators are also expected to incorporate this change in their curriculum. Many software tools are introduced to facilitate this process but it is clear that model checking

* Special thanks to John C. Knight for his help on this paper.

J. Gibbons and J.N. Oliveira (Eds.): TFM 2009, LNCS 5846, pp. 72–91, 2009.
© Springer-Verlag Berlin Heidelberg 2009

is one of the winners [13]. Students who know mathematical notation may now need to learn a separate notation to write and check their models. Given the problem of lack of interest already present in Formal Methods classes [7,16,19], it is unreasonable to expect students to be highly motivated to learn various notations and be able to apply all of them later on.

As Z is one of the most popular formal notations, students should learn a technique that enables them to master tool usage besides mathematical knowledge. However, this should be accomplished without the burden of learning a new language. We believe that Alloy, in fact, provides such a functionality and is substantially similar to Z as noted by Jackson [8]:

> "The language, Alloy, is deeply rooted in Z. Like Z, it describes all structures (in space and time) with a minimal toolkit of mathematical notations, but its toolkit is even smaller and simpler than Z's."

The advantages of using Alloy in class can be summarized as follows: (i) very roughly, Alloy can be viewed as a subset of Z [2], (ii) Alloy and Z are both based on logic and set theory, (iii) Alloy, unlike many theorem provers for Z, performs fully automatic analysis without any guidance from an experienced user (as students are obviously novices), (iv) Alloy Analyzer [1] is consistently maintained by a group of researchers at MIT, (v) unlike some other model checkers, Alloy is free and can be used in the classroom, (vi) Alloy users share materials on the website forums in an online community format.

One would expect that Alloy would have been employed often in classes that teach Z. However, we see that there is a lack of examples and educational material for the transition from Z to Alloy. In particular, educators develop examples and material for either Z or Alloy yet there is insufficient courseware for Z to Alloy. Moreover, the examples that have the two versions (e.g. the Hotel locking example found in Jackson [8]) do not adequately address the relationship and differences between these languages. We suggest that this gap should be filled in with interesting examples and well-documented lesson plans.

However, there are a number of challenges in making Z to Alloy comprehensible for students. We think that employing educational theories in the preparation of these materials plays an important role. For example, the Montessori method of directing students' interests with the increasing complexity of the material [15] is helpful. Moreover, it is essential to point out both the similarities and differences between the languages. Accordingly, developing user-friendly interfaces for educational documents is also crucial to support navigation through content, for example, from reviewing discrete math and logic background, to showing formal notations, and finally, to using a tool.

As the previous paragraphs emphasize, to populate the educational materials for Z to Alloy, we suggest the use of a running example that is easy for students to understand and yet self-explanatory and comprehensive to illustrate the required processes involved in the transition. We argue that it would not be difficult for students to learn Alloy particularly if we employ our Kitchen Environment project [23]. Our contributions in this paper are twofold:

1. First of all, we present our online tutorial to teach Z to Alloy with the Kitchen Environment real-world example. In order to do that, we implement a conversion technique that simplifies this process.
2. Finally, we share our experiences with novice students using the tutorial.

The rest of the paper is organized as follows. First, we survey previous work in the area. Second, we show our full tutorial implementation. Third, we explain our case studies with novice students. Then, we discuss the results and implications of our design. Finally, we conclude with future work.

2 Related Work

The related work can be investigated from three different perspectives. First, we talk about the interactive application of Formal Methods. Next, we look at educational materials similar to ours. Finally, we mention how content-wise we employ previous work in our tutorial.

Dean [5] talks about the development of an interactive case e-study. Formal Methods materials usually reside on conventional paper and there are some difficulties in converting them to interactive electronic versions. Therefore, Dean has developed a hyperlink structure, which is a mixture of HTML files and PDF documents. One advantage is that there is no need to save paper so the material can easily be broken up into manageable sections that are interrelated. The downside is it becomes essential to match the dimensions of the computer screen. In addition, the most important aspect of the design is navigation through the document. We mostly agree with these comments but we do not make use of PDF in our implementation since the level of expected interactivity is high in our case and PDF format cannot completely fit our needs. We have used images due to the heavy mathematical content, however, there are packages like T$_T$H [24] that can convert LaTeX files easily into HTML.

Similarly, Pandora [4] is a tool developed for teaching first order natural deduction. It contains a friendly e-tutor component that provides hints, explanations, warnings, and counterexamples for corresponding student actions. This context sensitive e-tutorial has help and various other facilities for saving, loading, and printing proofs by exporting them to LaTeX. Pandora has been extensively used in class by students for their coursework and exams. When the e-tutorial is started, four tutorials of propositional exercises are available. The first consists of a fixed set of exercises and is useful in laboratory sessions. For the others, there are three levels for "easy," "medium," and "hard" each with five exercises that get randomly selected at run time by the e-tutor. Our tutorial, on the other hand, is not developed for the purpose of teaching natural deduction. Moreover, our interactive tutorial implementation is not part of such a tool, although it can easily be incorporated into one as it runs from the web.

Rosa [20] suggests the use of Piaget's theory [17] in Formal Methods education. More specifically, the paper proposes a shift of focus from the development of calculation skills to the encouragement of active participation in discrete math education. To do that, new epistemological frameworks are necessary. One such

framework is provided by the genetic epistemology theory of Piaget [18], which claims that acquiring knowledge is governed by the laws of human biological development. The paper presents an application of this theory to students, who design algorithms for solving problems. Finally, it argues that supporting instructional material is essential. In accordance, we borrow similar ideas from the pedagogical work of deBry [6] inspired by Kolb's learning theory [10]. We particularly make use of the iterative process involved in the learning of humans via employing an immediate feedback mechanism integrated into our tutorial.

Apart from such a framework, Rudall [21] develops a module from Z to SPIN. The module is divided into two parts, whose first part deals with formal specification whilst the second part deals with formal verification. The main focus of the formal specification is the Z language. The topics covered are schemas, relations, functions, and sequences; however, not every detail, e.g. the refinement and proof, of Z notation is shown. In the second part of the module, at first, temporal logic is introduced so that the Promela [25] modeling language and the SPIN [26] model checker can be taught. Promela is used to model complex process interactions and SPIN is used to verify these models in the lab sessions. The paper indicates that the examples and theory are worked through on the board in class mainly with student interaction but for practice, students are expected to follow the tutorials using software tools in the labs on their own. It is believed that to occupy the interest of students, a broad base course is more useful than a narrow focus. Although the author clearly identifies the reason for the selection of Z, she admits that the Promela language is chosen subjectively. We depart from this study in the following ways. We make use of the Alloy modeling language and tool as it is very similar to the Z notation. Furthermore, we embed the in-class interaction within the tutorial.

Many studies reflect on the fact that real-life examples make learning of Formal Methods more interesting for the students [3,11,12,19]. Brakman [3] uses a project about a Bluetooth communication protocol to increase the "fun-factor," and henceforth, the attendance rates of the classes. Similarly, to motivate students with hands-on experience, Larsen [11] assigns students interesting class projects to work on. Moreover, Lightfoot [12] proposes an interesting group competition based on the vote-recording software of the Eurovision Song Contest. Finally, Reed [19] stresses that the use of small, simple, and practical examples (Fibonacci numbers, integer division, invariants, the Needham-Shroeder protocol) is helpful for novice programmers in the early stages. Inspired by the successes of these prior studies, in our tutorial, we make use of first a simple well-known example, the Birthday Book [22], and later a comprehensive but everyday-life example that we developed, the Kitchen Environment.

3 Z to Alloy Tutorial

In this section, we describe the details of our online interactive tutorial, which is publicly available at our website [28].

3.1 Tutorial Content

The content is prepared to teach the concept of "model checking formal specifications" to novice students. It is organized as follows:

– **Introduction.** This section covers the motivation for what formal methods are, what they are used for, and some examples that illustrate disastrous cases when they are not properly applied in practice.

– **Birthday Book.** This simple example is presented in detail with two separate implementations (one for the formal Z specification and the other for the Alloy model) to give the mathematical and logic background as well as to explain the details of Z and Alloy languages.

– **Alloy Analyzer.** The tool is reviewed to show what its capabilities are.

– **Kitchen Environment.** This comprehensive example is developed using a conversion technique from the Z specification to the Alloy model. This model is pretty long compared to the specifications that are covered in textbooks and therefore successfully relates the concept to real software specifications used in industry.

– **Quiz.** A quiz is given to assess the student's understanding of the material. The questions ask the student to complete and fix an Address Book model [8]. This is similar to the Birthday Book with a difference that it saves people's addresses.

We explain the Birthday Book and Kitchen Environment examples in more detail in the sections that follow.

The Birthday Book. Our Z specification of the Birthday Book is adapted from Spivey [22] and our Alloy model is directly taken from the Alloy distribution of the sample models [1]. The Birthday Book records people's birthdays (a name and a date for each person), and places a reminder when the appropriate day comes. It can be populated and depopulated, and there is a search option.

The Birthday Book specification includes the type declarations, the data objects, the state space, the initial state, and the operations – add, delete, find, and remind – as schemas. Within this specification, we review concepts from set theory and more specifically, the operators and symbols that are commonly used in the Z notation.

The corresponding Alloy model is then used to automatically check the subsequent assertions:

1. Adding a birth date to the book indeed works.
2. Deletion is an undo of addition.

To explain the Alloy model, we introduce the Alloy notation, i.e. modules, signatures, atoms, relations, relational product operators, multiplicity markings, predicates, assertions, and checks, as well as how to interpret the output.

The Kitchen Environment. This example is adapted from Tarkan [23]. The task is to simulate the actions of a kitchen chef to direct other cook(s) in the preparation of a dish (given the recipe). The primitive object types are `cook`,

ingredient, kitchen_item, and measurement and the built-in functions are Bake, Clean, Cut, Knead, Mix, Preheat, and Put.

The complications in the specification are identified in the following manner. Multiple cooks can function concurrently, some kitchen items work autonomously, multiple ingredients get composed to make others, ingredients change their states from Raw to Baked, Cut, Kneaded, Mixed, or Processed. To this end, an event-driven programming approach is suggested. For this purpose, we first write an approximate Z specification and then convert it into a full Alloy model that enables the student to perform automatic analysis. We do not develop a full Z specification since it is going to be modified to meet the Alloy language requirements. Lastly, we identify the following list of assertions as crucial for our system.

1. Item is the same tool after getting Cleaned.
2. Preheat always precedes Bake.
3. Preheat is the only way to heat an item.
4. A HeatedItem should always be available for use to protect against fire.
5. A UsedIngredient never becomes available again.
6. A newly created AvailableIngredient[1] is composed of only UsedIngredient(s), which were available prior to the creation of this ingredient.
7. Raw ingredients are never a composition of other ingredients.
8. All AvailableIngredients that are not Raw are compositions of other ingredients.

Given the problem statement, we show how to convert sets in the Z specification to signatures in the Alloy model. We present the sets of cooks, kitchen items, ingredient names, measurements, and event identifiers in the Z notation. Apart from these, an ingredient is represented with a schema that has fields for its name, amount, and its constituents that change over time. The subsets of ingredients are also defined for Baked, Cut, Kneaded, Mixed, Processed, and Raw. Direction saves all the events called so far and the arguments to these calls. We first convert the sets for cook and kitchen item to signatures. At this point, we tell the student what the limitations of model checking are. For instance, some sets that do not affect the state are not declared in the Alloy model to prevent extensive memory usage. We also explain the limited support for integers in Alloy as they are infinite and introduce our Time signature that is represented with an integer in Z. We introduce built-in modules such as ordering supported by Alloy. We show them the object-oriented nature of Alloy as opposed to Z with the Baked, Cut, Kneaded, Mixed, Processed, and Raw disjoint subsets of the ingredient type. In this manner, we clearly point out the differences and similarities between these approaches.

Next, we show the state specification in Fig. 1. We again first talk about the Z schema and reflect on our decisions throughout its definition, especially the relationship to time is stressed and how this is employed in the form of relations within the Z version. We finally talk about the invariants and their importance

[1] This ingredient cannot be raw but can only be in one of the following states: baked, cut, kneaded, mixed, or processed.

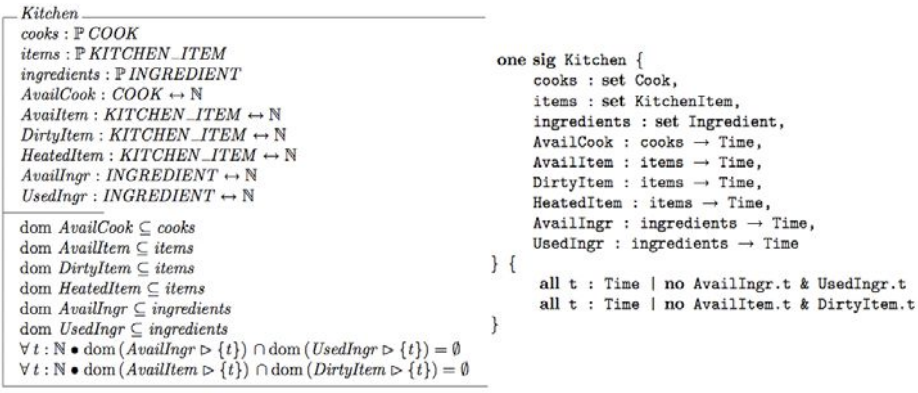
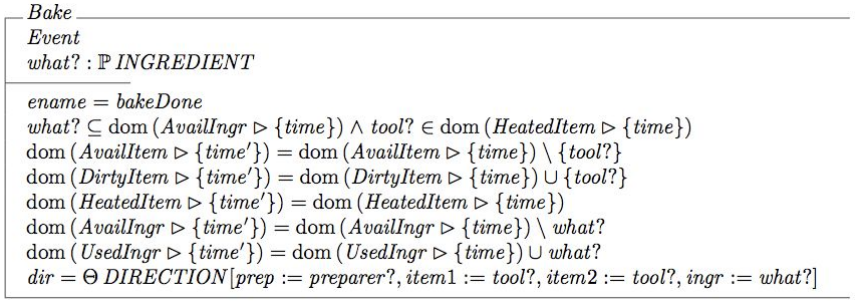


Fig. 1. Kitchen schema and signature. Kitchen acts as a database in the system. There are cooks, items, and ingredients in this world. Kitchen records available cooks, items, and ingredients, dirty and heated items as well as used ingredients. All of these entities depend on time. There are two important invariants of the system: an item cannot be both available and dirty at the same time and analogously available and used ingredients are disjoint at all times.

to our system. With this, we complete our Z schema and start converting it into an Alloy model. We immediately comment on Alloy's capabilities for the multiplicity of signatures. After simply saying that the body is very similar, we note the fact statement that is appended to the signature. We introduce how facts can be used to specify the invariants. The final step is to call to the student's attention the absence of the `Timer` schema in the Alloy model. Because a timer is a mechanism that keeps track of incremental properties like time and event count, after explaining all the details related to this schema, we inform the student that such entities will be handled with an equivalent but space-saving "trick" in Alloy.

As we are finished with the state specification, we move to the state initialization for the Z version and talk about the details in a similar fashion as the previous one. As soon as we start writing the Alloy model, we highlight the difference between Z, which represents operations with schemas, and Alloy, which uses a distinct notation for the dynamism, i.e. predicates. Afterwards, we proceed to the conversion.

One of the complications in this example is the need for events that are the cause of concurrency. We split events into the first call and the announcement of completion. Therefore, we have events for `Bake`, `Clean`, `Cut`, `Knead`, `Mix`, `Preheat`, and `Put` but also `BakeDone`, `CleanDone`, `CutDone`, `KneadDone`, `MixDone`, `PreheatDone`, and `PutDone`. When we are done with the state space and initialization, we convert the events defined as schemas in the Z notation to predicates in the Alloy version. We first start with `Event` and `EventDone` schemas that will be reused in the specific event calls and thus, contain definitions common to all of them. We review all of their definitions. Moreover, as



Bake

Event

$what? : \mathbb{P}\, INGREDIENT$

$ename = bakeDone$

$what? \subseteq \mathrm{dom}\,(AvailIngr \rhd \{time\}) \land tool? \in \mathrm{dom}\,(HeatedItem \rhd \{time\})$

$\mathrm{dom}\,(AvailItem \rhd \{time'\}) = \mathrm{dom}\,(AvailItem \rhd \{time\}) \setminus \{tool?\}$

$\mathrm{dom}\,(DirtyItem \rhd \{time'\}) = \mathrm{dom}\,(DirtyItem \rhd \{time\}) \cup \{tool?\}$

$\mathrm{dom}\,(HeatedItem \rhd \{time'\}) = \mathrm{dom}\,(HeatedItem \rhd \{time\})$

$\mathrm{dom}\,(AvailIngr \rhd \{time'\}) = \mathrm{dom}\,(AvailIngr \rhd \{time\}) \setminus what?$

$\mathrm{dom}\,(UsedIngr \rhd \{time'\}) = \mathrm{dom}\,(UsedIngr \rhd \{time\}) \cup what?$

$dir = \Theta\, DIRECTION[prep := preparer?, item1 := tool?, item2 := tool?, ingr := what?]$

```
pred Bake[t, t' : Time, preparer : Cook,
        tool : KitchenItem, what : set Ingredient] {
    preparer in Kitchen.AvailCook.t
    tool in Kitchen.HeatedItem.t
    tool in Kitchen.AvailItem.t
    some what and what in Kitchen.AvailIngr.t
    Kitchen.AvailCook.t' = Kitchen.AvailCook.t - preparer
    Kitchen.AvailItem.t' = Kitchen.AvailItem.t - tool
    Kitchen.DirtyItem.t' = Kitchen.DirtyItem.t + tool
    Kitchen.HeatedItem.t' = Kitchen.HeatedItem.t
    Kitchen.AvailIngr.t' = Kitchen.AvailIngr.t - what
    Kitchen.UsedIngr.t' = Kitchen.UsedIngr.t + what
    noComposedChangeExcept[t, t', none]
}
```

Fig. 2. Bake schema and predicate. Bake expects available ingredients and an available and heated item. At the end of the call, the ingredients are used and the tool becomes a dirty item.

the cook operates separately from the tools, `CookDone` is introduced in the same manner. The simple predicate `noComposedChangeExcept` is introduced before the conversion process as it is included in many predicate bodies. We refer to the `Directions` schema to explain the unavailable state of the preparer in the Z version. We again draw attention to the differences between two versions when we include our pre-specified predicate in others. At last, we defer, to later, the discussion about time ordering using a factual statement.

When the general schemas are ready, we specify more concrete events. The first one is `Bake` in Fig. 2. As `Bake` schema is introduced, we underline the differences and similarities with its model. The first one is the `some` multiplicity marking on the ingredient. We say that some important facts like these are not imposed in the Z specification as it serves as an initial draft for our Alloy model. However, the Alloy model is iteratively refined after analyzing the results from the previous steps and hence, is more accurate. When we get to `BakeDone` conversion, one key note is its name, which is `BCKMPDone` in the Alloy version. We say that all `BakeDone`, `CutDone`, `KneadDone`, `MixDone`, and `PutDone` affect the state similarly so they are combined in the implementation. Consequently, those differences related to generalizing it are listed. More specifically, we say that we can only assert that the type of the ingredient produced cannot be

```
fact DoneAfterEvent {
    all t'' : Time - TO/last[], c : Cook | some i, k : KitchenItem, w : Ingredient |
        let t' = TO/prev[t''], t = TO/prev[t'], t''' = TO/next[t''] |
            CookDone[t'', t''', c] ⇒ (Bake[t, t', c, i, w] or Clean[t, t', c]
            or CKMP[t, t', c, i, k, w]  or Preheat[t, t', c, i])
}
```

Fig. 3. DoneAfterEvent fact. CookDone event only takes place after one of Bake, Clean, CKMP, or Preheat events.

Raw. Moreover, HeatedItem is not handled here because different done events act upon it differently. Clean event does not need further explanation as it is very similar to Bake. However, with Cut event we see that in the conversion we combine Cut, Knead, Mix, and Put together under CKMP as they all do same things, too. We do not further discuss Knead, Mix, and Put schemas as they are same with Cut. Our final schema is Preheat. We say that the input to this schema is of type measurement and there is no need to model that in the Alloy as it does not change the state. PreheatDone is also parallel with BCKMPDone.

Later, the schemas related to the event handler in the Z specification are represented with fact statements in the Alloy model. For this, we first recall our deferred discussion on the fact statements that were supposed to constrain the system. We initially write a fact statement called Traces that takes care of the ordering of the states. We summarize that we cannot advance time by one unit at each tick and to circumvent this, we jump from the end time of one task directly to the start time of the next task. We review the body of the fact statement and any new notation that is encountered for the first time. We conclude with the remark that there is still no event handling and any event can take place at any point in time. In that regard, we formalize a guarded implementation. This concept is presented as follows. An event happens only when those events that must precede it already have taken place and those that must succeed it are guaranteed to take place. Thus, instead of depending on the time increment, the system jumps between events. As an example, we show DoneAfterEvent in Fig. 3, which stipulates that CookDone comes after Bake, Clean, CKMP, or Preheat. Similarly, BakeBeforeDone requires that every Bake implies there is a BCKMPDone event that will happen and a CookDone event succeeds it. DoneAfterBCKMP states that BCMKPDone follows Bake or CKMP but it introduces some more constraints, which were omitted beforehand in the predicate definitions, about the relationship between the first and second events. It is pointed out that in the Z implementation, we prepared an extra schema called Directions to record each call and Timer schema keeps track of all the event handling but in Alloy, we use such facts to fill in these details. CleanBeforeDone, PreheatBeforeDone, DoneAfterCP, and CKMPBeforeDone facts are written analogously.

Finally, to convince the student that our model actually satisfies our assertions, we carry out the final step of writing the verbal assertions above formally and checking in the Alloy modeling language. We first show DirtyAndCle- anSame and explain why the check command needs more time steps than any other

instances of sets for this assertion. We emphasize the distinction between under-constraining and over-constraining and that if the range is not assigned properly, the Analyzer will fail to find counterexamples. We note that it is better to be on the safe side and check with the maximum possible range without burdening the tool too much. Subsequently, `PreheatBeforeBake`, `HeatUnheated`, `NoFire`, `UsedNeverAvailable`, `AvailableOnlyUsed`, `RawNonComposed`, and `NonRawComposed` are developed as are the rest of the assertions.

Although the conversion seems straightforward, some complications arise and these are addressed and documented in the tutorial. In this way, we expect the students to be able to understand both the similarities and differences between the two versions so that when they work on their own specifications and models, they will have acquired the necessary background. At the end of the tutorial, we encourage the students to think and write more assertions that makes them feel satisfied with their expectations. Our final Alloy model is fairly long (≈ 500 lines) but we clearly state that once a rough Z specification is available, Alloy development is not very cumbersome.

3.2 Implementation

The implementation of the online tutorial consisted of two phases.

Webpage Design. As can be seen from Fig. 4, we decided to split the tutorial into the following sections, which are shown on the upper left-hand side as a list box. The numbers in the parentheses below show the number of questions in each section, which will be covered in the subsequent paragraph.

(1) Introduction
(2) Birthday Book Specification (20)
(3) Alloy Analyzer Walkthrough
(4) Birthday Book Model (26)
(5) Kitchen Environment
(6) Sets as Signatures (8)
(7) State Specification & Initialization (11)
(8) Events as Predicates (23)
(9) Handler as Facts (7)
(10) Assertions (9)
(11) Conclusion
(12) Questions (4)

The interactivity of the pages is supported as follows. Although Section 3.1 presented as if the tutorial expects the students to be passive readers of the tutorial, this is indeed not the case in our tutorial. According to Kolb's theory, the student takes some information, which gets combined with the existent knowledge of the student. With questions that provoke further reasoning, the educator can immediately act upon the student's understanding to correct the misunderstandings. The reaction of the educator also becomes part of the student's prior knowledge so the educator asks more questions to steer the student's

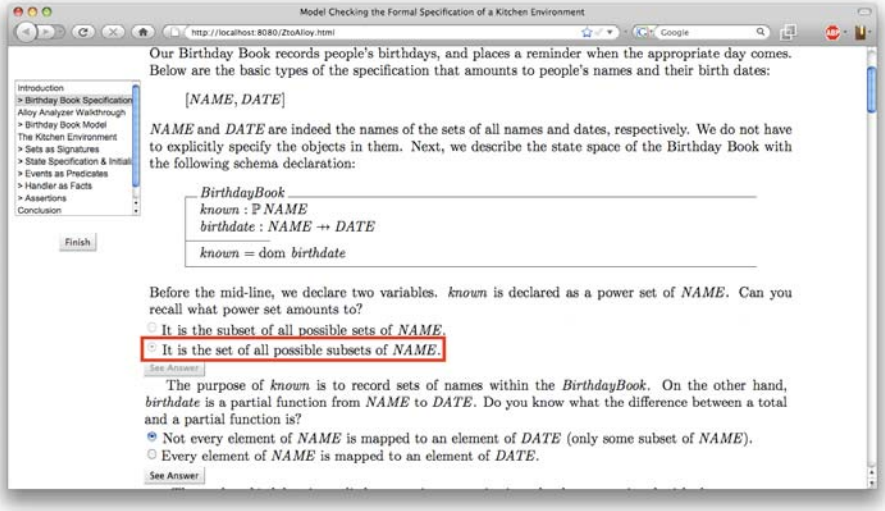


Fig. 4. The Online Interactive Tutorial. The webpage contains teaching material as well as some questions to trigger student's learning. The radio buttons are placed for multiple-choice questions while *See Answer* buttons are used for submitting student's answer and seeing the expected answer. When revealed, the correct answer is highlighted with a red border and the interactivity is disabled. The sections are listed on the left-side panel to enable easy navigation within the document besides the conventional "Prev" and "Next" buttons located at the end of each page. The sections whose names start with a ">" symbol are interactive. When all of the questions within a section are answered, this symbol disappears from the name of that section so that the student knows they have completed the section. The *Finish* button yields the end of the tutorial.

learning. With this theory, we decided to implement multiple-choice questions in the tutorial. Most of these questions would ask the student to think beyond what is given rather than passively reading the text. The students are given the option to reveal the correct answer once they are satisfied with their answer. Consequently, the student has the chance to correct their mistakes further on. Apart from this interactive content, the last section, Questions, is organized as a quiz whose questions are free-fill. Every question contains a line that must be filled in and submitted by the student.

Finally, we embedded two surveys within the tutorial. The pre-survey asks 17 questions on the student's background with Formal Methods and the post-survey with 30 questions asks for feedback. These surveys are prepared using the Likert scale from 1 ("Strongly Disagree") to 7 ("Strongly Agree"), and include an option for the "Don't Know" case. There are also some open-ended questions within these surveys.

Server to Database Connection. Because we wanted to make the tutorial accessible from the Internet through students' computers, we implemented a server component that records client answers in a database and sends the correct answers to the client. The students should be given an identifier and a password by the instructor to access the tutorial. Each student is assigned a user identifier of 5 characters, e.g. 'usr09.'

4 Case Studies with Novice Students

In this section, we report the experiences of our novice students with the tutorial.

4.1 Task and Procedure

In a single session in our lab, the students were asked to read the tutorial and to answer all of the multiple-choice and free-fill type questions on the tutorial. They were also asked to fill out the two surveys. Consequently, they responded to 104 multiple-choice, 4 free-fill, and 47 survey questions in total. Furthermore, the online interface and its functionality were explained before the session. The students were allowed to ask questions during the entire session and also take a break at any time. We used a single computer and the same web browser for all of the sessions. Each student spent between 1 to 3 hours on the tutorial and we accepted verbal comments from the students during or after the tutorial to receive their feedback.

4.2 Student Background

In total we had 8 students (2 females and 6 males) take our tutorial. They were aged between 20 to 30 with an average of 25.75. All of them were Computer Science students with one who had a minor in Mathematics. One of them was a sophomore, two were Master's students, and five were Ph.D. students. They strongly agreed that they have experience with small- and medium-sized software projects (with an average of 6.88/7 and 6.25/7, respectively). They also had an above average (5.38/7) discrete math and logic background. They moderately disagreed that they have strong formal methods background (mean = 3/7 excluding one student who answered they did not know the answer) and they were neutrally interested in formal methods (with a mean of 3.6/7 excluding three students who replied that they don't know the answer). Four of the students who answered the questions thought that formal methods are useful (mean is 4.75/7) and that they are difficult to learn and understand (average is 4/7). Out of the five people who answered the question, the students were willing to make use of formal methods in the future (mean = 5/7). Six students responded that they have seen formal specifications written in English (average = 4.33/7). Seven students strongly agreed that they have not had much experience with formal methods with an average of 6.14/7.

On the open ended questions, the students replied that they use unit testing (e.g. JUnit), smoke testing, nightly builds, code review, trial-and-error, debug printing, incremental development, debuggers (e.g. gdb), running the code, writing test cases, and user studies to detect errors in their code. For their class background, they mentioned an undergraduate level discrete math course or a graduate level programming language or software engineering course. Four people did not report any class background. Only one person indicated that he or she had used SPIN as a formal software verification tool in the past.

4.3 Results

In this section, we report the results from two points of view. First, we present the results in terms of the tutorial's comprehensibility. Next, we report them based on each student's performance.

Tutorial Questions. Here, we talk about the students' answers for the multiple-choice questions. All of these questions would require the student to decide between two answers. At first, we would like to further split our questions into two parts: (i) Birthday Book, and (ii) Kitchen Environment questions.

For the Birthday Book example, there were 46 questions in total. As we introduced the Z specification of the Birthday Book, some initial questions were directed to review the mathematical background of students, but later as students gradually improved their understanding of the notation, more and more questions became related to the specifics of the Birthday Book example. A similar approach was taken for the Birthday Book model. We started asking questions related to the mathematical interpretations of Alloy and as we moved further into the details of the model with a better understanding of the student, we elaborated on the questions. Overall, students did well in the Birthday Book example and we were satisfied that they understood the material. Here we report all of the eight students' results. For the Z specification of the Birthday Book, they answered 17 to 19 questions correctly (on average 18.25/20). The Birthday Book model was well-received, as well. Students gave between 20 and 24 correct answers (mean = 22.25/26) to the questions. Overall for the Birthday Book, their correct answers ranged between 38 and 43, with a mean of 40.5/46.

The Kitchen Environment example contained a total of 58 questions. For this part of the tutorial, we did not ask for any mathematical notation but instead we initially asked questions about the student's recollection of the concepts taught in the Birthday Book. As this example was developed side-by-side with the Z and Alloy versions, we distinguish them according to our conversion technique. When converting sets to signatures, all eight students got 5.88/8 on average the questions right (a minimum of 4 and a maximum of 7). However, when they got to the State Specification & Initialization section of the tutorial, one of the students dropped out of the session because they have not recently studied mathematics and would need to go over the material more than once to truly comprehend it. Therefore, seven students remained to report on this section. Students answered 7 to 10 questions right with the mean being 9/11. By the

end of the section that converts events to predicates, another student decided to stop continuing the tutorial because they were too tired that day. However, they had been able to answer 19 of the questions and got 9 of them right. Excluding this person, six students gave 14 to 21 correct answers on this section and their average was 16.5/23. For the rest of the Kitchen Environment example, we only provide the results of the six remaining students. Their correct answers on the conversion into fact statements was between 4 and 7 with a mean of 5.67/7. These students had 6 to 8 correct answers (with a 7.17/9 average) for the last section that showed Alloy assertions. Excluding the students who could not finish this example, on average, there were 45/58 correct answers (51 at most and 41 the least).

For those six students who were able to complete both of the examples successfully, out of all the multiple-choice questions, they had a mean of 85.5/104 correct answers (93 was the maximum while 79 was the minimum number).

Assessment. The assessment was based on the 4 quiz questions. The questions asked the student to type one line of Alloy code to complete and correct a different model than the ones that are presented throughout the tutorial. The first question asked the student to write the body of a predicate, the second one asked the student to write the body of an assertion while the next one asked for its check statement. The final question tested the student's ability to modify the code so that it excludes ill-formed cases.

As noted in the previous section, because of the students who stopped early, we have only six students whose results we can present. Moreover, some students missed this section as it was after the Conclusion section within the tutorial and they went directly into the post-survey questions. Thus, there were some blank answers. The questions were evaluated based on legitimate rationale rather than exact syntax. Thus, minor syntax errors were disregarded. Based on this grading key, our students got between 0 and 3 answers right with a mean of 2/4.

We also briefly mention that overall, on both the tutorial and the quiz, the correct answers ranged between 80 to 96 (with a mean of 87.5/108) for those six students who finished it.

Student Performance. Instead of reporting each student's performance, we classified the students based on their performance on each section into three clusters as can be seen from Table 1. More specifically, we identified three students whose performance was the best on the tutorial. Additionally, three students did well in answering the questions but not as well as the best. Apart from these, as previously mentioned, two students finished the tutorial with an incomplete status.

As the standard deviations are small, it is obvious that those clustered together have performed similarly. To do a comparison between three different student performances, good students performed slightly worse than the best students in the Birthday Book. On the Kitchen Environment example, this trend seemed to continue. On the Questions section, the correct answers were equal implying that they all learned the concepts. The totals indicate that above 90

Table 1. The summary of the results for those students who achieved best performance, and who did well, and those who could not complete the tutorial

	Best Performance		Good Performance		Incomplete Tutorial	
	Mean	Std dev	Mean	Std dev	Mean	Std dev
Birthday Book specification	18.67	0.58	17.33	0.58	19	0
Birthday Book model	23.33	1.15	21.67	0.58	21.5	2.12
Birthday Book (total)	42	1	39	1	40.5	2.12
Sets as signatures	6.67	0.58	6	0	4.5	0.71
State specification & initialization	9.67	0.58	9	1	7	N/A
Events as predicates	18	2.65	15	1	9	N/A
Handler as facts	6.33	0.58	5	1	N/A	–
Assertions	7.33	1.15	7	1	N/A	–
Kitchen Environment (total)	48	2.65	42	1	12.5	–
Birthday Book & Kitchen Environment	90	2.65	81	2	53	10.6
Questions	2	1.73	2	1	N/A	N/A
Total	92	3.61	83	3	53	8.49

is the best, while above 80 is good. An interesting result comes out for the incomplete group. They did better in the Birthday Book example than the good students, however, because they were not able to continue, we do not know how they would have performed in the Kitchen Environment example. We believe that they could in fact perform better again giving us better results for our tutorial.

4.4 Student Feedback

The post-survey asked for each student's feedback. However, we have only records for seven of the students because one quit before filling it out. Also, some students answered some of the questions as they don't know the answer and these were not counted. All the numbers reported in this section range from 1 to 7 (the Likert scale).

First of all, we had questions related to Formal Methods, i.e. the appreciation of their benefits, confidence in proficiency, credence to automatic verification tools. For these, students moderately agreed and gave 4.5, 3.29, and 4.57 points on average, respectively. Next, we asked questions about the student's opinion about formal specification (their comfort with the Z language and ability in writing Z specifications) and students were neutral (with means of 4.86 and 3.14, respectively). We also directed questions on model checking – learning of model checking, usefulness, competency in Alloy, ease-of-use for the Alloy Analyzer. Students moderately appreciated model checking (with means for learning as 5, usefulness as 4.57, and ease-of-use as 3.67) but they did not feel competent enough yet (mean = 2.86).

Apart from these, there were questions on the Z to Alloy transition. More specifically, we asked students about the ease of learning Alloy after Z, whether Alloy is similar to Z, their ability to convert Z to Alloy, the straightforwardness of the Z to Alloy conversion, their preference to write Z beforehand, and the favorability of completely transitioning to Alloy. Students strongly agreed with our statements on the ease of transitioning from Z to Alloy with the tutorial (receiving 4.57, 5.29, 4.43, 4.86, 4.33, and 5.8 averages, respectively).

Finally, there were questions about the teaching material. We asked whether it was a meaningful (5), efficient and effective (5.86), and motivating material (5.14), whether its organization facilitated learning (4.71), whether it included enough math and logic background (5.67), the appropriateness of each example's level of complexity (4.29) and mental effort (4.71), and interest (4). The students responded that they agree with these statements as shown with the means given in the parentheses. Moreover, some questions asked if they knew similar examples as the Birthday Book (2), the comprehensiveness of the Kitchen Environment example (6.14) and its presentation's level of detail (5), and the success of the conversion (5.14). As the averages between the parentheses state, students agreed that Kitchen Environment example was good even though they have not been exposed to simple examples like the Birthday Book. Finally, we directed questions at the student's ability to design their own specifications and model check them (3.57), their satisfaction with the tutorial (4.71). The averages indicate that the students gained some moderate capability at model checking and that they were satisfied.

At the end of the post-survey, we expected them to share what they liked/disliked and their opinions on how to improve the tutorial with a last open-ended question. A lot of ideas were related to the web layout of the tutorial.

– "i had to do too much scrolling."
– "Questions about figures should come after the figures themselves."
– "1. make the listbox larger 2. put prev next below the listbox. . . "
– "the figures for the questions were sometimes far beneath the questions themselves."
– "Referring to multiple models on other pages was distracting – especially since the page did not either reset the scroll bar or return to where you were when you previously viewed that page."

Moreover, some students stated both in their verbal comments and in the post-survey that they wanted visualizations (pictures, diagrams, and tables).

– "I would have liked to see a better comparison of English/Z/Alloy – maybe a few figures where they were compared and explained step by step next to each other for full effect. . . I think a more succinct table of some of the Alloy examples would have helped me understand better."

Some of the students said that they needed more intermediate examples.

– "kitchen example was very complex and maybe a too big big step from the very simple birthday book."

– "The kitchen example lost me halfway through. The birthday book was simple and made sense; the kitchen example introduced too much too quickly. It was well written and presented, but was too difficult for me to quickly grasp. More intermediate examples should have been presented."

There were cases when the students thought that one tutorial was not enough for them.

– "I felt the need of a some math and logic background. Moreover, I don't think it was easy to grasp all the definitions in the first go."

Some students found the (English) language in the tutorial hard to understand.

– "The language in the tutorial is hard to understand in several places."
– "Some of the Alloy conventions were hard to understand."
– "I felt that the question choices often stated two sides of the same thing – perhaps I did not quite understand the intent of the statements. While the step-by-step conversion process was nice the grammatical structure of some of the sentences throughout this process seemed confusing to me."

Although they stated that they liked the use of everyday examples, they complained that just examples are not enough.

– "trying to explain the importance of formal methods using simple day-to-day examples was quite motivating."
– "I feel that this tutorial brought about an appreciation for model-checking in a roundabout way; rather than coherently explaining the process it seemed to jump immediately into examples and never really get into a broad understanding of the topic."

In their verbal comments, one student stated that although he/she thinks it is certainly useful to perform model checking, it is expensive and onerous to write for his/her own projects. Similarly, another student commented that he/she does not intend to make use of model checking, because as researchers, they are satisfied with software that is "good enough." However, both of them agreed that they clearly appreciate the necessity in critical aspects (security, concurrency, etc.) of a big software system.

5 Discussion

In this section, we share what we learned from our studies and what could have been improved and in what way.

Generally, speaking we are encouraged with the results. Given the weak background of the students on formal methods and their lack of interest in the formal methods topics, the results show that the tutorial brought some appreciation for formal specification, model checking, and verification tools. Looking at the survey responses, we are also pleased that the students thought Alloy is similar

to Z and that they appreciated our conversion method in compliance with our expectations. However, because none of them had prior Z background, they were inclined to like Alloy more than Z as it enables them to directly jump to programming. Our Kitchen Environment example was comprehensive but compared to the simple Birthday Book, the student performance did not show a huge drop. This can be seen from the percentages of correct answer means for the Kitchen Environment (77.5%) and the Birthday Book (88%). It is clear that students were able to comprehend even such a complicated example.

What we learned from this study is that students do seem to appreciate paper specifications but they want to be able to practice and correct their designs using automatic tools. In that respect, they believe that Z and Alloy are similar and can be taught together. However, there needs to be some appropriate material in this field. The available textbooks teach Z or Alloy but not both and most of them only use toy examples of 10 to 50 lines. We tried to approach this problem by making use of a conversion technique to facilitate Z to Alloy transition with a comprehensive example; however, students needed well-documented intermediary examples, too. Compared to most of the textbooks we encountered, our complex example was 500 lines. The students surely felt that this was a huge gap that needs to be filled in by the instructors. Finally, they liked the idea of real-world examples but requested pictures to make them more appealing. Educators should take this into account when designing their materials.

One downside of the web component was that it was offered as one large tutorial instead of being broken up over two or more lectures. This was the reason for some students feeling too overwhelmed to finish it in a single session. There were also some problems with the online version of the tutorial that students candidly addressed in their complaints about the interface. Even with such problems, students did not criticize the content of the tutorial. This suggests that they were satisfied that they learned something new.

Moreover, our student pool was not representative of real software developers. We mostly conducted studies with graduate students, who do not often implement extensive software projects. Moreover, these students had not studied discrete math and logic in several years, and their interests were specialized in other areas of the computer science. This gave rise to issues in recalling the concepts. Since the studies were conducted at the end of the semester and during the final exam week, we could not attract many undergraduate students to participate. The long duration of the study and lack of compensation deterred some students who were initially interested in the study. Some of the students who took our tutorial did not take the questions on the quiz seriously enough nor did they complete all sections. However, looking at the in-tutorial answers, we are satisfied that participants did put enough effort into reading and answering the questions as none of them had correct answer rates below 70%. In terms of time, it took more than an hour for each student to complete the tutorial. Also, due to time limitations, we could not run enough pilot studies to correct the implementation errors and thus, some answers got unrecorded.

Finally, we allowed students to have access to the entire content throughout the session and did not save times along with the answers. Had this been implemented in our tutorial, we think we could have revealed some common patterns in the learning of Z to Alloy that could be useful for instructors.

6 Conclusion

In this paper, we first presented our online tutorial to teach Alloy with Z using the Kitchen Environment real-world example. In order to do that, we presented our conversion method that is expected to simplify this process. We also shared our experiences with novice students using our tutorial.

According to our findings, we recommend that educators who teach the Z language also focus on the Alloy tool in their classes. In addition, it is necessary to provide well-explained and interesting intermediate-level examples for the Z to Alloy transition. Our tutorial provides both a concrete Z-to-Alloy transition process and the intermediate-level Kitchen Environment example. Initial experience with novices suggests that students are able to learn Z and Alloy and answer questions about an intermediate-sized example after only a couple hours of study. Interested readers may access this tutorial through our website [28].

References

1. Alloy Analyzer 4, `http://alloy.mit.edu/alloy4/` (2009-07-15)
2. Alloy FAQ, `http://alloy.mit.edu/faq.php` (2009-07-15)
3. Brakman, H., Driessen, V., Kavuma, J., Bijvank, L.N., Vermolen, S.: Supporting Formal Method Teaching with Real-Life Protocols. In: Formal Methods in the Teaching Lab: Examples, Cases, Assignments and Projects Enhancing Formal Methods Education, pp. 59–68. McMaster University, Canada (2006)
4. Broda, K., Ma, J., Sinnadurai, G., Summers, A.: Friendly e-tutor for Natural Deduction. In: TFM: Practice and Experience. BCS-FACS, London (2006)
5. Dean, N.: Development of an Interactive Case e-Study. In: TFM: Practice and Experience Workshop, pp. 13–20. BCS-FACS, Oxford Brookes University (2003)
6. deBry, R.: Learning exercises for the rest of the brain. J. Comput. Small Coll. 20(1), 291–296 (2004)
7. Duke, R., Miller, T., Strooper, P.: Integrating Formal Specification and Software Verification and Validation. In: Dean, C.N., Boute, R.T. (eds.) TFM 2004. LNCS, vol. 3294, pp. 124–139. Springer, Heidelberg (2004)
8. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press, Cambridge (2006)
9. Jonathan, J.: The way of Z: Practical programming with formal methods. Cambridge University Press, NY (1996)
10. Kolb, D.A.: Experiential learning: Experience as the source of learning and development. Prentice-Hall, Englewood Cliffs (1984)
11. Larsen, P.G.: Two courses on VDM++ for Embedded Systems: Learning by Doing. In: Formal Methods in the Teaching Lab: Examples, Cases, Assignments and Projects Enhancing Formal Methods Education, Canada, pp. 21–26 (2006)

12. Lightfoot, D.: Voici les votes! – formal specification as light entertainment: An example of audience participation in developing a specification. In: TFM: Practice and Experience Workshop, pp. 71–75. BCS-FACS, Oxford Brookes U. (2003)
13. Mandrioli, D.: Advertising Formal Methods and Organizing Their Teaching: *Yes, but...* In: Dean, C.N., Boute, R.T. (eds.) TFM 2004. LNCS, vol. 3294, pp. 214–224. Springer, Heidelberg (2004)
14. Martin, J.M.: Teaching Formal Methods: An Industrial Perspective. In: TFM: Practice and Experience Workshop, pp. 35–39. BCS-FACS, Oxford Brookes U. (2003)
15. Montessori, M.: The Montessori Method, Schocken (1988)
16. Pepper, P.: Distributed Teaching of Formal Methods. In: Dean, C.N., Boute, R.T. (eds.) TFM 2004. LNCS, vol. 3294, pp. 140–152. Springer, Heidelberg (2004)
17. Piaget, J., Mays, W., Beth, E.W.: Mathematical Epistemology and Psychology. D. Reidel Publishing Company, Dordrecht-Netherlands (1966)
18. Piaget, J., Garcia, R.: Psychogenesis and the History of Sciences. Columbia University Press, New York (1980)
19. Reed, J.N., Sinclair, J.E.: Motivating Study of Formal Methods in the Classroom. In: Dean, C.N., Boute, R.T. (eds.) TFM 2004. LNCS, vol. 3294, pp. 32–46. Springer, Heidelberg (2004)
20. Rosa, S.: Designing Algorithms in High School Mathematics. In: Dean, C.N., Boute, R.T. (eds.) TFM 2004. LNCS, vol. 3294, pp. 17–31. Springer, Heidelberg (2004)
21. Rudall, J.: From Z to SPIN in One Module. In: TFM: Practice and Experience Workshop, pp. 71–75. BCS-FACS, Oxford Brookes U. (2003)
22. Spivey, J.M.: The Z Notation: A Reference Manual, 2nd edn. Prentice-Hall International, UK (1992)
23. Tarkan, S.: The Formal Specification of a Kitchen Environment. Master's scholarly paper, University of Maryland (2009)
24. TEXto HTML translator, `http://hutchinson.belmont.ma.us/tth/` (2009-07-15)
25. The Promela Language, `http://en.wikipedia.org/wiki/Promela` (2009-07-15)
26. The SPIN Model Checker, `http://spinroot.com/` (2009-07-15)
27. Woodcock, J., Davies, J.: Using Z: specification, refinement, and proof. Prentice-Hall, Upper Saddle River (1996)
28. Z to Alloy Tutorial, `http://ztoalloy.cs.umd.edu/` (2009-07-15)

Teaching Program Specification and Verification Using JML and ESC/Java2

Erik Poll

Radboud University, Nijmegen, The Netherlands

Abstract. The paper summarises our experiences teaching formal program specification and verification using the specification language JML and the automated program verification tool ESC/Java2. This technology has proven to be mature and simple enough to introduce students to formal methods, even undergraduate students with no prior knowledge of formal methods and even only very basic knowledge of (Java) programming. However, there are some limitations on the kind of examples that can be comfortably tackled.

1 Introduction

Over the past years we have taught formal program specification and verification using the JML specification language for Java and the automated program verification tool ESC/Java2 to a variety of audiences. We have taught this as a small module as part of larger courses. The module consists of a 2 hour lecture to introduce the basic concepts and notations, and an afternoon exercise lab. The set-up of the practical work is that students annotate example code with JML contracts – expressing preconditions, object invariants, and to a lesser extent postconditions – in response to feedback from the tool.

We have given such classes to students taking a course on formal semantics and program logics (so that they know what Hoare triples and weakest preconditions are), but most classes have been given to students without any exposure to formal methods apart from basic propositional logic. We have also taught the module to Information Science[1] students who only have very basic knowledge of programming.

The outline of the rest of the paper is as follows. Section 2 discusses the motivation and aims for the course module. Sections 3 and 4 discuss JML and ESC/Java2, respectively. Section 5 explains the set-up of the exercise classes and Section 6 discusses the pitfalls and limitations in letting students work with ESC/Java2. Section 7 gives pointers to our course material and discusses some related possibilities. Finally, sections 8 and 9 evaluate and conclude.

2 Context and Goals

Our original motivation for the module was that in an existing course on semantics and logics students only experienced techniques such as Hoare-logic

[1] In Dutch: Informatiekunde.

J. Gibbons and J.N. Oliveira (Eds.): TFM 2009, LNCS 5846, pp. 92–104, 2009.
© Springer-Verlag Berlin Heidelberg 2009

and weakest-precondition calculus as paper-and-pencil exercises. We thought it would useful if students experienced the possibilities of such techniques in programming tools, not just to show the capabilities of such tools, but also to make the connection with programming as they know it, in normal Java rather than some toy imperative programming language.

As it became apparent that students did not really need much theoretical background to do practical exercises using ESC/Java2, and that students found them interesting, we reused the idea in other settings, for instance to make information science students appreciate the importance of documenting assumptions and constraints as part of specifications.

The aims of the course are

- to make students aware of the hidden assumptions and implicit constraints and design decisions there are in typical programs, or indeed in specifications;
- to teach them how such assumptions and constraints can be documented in contracts, esp. with preconditions and invariants, using JML;
- to let them experience the added value of doing this in a formal language amenable to tool support, namely that they can run the program checker ESC/Java2;
- for students with knowledge of program logics such as Hoare-logic and wp-calculi: to let them experience what using such techniques in practice can be like.

To achieve these aims, the exercises are designed to include implicit assumptions that are so obvious that they are easy to overlook (e.g. in the example in Fig. 2) and properties where the precision of a more formal language than English (or Dutch) is really useful (e.g. in the example in Fig. 1).

3 JML

JML is a specification language tailored to Java. It allows specifications to be added to Java code, as special comments after `//@` or between `/*@ ... @*/`, in the Design-by-Contract style of Eiffel [17]. The core constructs of JML are preconditions, postconditions and object invariants[2]. JML offers a large range of additional constructs, but these are typically best avoided by a novice user. The initiative to develop JML was taken by Gary Leavens [14], but it has grown into a wide collaboration, with many people contributing to the language definition and using it as specification language in tools. Many program analysis tools for Java support JML in one form or another. The original use of JML was for runtime assertion checking [8], but it has also been used by program verification tools, for instance ESC/Java(2) [11,12], JACK [6], KeY [1], Krakatoa [16], and LOOP [4], and the Java model checker Bogor [19]. For an – already somewhat outdated – overview of JML and JML tool support see [5]. More on ESC/Java(2) below in Section 4.

[2] Object invariants are sometimes called class invariants, but in our opinion this is confusing terminology.

Experiences with JML. One of the main design goals of JML is that it should be easy to understand and use for any Java programmer. Although the more complicated constructs of the language are certainly not suitable for the average Java programmer (their precise semantics can still lead to heated debates between experts on the JML mailing list), for the basic JML constructs, such as pre- and postconditions and object invariants, this is in our experience certainly the case. After a short explanation of these notions in a lecture all students can cope with this. Only a minimal amount of syntax needs to be learned, namely just the keywords `requires`, `ensures`, and `invariant`, and the syntax for implication `==>`, bi-implication `<==>`, and universal quantification `\forall`. (To prove the point, we will use JML syntax in the remainder of this paper without any further introduction.)

Only the notion of object invariant requires some attention. Courses on program verification typically include *loop* invariants, but not *object* invariants. (In the practice of writing modern OO code, the notion of object invariant may well be more relevant for students to know!) Intuitively, the notion of object invariant can be explained as being implicitly included in the pre- and postconditions of all methods, and in the postconditions of all constructors. However, one should be aware that this is oversimplifying things, and cutting some corners! Precisely defining the semantics of the apparently simple notion of object invariant is notoriously complicated in the presence of call-backs, dynamic binding, subclassing, and aliasing. This might be an interesting topic to explore in a more advanced course on formal methods, but is best avoided in a first introduction to the notion of Design-by-Contract. More about potential hassle with invariants later.

During exercise classes we notice that many students need a hint before they realise that a precondition that they keep repeating needs to be turned into an invariant. This is in part caused by the fact that the work is tool-driven, and ESC/Java2 will complain about missing preconditions, but not about missing invariants. It is good to point out that for nearly every field in a class there is an associated object invariant, even if it is just saying that some reference field `t` is never null,

```
//@ invariant t != null;
```

or some integer field `i` that is always non-negative,

```
//@ invariant i >= 0;
```

JML includes the concept of *exceptional* postconditions, aka `signals`-clauses, which express the postcondition that holds in case an exception (or an exception of a certain type) is thrown. In our experience, this notion is best avoided. It is very easy to get confused between specifying when an exception *may* be thrown and when it *must* be thrown. Our exercises, and indeed the basic setup of ESC/Java2, are geared to proving the absence of all runtime exceptions as a first step (and possibly only step!) in the verification. In our experience just proving this can expose plenty of implicit design decisions.

JML includes the possibility to express frame conditions by so-called *assignable clauses* (aka modifies clauses). While frame conditions are an interesting

concept, and crucial to the verification of imperative programs, it is best omitted for a first introduction to formal program verification. (By the way, the notions of object invariant and frame condition are the most important notions missing in traditional approaches to program verification, which just consider pre- and postconditions and loop invariants.)

Finally, we noticed that some students would include some superfluous universal quantifications in object invariants. For example, the invariant about the field `i` above might be written as

```
//@ invariant (\forall SomeClass s; s.i >= 0);
```

where `SomeClass` is the class where the invariant occurs. This is superfluous because object invariants specified for `this` are already implicitly quantified over all objects of the current class. An invariant with such a universal quantification is not only bad style, but it also causes complications in automated verification, as the use of universal quantification is a major bottleneck for automated theorem provers. More on this issue below in our discussion of the experiences with ESC/Java2 exercises in Section 5.

4 ESC/Java(2)

ESC/Java is a program verification tool developed at Compaq (formerly DEC, and subsequently HP) by Rustan Leino and his co-workers [11]. After the disbanding of that research group at Compaq, David Cok and Joe Kiniry have led valiant efforts to keep ESC/Java alive and further improve it, resulting in what is now called ESC/Java2 [12]. In the meantime, Rustan Leino has gone on to develop the Spec# specification language for C# and the associated Boogie verification tool at Microsoft research labs [3][3].

ESC stand for Extended Static Checker. The name was chosen to stress that using the tool is intended to be similar in experience to using an automated, push-button static analysis tool, or a type checker. The tool is geared to a 'lightweight' form of program verification, i.e. verifying relatively simple properties of code rather than detailed functional specifications. (Indeed, this limitation is one of the possible pitfalls we discuss later.) Still, the tool does program verification in the classical way, using weakest precondition generation[4] to produce verification conditions that are fed to an automated theorem prover, Simplify [10]. The users do not see the back-end theorem prover or the verification conditions that are generated, but get feedback about violated invariants, violated pre- or postconditions or unexpected runtime exceptions in specific execution paths.

The builders of ESC/Java have been keen to point out that their tool is neither sound nor complete, but aims to spot as many potential bugs with the minimum of effort. While this has proven to be a successful design decision, and it is a nice

[3] See `http://research.microsoft.com/specsharp`

[4] Or strongest postcondition generation, but the user does not even see the difference.

bold statement to challenge some fundamentalist doctrine about formal methods, the disclaimer that the tool is not sound can give the wrong impression. In our experience, ESC/Java is a lot "more sound" than some verification tools that do not make such explicit disclaimers. In particular, ESC/Java takes a rigorous (albeit not completely sound) approach to ensuring that object invariants are not broken, where it makes worst case assumptions about potential aliasing. Note that traditional Hoare logics or weakest precondition calculi do not take into account the notion of object invariant (or indeed aliasing), thus ducking this major complication in program verification.

5 Simple Exercises Using JML and ESC/Java2

For practical exercises we provide students with example programs that they have to annotate with JML and for which they possibly have to correct bugs in the Java code. They have to add specifications, either in response to warnings by ESC/Java2 (initially about unexpected runtime exceptions, later also about broken invariants and preconditions), or to formally express informal specifications that are given to them.

We do these exercises in a terminal room where there is some help around to answer questions and sometimes point students in the right direction. We have typically had 20 to 30 students doing the exercises, with initially three people around to help, but once the initial peak of questions and technical hassle in getting things to work has passed two people can cope easily. Plenty of students will manage to do the exercises on their own machine without any help.

Experiences Using ESC/Java2

It is nice to see students realising the added-value of formal specifications, in that tools can help them to spot bugs, including some subtle bugs that are very easy to overlook. Also, it brings home the message about the importance of making implicit assumptions and design decisions explicit, not just for tools to make sense of code, but also for humans to understand the code. Of course, in standard programming courses students will be taught to document design decisions, as informal comments in code, but then there is typically no tool actually using these comments, so that writing the documentation is only extra work without any immediate benefit.

Of course, the exercises are designed to illustrate that knowing which object invariants hold is useful, if not crucial, to de-bug code. In all honesty, some parts of our sample programs have been very carefully crafted to contain subtle bugs that formal specifications will reveal.

There are some simple practical tips that can help the students. Splitting large invariants with conjunctions into smaller ones will improve feedback from the tool. Adding `assert` annotations to the code – to find out what holds or does not hold at a particular program point – is a useful way to figure out why something fails to verify.

```
/* Objects of this class represent euro amounts. For example, an Amount
   object with euros == 1 cents == 55 represents 1.55 euro.

   1) We do not want to represent 1.55 euro as an object with
       euros == 0, cents == 155
      Specify an invariant that rules this out.

   2) We do not want to represent 1.55 euro as an object with
         euros == 2, cents == -45
      Specify one (or more) invariant(s) that rule this out.
  */

public class Amount{

 private int cents, euros;

 public Amount(int euros, int cents){
   this.euros = euros;
   this.cents = cents;
 }

 public Amount negate(){
   return new Amount(-cents,-euros);
 }

 public Amount add(Amount a){
   int new_euros = euros + a.euros;
   int new_cents = cents + a.cents;
     if (new_cents < -100) {
      new_cents = new_cents + 100;
      new_euros = new_euros - 1;
   }
   if (new_cents > 100) {
      new_cents = new_cents - 100;
      new_euros = new_euros - 1;
   }
   if (new_cents < 0 && new_euros > 0) {
       new_cents = new_cents + 100;
       new_euros = new_euros - 1;
   }
   if (new_cents >= 0 && new_euros <= 0) {
       new_cents = new_cents - 100;
       new_euros = new_euros + 1;
   }
   return new Amount(new_euros,new_cents);
 }
}
```

Fig. 1. Example exercise, with a non-trivial invariant to specify

```
public class Taxpayer {
  boolean isFemale, isMale, isMarried;
  Taxpayer father, mother, spouse;
  //@ invariant isMarried ==> spouse.spouse == this;
  int age, tax_allowance;

  //@ requires newSpouse != null;
  public void marry(Taxpayer newSpouse) {
    spouse = newSpouse;
   isMarried = true;
  }

  public void divorce() {
    spouse.spouse = null;
    spouse = null;
    isMarried = false;
  }
```

Fig. 2. Fragment of an example exercise, with an initial attempt at capturing some of the (many!) implicit invariants involved and preconditions needed to ensure that none of these are broken

Something that we did not anticipate was that for some students ESC/Java2 is their first experience of using an automated theorem prover. Simplify, the back-end theorem prover of ESC/Java2, is quite good at propositional logic, so it is an eye-opener for some students that they can use the tool to spot some less obvious consequences of their specifications, for instance when they are struggling with subtly different formulations of object invariants for the example in Figure 1. For example, after specifying (incorrectly, by the way)

```
//@ invariant euros > 0 ==> cents > 0;
//@ invariant euros < 0 ==> cents < 0;
```

ESC/Java2 will point out errors in claims such as

```
somemethod(){
 //@ assert !(euros <= 0 & cents => 0);
 ..
}
```

at particular program points.

A danger when using an automated program verification tool like ESC/Java2 is that students end up 'mindlessly' trying out specifications to stop the tool from complaining, without really thinking about the meaning of the specifications they write. It is useful to ask them questions to reflect on what they are doing and why. Having them work in pairs and discuss with others helps here. It is also useful to let students examine the code, and make them think about possible object invariants, *before* letting them use the tool.

One cause for confusion for students is that they initially do not realise that program verification is done in a modular fashion, per-method or per-constructor. If method `m()` calls method `n()`, then when the tool verifies `m()` it will only use the contract for method `n()`, and not look at its code. This means that the tool will complain about programs that are – in the eyes of the student – obviously correct, because they know the code of `n()`.

There is a deeper reason for this modularity, namely that method n may be overridden in a subclass. JML enforces the notion of behavioural subtyping, which says that any methods overridden in a subclass have to satisfy any contracts written in the parent class.

The same issue of modularity can also cause some confusion with constructors and invariants. For example, students are typically surprised that the tool complains about an integer field `n` potentially being negative, even if all constructors obviously initialise `n` to a non-negative value, and no code anywhere assigns negative values to `n`. In such cases the implicit invariant needs to be made explicit, by adding

```
//@ invariant n >= 0;
```

to the code.

The same deeper reason for this applies, of course: there could be constructors in subclasses that fail to establish the property, or methods that break it. Or, simpler still, someone making changes to the code of the original class could unwittingly break such representation invariants. Of course, the whole point of explicitly documenting design decisions – such as which invariants are supposed to hold – is to avoid such problems.

6 Limitations and Pitfalls in the Use of ESC/Java2

It would of course be nice to move to more ambitious programs for students to specify and verify, and also programs they write themselves, rather than programs that are given to them. However, there are some practical limitations to be aware of:

1. Firstly, there is the limited power of the back-end automated theorem prover. If specifications become too expressive, the verification conditions may be too complicated for the theorem prover.

   This problem typically surfaces when people try to write detailed function specifications that involve universal quantifiers. For example, attempts to verify the full functional correctness of some sorting algorithm – one of the standard examples in traditional course material on program verification – are unlikely to be successful[5]. Here the fact that the user doesn't get to

[5] In private communication, Cormac Flanagan has reported good experiences with letting students write an implementation of quicksort for which they have to check a partial specification, which only states that the result array is sorted, not that it is a permutation of the input array; this is still simple enough to avoid running into this problem.

see the back-end theorem prover becomes something of a disadvantage. The user notices that the tool cannot prove a specification, but cannot see where it goes wrong or where it misses some additional information. (By the way, exercises where there are some relatively basic properties to specify, such as object invariants, are in our opinion more realistic than the examples involving full-blown functional specifications that traditionally feature in course material on program verification! For larger programs functional specification quickly becomes infeasible, but specifying more basic properties, such as object invariants, remains feasible and interesting.)

2. A second limitation is the need for API specifications. To verify a piece of code one will need formal specifications of any API methods it uses. There have been some collective efforts to write JML specifications for parts of the Java API (see `http://www.jmlspecs.org`), but these only cover small parts of the API. Moreover, not only the *absence* of API specifications can be a problem, but also their *presence* can be: if the specifications are too expressive, one runs into the first limitation mentioned above.
3. A third limitation is a more fundamental complication for program verification of object-oriented programs, or indeed any imperative programs: *aliasing*, especially the way it interacts with the meaning of object invariants.

   Intuitively, during the execution of a method the invariant of the current object may temporarily be broken, as long as it is re-established at the end. However, an additional complication is that invoking a method on one object may break the invariant of another object. This may happen if the invariant of the one object refers to field of the other object, or if the object have fields that could potentially be aliased. ESC/Java2 is quite good (or, a less positive way of phrasing it, extremely paranoid) when it come to spotting potential trouble caused by aliasing. If two different objects have fields of compatible types, ESC/Java2 will consider the (worst case) scenario that these objects may be aliased, unless specifications explicitly rule this out.

   This is probably the major source of confusing error messages that ESC/Java2 may report to the unsuspecting user. There are ways to solve these issues, but simply spotting the source of the problem can be tricky.
4. Finally, there are limitations to the Java features that ESC/Java2 can handle. Most importantly, it does not support generics. The ongoing evolution of a programming language such as Java poses a serious challenge to the development of tool support, despite ongoing initiatives such as JML4 [7].

We have given one course where students did use ESC/Java2 on code that they wrote themselves from scratch. In that course students wrote Java Card code, for execution on smartcards. For Java Card applications the pitfalls mentioned above can be avoided:

1. By instructing students to specify only very simple properties (basically, absence of runtime exceptions), the first pitfall can be avoided.

2. Because Java Card provides only a very limited API, and there are good specifications for this entire API [18], the second pitfall mentioned above can be avoided.
3. Because Java Card programs are not very object-oriented – most objects are just arrays – problems with aliasing are relative easy to control.
4. Finally, there are no generics in Java Card.

Not surprisingly, many program verification tools for Java have focused on Java Card as an interesting application area, e.g. [6,16,1]. The fact that smartcard code is hard to debug – in the absence of a screen, you cannot debug by adding println's to the code – was a nice additional motivation for students to verify the code. Still, writing Java Card code is something of an obscure specialism, and installing this software on smartcards requires special skills, so this idea is not easy to re-use by others.

7 Pointers and Related Tools

All material we use is available on-line[6]. Much more teaching material using JML is available via the JML website[7]. There is a mailing list[8] to get help from experienced ESC/Java2 users, should that be necessary.

For years we have used the stand-alone version of ESC/Java2[9] which can be run from the Windows, Linux, UNIX, or MacOS command line, and proved easy to install. There are now also stable versions of ESC/Java2 available as Eclipse plugin, in the form of the Mobius Program Verification Environment[10], which is also integrated with the JML4 initiative [7]. Beware that some of the earlier attempts at Eclipse plug-ins for ESC/Java2 might not be easy to install.

A more ambitious course that aims at a thorough integration of JML (as well as BON) into a software engineering course, has been developed by Joe Kiniry and Daniel Zimmerman [13].

There are other program verification tools that one could use for exercise courses, notably Spec#/Boogie [3] for C#, or KeY [1] or Krakatoa [16] for Java. For Spec# there have been efforts to improve the handling of set comprehensions such as sum, min, and max to make the Spec# program verifier capable of verifying standard textbook examples fully automatically [15]. KeY and Krakatoa can expose more of their internal working, which may be useful as part of a course on say Hoare logics, where would want to show the internal workings, and not just use the program verifier as a black box. The KeY tool supports a simple while-language that could be used for this purpose [2]. Teaching material for the KeY tool is available from `http://www.key-project.org/teaching`.

Instead of program verification there is also the possibility to let students use a runtime assertion checker for JML, for instance using the JMLUnit combination

[6] From `http://www.cs.ru.nl/~erikpoll/Teaching/JML`
[7] At `http://www.jmlspecs.org/teaching.shtml`
[8] `https://lists.sourceforge.net/lists/listinfo/jmlspecs-escjava`
[9] Available from `http://kind.ucd.ie/products/opensource/ESCJava2`
[10] Available from `http://kind.ucd.ie/products/opensource/Mobius`

of JML runtime assertion checking with JUnit [9]. Beware that the collection of JML tools available of the web, including JML2, JML4, JML5, and OpenJML, can be a bit bewildering.

8 Evaluation

This JML and ESC/Java2 module has only been a small part in larger courses (in the order of one week in a 14 week term). Since course evaluations (via web questionnaires) are done for a course as a whole, these do not provide a lot of detailed information about this particular module. Still, students do regularly mention it as the most interesting part of the course in the section for open comments on the evaluation form. This confirms a lot of positive feedback we get from students in class.

The experience helping out during the exercise classes does confirm that the messages that the course tries to make come across. But apart from doing the exercises, there is no additional exam that would allow a more impartial assessment if the course meets its aims.

The main difference we noticed between information science students and those having some background in formal methods is that the latter are much more at ease with using propositional logic.

9 Conclusions

The good news is that program verification technology in tools such as ESC/Java2 is mature enough for *any* student to use – even first year undergraduates, and even as part of courses which are *not* specifically about formal methods, such as standard programming or software engineering courses. It is straightforward to explain students what they need to know in a two hour lecture and then let them play with the tool in a practical session for a couple of hours. It seems a missed opportunity if not all computer science students experience using such a program verification tool. The theory of program verification might be relegated to more specialised (Master) courses that not all students take, but the use of verification tools should not be.

The bad news is that the use of the tool is best limited to controlled experiments, where the students work with (essentially toy) programs that are supplied, rather than code they develop themselves, to avoid running into the problems mentioned in Section 6. Moreover, traditional program verification exercises, that involve detailed and complete functional specifications are best avoided, as explained in Section 6, though these might be feasible using Spec#, as discussed in Section 7.

An important positive aspect of using JML is that students see that formal program specification and verification can be applied to a real programming language, Java, rather than some toy while-language or guarded command language. Another positive aspect of getting students to use tools is that they experience the potential added value of writing formal specifications, namely that tools can

them help them to identify bugs and expose implicit hidden assumptions. We do not think it is a good idea to let students write formal specifications *without* ever letting them experience using a tool that shows them what the potential benefits might be. Most importantly perhaps, most students seem to enjoy playing with ESC/Java2.

Acknowledgements. Credit goes to the many people have contributed to the development of ESC/Java(2) over the years. ESC/Java was designed by Rustan Leino and Jim Saxe, and developed with help from Cormac Flanagan, Rajeev Joshi, Mark Lillibridge, Todd Millstein, Greg Nelson, Raymie Stata, and Caroline Tice. The ESC/Java2 initiative has been led by Joe Kiniry and David Cok, and includes contributions from Patrice Chalin, Julien Charles, Dermot Cochran, Matthew Dwyer, Arnout van Engelen, Alexander Fuchs, Connor Gallagher, Robin Green, Radu Grigore, George Hagen, Clément Hurlin, Perry James, Mikoláš Janota, George Karabotsos, Hermann Lehner, Alan Morkan, Michal Mosal, Carl Pulley, Frédéric Rioux, Will Sargent, and Aleksy Schubert.

References

1. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY tool. Software and System Modeling 4(1), 32–54 (2005)
2. Ahrendt, W., Bubel, R., Hähnle, R.: Integrated and tool-supported teaching of testing, debugging, and verification. In: 2nd Int. Conference on Teaching Formal Methods, TFM 2009 (to appear, 2009)
3. Barnett, M., DeLine, R., Fähndrich, M., Jacobs, B., Leino, K.R.M., Schulte, W., Venter, H.: The Spec# programming system: Challenges and directions. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 144–152. Springer, Heidelberg (2008)
4. van den Berg, J., Jacobs, B.: The LOOP compiler for Java and JML. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 299–312. Springer, Heidelberg (2001)
5. Burdy, L., Cheon, Y., Cok, D.C., Ernst, M.R., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer (STTT) 7(3), 212–232 (2005)
6. Burdy, L., Requet, A., Lanet, J.-L.: Java applet correctness: A developer-oriented approach. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 422–439. Springer, Heidelberg (2003)
7. Chalin, P., James, P.R., Karabotsos, G.: JML4: Towards an industrial grade IVE for Java and next generation research platform for JML. In: Shankar, N., Woodcock, J. (eds.) VSTTE 2008. LNCS, vol. 5295, pp. 70–83. Springer, Heidelberg (2008)
8. Cheon, Y., Leavens, G.T.: A runtime assertion checker for the Java Modeling Language (JML). In: Arabnia, H.R., Mun, Y. (eds.) The International Conference on Software Engineering Research and Practice (SERP 2002), June 2002, pp. 322–328. CSREA Press (2002)
9. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: The JML and JUnit way. In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 231–255. Springer, Heidelberg (2002)

10. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM 52(3), 365–473 (2005)
11. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI 2002), pp. 234–245 (2002)
12. Kiniry, J.R., Cok, D.R.: ESC/Java2: Uniting ESC/Java and JML. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 108–128. Springer, Heidelberg (2005)
13. Kiniry, J.R., Zimmerman, D.M.: Secret ninja formal methods. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 214–228. Springer, Heidelberg (2008)
14. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06t, Iowa State University, Department of Computer Science (June 2002)
15. Leino, K.R.M., Monahan, R.: Automatic verification of textbook programs that use comprehensions. In: ECOOP workshop on Formal Techniques for Java-like Programs, FTfJP 2007 (2007)
16. Marché, C., Paulin-Mohring, C., Urbain, X.: The KRAKATOA tool for certification of Java/JavaCard programs annotated in JML. J. Log. Algebr. Program. 58(1-2), 89–106 (2004)
17. Meyer, B.: Object-oriented Software Construction, 2nd edn. Prentice Hall, Englewood Cliffs (1997)
18. Mostowski, W.: Fully verified Java Card API reference implementation. In: Beckert, B. (ed.) Verify 2007: 4th International Verification Workshop, July 2007. CEUR WS, vol. 259 (2007)
19. Robby, Rodríguez, E., Dwyer, M.B., Hatcliff, J.: Checking JML specifications using an extensible software model checking framework. STTT 8(3), 280–299 (2006)

How to Explain Mistakes

Stefan Hallerstede and Michael Leuschel

University of Düsseldorf
Germany
{halstefa,leuschel}@cs.uni-duesseldorf.de

Abstract. Usually we teach formal methods relying for a large part on one kind of reasoning technique about a formal model. For instance, we either use formal proof or we use model-checking. It would appear that it is already hard enough to learn one technique and having to cope with two puts just another burden on the students. This is not our experience. Especially model-checking is easily used to complement formal proof. It only relies on an intuitive operational understanding of a formal model.

In this article we show how using model-checking, animation, and formal proof together can be used to improve understanding of formal models. We demonstrate how animation can help finding an explanation for a failing proof. We also demonstrate where animation or model-checking may not help and where proving may not help. For most part use of another tool pays off. Proof obligations present intentionally a static view of a system so that we focus on abstract properties of a model and not on its behaviour. By contrast model-checking provides a more dynamic view based on an operational interpretation. Both views are valuable aids to reasoning about a model.

1 Introduction

In Event-B [2] formal modelling serves primarily for reasoning: reasoning is an essential part of modelling because it is the key to understanding complex models. Reasoning about complex models should not happen accidentally but needs systematic support within the modelling method. This thinking lies at the heart of the Event-B method.

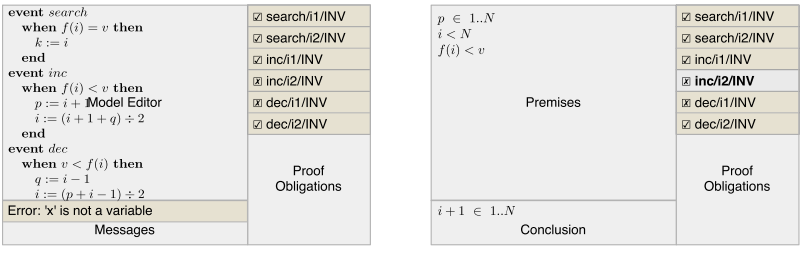
We use refinement to manage the many details of a complex model. Refinement is seen as a technique to introduce detail gradually at a rate that eases understanding. The model is completed by successive refinements until we are satisfied that the model captures all important requirements and assumptions. In this article we concern ourselves only with what is involved in coming up with an abstract model of some system. Note that refinement can also be used to produce implementations of abstract models, for instance, in terms of a sequential program [1,11]. But this is not discussed in this article.

We present a worked out example that could be used in the beginning of a course on Event-B to help students develop a realistic picture of the use of formal methods. The challenge is to state an example in such a way that it is easy to follow but provides enough opportunity to make (many) mistakes. We chose to use a sized-down variant of the access control model of [2] which we have employed for lectures at ETH Zürich (Switzerland) and at the University of Southampton (United Kingdom). We have not

J. Gibbons and J.N. Oliveira (Eds.): TFM 2009, LNCS 5846, pp. 105–124, 2009.
© Springer-Verlag Berlin Heidelberg 2009

used the description of a computer program because that is too rigid to exhibit possible misunderstandings. We have tried this using linear and binary search (together) but these are easy to formalise. So we have worked out this example for the next winter semester at the University of Düsseldorf. We begin by stating a problem to be solved in terms of assumptions and requirements and show how the problem can be approached using formal methods. The process of creating the model is shown in [6]. In this article we focus on how to understand mistakes made during modelling and the ensuing corrections. Whereas [6] is mostly about how proof can be used to improve a model, in this article we focus on complementary techniques based on model-checking and animation. These are only hinted at in [6]. In this respect this article can be regarded a sequel to [6]. (But it can be read independently.)

Rodin [3] is an extensible tool for formal modelling in Event-B. It is designed to support an incremental style of modelling where frequent changes are made to a model. Rodin produces proof obligations from Event-B models that subsequently are either proved automatically by an automatic theorem prover or manually by the user of the tool if the automatic theorem prover fails. Rodin is implemented on top of the rich client platform Eclipse [5]. It comes with two default screen layouts that are considered to help making the connection between formal model, proof obligations, and proof. The Eclipse platform supplies everything that is needed to make switching between the two layouts easy. Figure 1 shows a simplified sketch of the two default layouts. The



(a) Modelling layout **(b)** Proving layout

Fig. 1. Layouts of Rodin for Modelling and Proving

modelling layout (Figure 1a) provides an area for editing models, one for showing error messages on the bottom, and another for viewing and selecting proof obligations by name on the right. When a proof obligation is selected, the layout is changed to the one shown in Figure 1b. The proof obligation is shown in two areas arranged vertically, the hypothesis on top, the goal below it. On the right hand side the proof obligation names are displayed to permit browsing proof obligations. The tool encourages editing and experimenting with formal models. The focus is on modelling. Technicalities of proof obligation generation of concepts such as substitution are pushed aside. We believe, this is important if the tool is to be used by students having first contact with formal methods.

ProB [7,9] is an animator for B and Event-B built using constraint-solving technology. It incorporates optimisations such as symmetry reduction (see, e.g., [14]) and has been successfully applied to several industrial case studies. ProB is also used at several universities for teaching the B-Method [1,12].[1] In that context, the following features of ProB are relevant:

1. automatic animation. In particular, the tool tries to automatically find suitable values for the constants of a B-model based on constraint solving techniques,
2. consistency checking, i.e., checking whether the invariant of a B model is true in all states reachable from the initial states (called the *state space* of a model),
3. visualisation of counter examples or the full statespace of a model. Several reduction techniques have been implemented to compress large statespaces for visualisation [10].
4. trace refinement checking [8].

Figure 2 shows a layout for ProB animation where the user can inspect the current state and the history of events executed, choose the next event to be executed, and follow state changes in a graphical view of the state space.

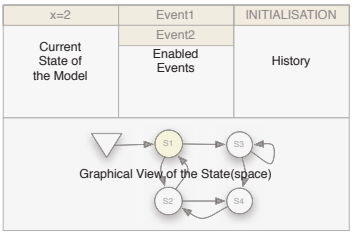


Fig. 2. Animation Layout of ProB for Event-B Models

We believe the combined use of proof, model-checking, and animation contributes highly to a better understanding of formal models.[2] They make it also easier to approach proof obligations and proof which are particularly hard to master and relate to formal models by novices. The figures shown are edited from the output that is provided by ProB. We have done this to improve readability of this article. The output of ProB is intended for viewing on a computer screen. Still, as a side effect of this exercise we have developed some ideas for improving the output of ProB.

Overview. In Section 2 we introduce Event-B. The following sections are devoted to solving a concrete problem in Event-B. In Section 3 the problem is stated. An abstract model is discussed in Section 4. In Section 5 we elaborate the model by refinement.

[1] For example, Besançon, Nantes in France; Southampton and Surrey in England; McMaster University, in Canada, Uppsala University in Sweden, and of course Düsseldorf in Germany.

[2] In a companion paper we also investigate the usefulness of graphical visualisation of formal models, which makes animation sequences even easier to comprehend.

2 Event-B

Event-B models are described in terms of the two basic constructs: *contexts* and *machines*. Contexts contain the static part of a model whereas machines contain the dynamic part. Contexts may contain *carrier sets*, *constants*, *axioms*, where carrier sets are similar to types [4]. In this article, we simply assume that there is some context and do not mention it explicitly. Machines are presented in Section 2.1, and proof obligations in Section 2.2 and Section 2.3. All proof obligations in this article are presented in the form of sequents: "premises" $\vdash$ "conclusion".

Similarly to our course based on [2], we have reduced the Event-B formalism so that a small subset of the notation suffices and formulas are easier to comprehend. In particular, the relationship between formal model and proof obligations is much easier to exhibit.

2.1 Machines

Machines provide behavioural properties of Event-B models. Machines may contain *variables*, *invariants*, *theorems*, *events*, and *variants*. Variables $v = v_1, \ldots, v_m$ define the state of a machine. They are constrained by invariants $I(v)$.[3] Theorems are predicates that are implied by the invariants. Possible state changes are described by means of events $E(v)$. Each event is composed of a *guard* $G(t, v)$ and an *action* $x := S(t, v)$, where $t = t_1, \ldots, t_r$ are the *parameters* of the event and $x = x_1, \ldots, x_p$ are the variables it may change[4]. The guard states the necessary condition under which an event may occur, and the action describes how the state variables evolve when the event occurs. We denote an event $E(v)$ by

$$
\begin{array}{lcl}
E(v) & \widehat{=} & \mathsf{any}\ \ t\ \ \mathsf{when} \\
 & & \quad G(t,v) \\
 & & \mathsf{then} \\
 & & \quad x := S(t,v) \\
 & & \mathsf{end}
\end{array}
\qquad \text{or} \qquad
\begin{array}{lcl}
E(v) & \widehat{=} & \mathsf{begin} \\
 & & \quad x := S(v) \\
 & & \mathsf{end}
\end{array}
\quad .
$$

The short form on the right hand side is used if the event does not have parameters and the guard is `true`. A dedicated event of the latter form is used for *initialisation*. The action of an event is composed of several *assignments* of the form

$$x_\ell := B_\ell(t, v) \quad ,$$

where x_ℓ is a variable and $B_\ell(t, v)$ is an expression. All assignments of an action $x := S(t, v)$ occur simultaneously; variables y that do not appear on the left-hand side of an assignment of an action are not changed by the action, yielding one simultaneous assignment

$$x_1, \ldots, x_p, y_1, \ldots, y_q := B_1(t, v), \ldots, B_p(t, v), y_1, \ldots, y_q \quad , \tag{1}$$

[3] Given the invariant I over the variables v, $I(t_1, \ldots, t_m)$ can be seen to stand for $I[t_1/v_1, \ldots, t_m/v_m]$. E.g., $I(v) = I$. We use a similar notation for other concepts, such as events, guards and actions.

[4] Note that, as x is a list of variables, $S(t, v)$ is a corresponding list of expressions.

where $x_1, \ldots, x_p, y_1, \ldots, y_q$ are the variables v of the machine. The effect of an action $x := S(t, v)$ of event $E(v)$ is denoted by the formula (1), whereas in the proper model we only specify those variables x_ℓ that may change.

2.2 Machine Consistency

Invariants are supposed to hold whenever variable values change. Obviously, this does not hold a priori for any combination of events and invariants $I(v) = I_1(v) \wedge \ldots \wedge I_i(v)$ and, thus, needs to be proved. The corresponding proof obligations are called *invariant preservation* ($\ell \in 1 \,..\, i$):

$$\begin{array}{l} I(v) \\ G(t,v) \\ \vdash \\ I_\ell\,(S(t,v)) \end{array} \quad , \tag{2}$$

for every event $E(v)$. Similar proof obligations are associated with the initialisation event of a machine. The only difference is that neither an invariant nor a guard appears in the premises of proof obligation (2), that is, the only premises are axioms and theorems of the context. We say that a machine is consistent if all events preserve all invariants.

2.3 Machine Refinement

Machine refinement provides a means to introduce more details about the dynamic properties of a model [4]. A machine N can refine at most one other machine M. We call M the *abstract* machine and N a *concrete* machine. The state of the abstract machine is related to the state of the concrete machine by a *gluing invariant* $J(v, w) = J_1(v, w) \wedge \ldots \wedge J_j(v, w)$, where $v = v_1, \ldots, v_m$ are the variables of the abstract machine and $w = w_1, \ldots, w_n$ the variables of the concrete machine.

Each event $E(v)$ of the abstract machine is *refined* by a concrete event $F(w)$. Let abstract event $E(v)$ with parameters $t = t_1, \ldots, t_r$ and concrete event $F(w)$ with parameters $u = u_1, \ldots, u_s$ be

$$\begin{array}{lcl} E(v) & \widehat{=} & \mathsf{any}\ \ t\ \ \mathsf{when} \\ & & \quad G(t,v) \\ & & \mathsf{then} \\ & & \quad v := S(t,v) \\ & & \mathsf{end} \end{array} \qquad \text{and} \qquad \begin{array}{lcl} F(w) & \widehat{=} & \mathsf{any}\ \ u\ \ \mathsf{when} \\ & & \quad H(u,w) \\ & & \mathsf{with} \\ & & \quad t = W(u) \\ & & \mathsf{then} \\ & & \quad w := T(u,w) \\ & & \mathsf{end} \end{array} .$$

Informally, concrete event $F(w)$ refines abstract event $E(v)$ if the guard of $F(w)$ is stronger than the guard of $E(v)$, and the gluing invariant $J(v, w)$ establishes a simulation of the action of $F(w)$ by the action of $E(v)$. The term $W(u)$ denotes *witnesses* for the abstract parameters t, specified by the equation $t = W(u)$ in event $F(w)$, linking abstract parameters to concrete parameters. Witnesses describe for each event

separately more specific how the refinement is achieved. The corresponding proof obligations for refinement are called *guard strengthening* ($\ell \in 1 \mathinner{..} g$):

$$\begin{array}{l} \quad I(v) \\ \quad J(v,w) \\ \quad H(u,w) \\ \vdash \\ \quad G_\ell\,(W(u),v) \quad , \end{array} \tag{3}$$

with the abstract guard $G(t,v) \;=\; G_1(t,v) \wedge \ldots \wedge G_g(t,v)$, and (again) *invariant preservation* ($\ell \in 1 \mathinner{..} j$):

$$\begin{array}{l} \quad I(v) \\ \quad J(v,w) \\ \quad H(u,w) \\ \vdash \\ \quad J_\ell\,(S(W(u),v),T(u,w)) \quad . \end{array} \tag{4}$$

Aside: Observe how the witness is used to reduce the complexity of the proof obligation compared to classical B, where a double negation appears in the refinement proof obligation [1]. Indeed, in classical B we have to prove that it is *possible* for the abstract model to make a corresponding step for every concrete step, or equivalently that it is *not* possible for the concrete model to make a step such that the abstract model can *not* imitate it and establish the gluing invariant (hence the double negation). Here, we require simply that for *all* abstract parameters having corresponding concrete parameters which make the witness predicate true, that the abstract event E can be triggered and establishes the gluing invariant. In general, Event-B contains many concepts that have been simplified compared to classical B. As such, it is inherently better suited for teaching formal methods.

2.4 Operational Interpretation

For the purpose of linking between Event-B to animation and model-checking it is convenient to give an operational interpretation to Event-B models [4]. We can observe events occurring and the resulting state changes. No two events may occur simultaneously. For the progress of "execution" resulting from event occurrences there are two possibilities:

(i) Some event guards are true: one of those events must occur.
(ii) All event guards are false: "execution" stops.

Following the informal description we can build a labelled transition system which represents our operational interpretation. We treat events as relations on a state space. The state space of a model is defined as the Cartesian product of the types of each of the model's variables. For convenience, we assume that every possible value can also be written as a constant expression.[5] A state in the state space is thus a vector of constant expressions describing the values for the variables.

[5] This is true for booleans, integers, enumerated sets and combinations thereof. It is generally not true for carrier sets; but in that case we can assume that a carrier set is instantiated by an enumerated set just for the purpose of animation and model checking.

Let $E(v)$ be an event with guard $G(t, v)$ and action $x := S(t, v)$, as defined above in Section 2.1. The events induce a labelled transition relation on states in the state space: state s is related to state s' by event $E(v)$ with parameter values a, denoted by $s \rightarrow^{M}_{E.a} s'$, when $G(a, s)$ holds and s' corresponds to the left-hand side of formula (1), that is to the successor state, with $t = a$ and $v = s$.

The syntactic constraints on the initialisation event in Event-B are such that the outcome of the initialisation will be independent of the initial values of the variables. This means the initialisation has the form

$$\begin{array}{l}\textsf{begin} \\ \quad v := E \\ \textsf{end}\end{array} \tag{5}$$

where E is a constant expression. The expression E is used to define the initial state for a machine.

Graphically, the state space of an Event-B model looks like in Figure 3. (In general, we would have a set of initial states in a non-deterministic initialisation represented by a predicate P. For convenience we therefore add a special state $root$, where we define $root \rightarrow^{M}_{initialise} s$ if s satisfies the initialisation predicate P. The root is shown here because it is used in the general form of initialisation.)

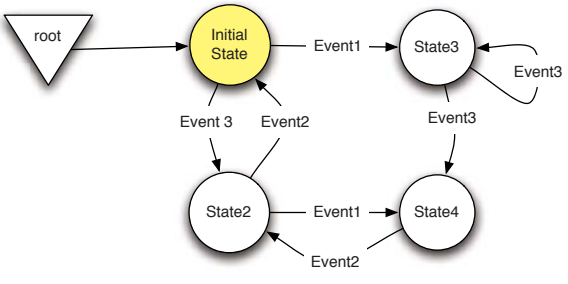


Fig. 3. A simple state space with four states

3 Problem Statement

In the following sections we develop a simple model of a secure building equipped with access control. The problem statement is inspired by a similar problem used by Abrial [2]. In [6] we have presented how the model could have been produced, especially, making mistakes that could have been made and subsequent improvements to the model. In this article we focus on how to comprehend mistakes made using different views on the model, easing the formal character of modelling. In order to do this we rely heavily on the ProB tool.

The model to be developed is to satisfy the following properties:

P1 : The system consists of persons and one building.
P2 : The building consists of rooms and doors.
P3 : Each person can be at most in one room.
P4 : Each person is authorised to be in certain rooms (but not others).
P5 : Each person is authorised to use certain doors (but not others).
P6 : Each person can only be in a room where the person is authorised to be.
P7 : Each person must be able to leave the building from any room where the person is authorised to be.
P8 : Each person can pass from one room to another if there is a door connecting the two rooms and the person has the proper authorisation.
P9 : Authorisations can be granted and revoked.

Properties P1, P2, P8, and P9 describe environment assumptions whereas properties P3, P4, P5, P6, and P7 describe genuine requirements. It is natural to mix them in the description of the system. Once we start modelling, the distinction becomes important. We have to prove that our model satisfies P3, P4, P5, P6, and P7 assuming we have P1, P2, P8, and P9.

4 Abstract Model

Our aim is to produce a faithful formal model of the system described by the properties P1 to P9 of Section 3. We choose to proceed in two modelling steps:

(i) the *abstract machine* (this section) models room authorisations;
(ii) the *concrete machine* (Section 5) models room and door authorisations.

To create an abstract model we need an abstract way of representing persons and rooms. Using these two concepts we can model property P4 as a relation between persons and rooms and property P3 as a function from persons to rooms. We declare two carrier sets for persons and rooms, $Person$ and $Room$, and a constant $\boldsymbol{O}$, where $\boldsymbol{O} \in Room$. Constant $\boldsymbol{O}$ models the *outside* of the building. We choose to describe the state by two variables for authorised rooms and locations of persons, arm and loc, with invariants

$inv1 : arm \in Person \leftrightarrow Room$	Property P4
$inv2 : Person \times \{\boldsymbol{O}\} \subseteq arm$	
$inv3 : loc \in Person \rightarrow Room$	Property P3
$inv4 : loc \subseteq arm$	Property P6

The invariant $inv2$ states that a person is always allowed to be outside, and as such partly formalises P7. These four invariants form the foundation of our abstract model. Next we present the events that model the dynamic aspects of the model.

In this and the next section we encounter mistakes in the model similar to those in [6]. However, we are mostly interested in motivating and explaining mistakes and relating them to the formal model. Novices often have problems when presented with more complicated formulas or proof obligations. A difficulty with formal methods, in general,

is that formulas get complicated rather quickly. By choosing different views at the same mistakes we can make them more approachable. For students to get the most out of this exercise, the same software tools that we use to demonstrate formal reasoning should be available to them. This encourages use of the tools and by experimenting with formal models the students can gain a deeper understanding of formal models they create.

In our abstract model to satisfy $inv2$, $inv3$ and $inv4$ we let

initialisation
  begin
    $act1 \;:\; arm := Person \times \{\boldsymbol{O}\}$
    $act2 \;:\; loc := Person \times \{\boldsymbol{O}\}$
  end .

We model passage from one room to another by event $pass$,

$pass$
  any p, r when
    $grd1 \;:\; p \mapsto r \in arm$ — p is authorised to be in r
    $grd2 \;:\; p \mapsto r \notin loc$ — but not already in r
  then
    $act1 \;:\; loc := loc \lhd\!\!\!- \{p \mapsto r\}$
  end .

Event $pass$ partially models property P8 ignoring doors for the moment. Granting and revoking authorisations for rooms is modelled by the two events

$grant$
  any p, r when
    $grd1 \;:\; p \in Person$
    $grd2 \;:\; r \in Room$
  then
    $act1 \;:\; arm := arm \cup \{p \mapsto r\}$
  end

$revoke$
  any p, r when
    $grd1 \;:\; p \in Person$
    $grd2 \;:\; p \mapsto r \notin loc$
  then
    $act1 \;:\; arm := arm \setminus \{p \mapsto r\}$
  end .

The two events do not yet model all of P9 which refers to authorisations in general, including authorisations for doors. Events $grant$ and $revoke$ appear easy enough to get right. However, a simple oversight can lead to a mistake. Event $revoke$ violates invariant $inv2$,

$$
\begin{array}{ll}
 & Person \times \{\boldsymbol{O}\} \subseteq arm \qquad \textit{Invariant } inv2 \\
 & p \in Person \qquad \textit{Guard } grd1 \\
 & p \mapsto r \notin loc \qquad \textit{Guard } grd2 \\
\vdash & \\
 & Person \times \{\boldsymbol{O}\} \subseteq arm \setminus \{p \mapsto r\} \qquad \textit{Modified invariant } inv2
\end{array}
$$

We could find out what is wrong only by inspecting the proof obligation. This would require carrying out some proof steps and understanding where it fails. Alternatively

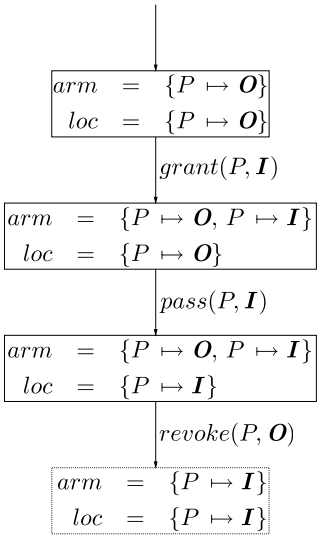


Fig. 4. A state trace leading to an inconsistent state

we can model-check our abstract model based on the operational interpretation. In an instance of the model with two different rooms $\boldsymbol{I}$ and $\boldsymbol{O}$ and one person P the model-checker yields the counter example in the form of a state trace shown in Figure 4. The state

$$arm = \{P \mapsto \boldsymbol{I}, P \mapsto \boldsymbol{O}\}, \quad loc = \{P \mapsto \boldsymbol{I}\}$$

is reachable in three steps. Letting parameter $r = \boldsymbol{O}$ in $revoke$, a state violating $inv2$ is reached. We see that we must not remove $\boldsymbol{O}$ from the set of authorised rooms of any person. To achieve this, we add a third guard to event $revoke$:

$$grd3 : r \neq \boldsymbol{O} \quad .$$

The counter example provides valuable information based on what the model "does". We can look at event $revoke$ to see what needs to be changed, or better, feed the finding into the proof obligation. With the new information at hand we can see clearly that the conclusion $Person \times \{\boldsymbol{O}\} \subseteq arm \setminus \{p \mapsto r\}$ does not hold if $r = \boldsymbol{O}$.

The model we have obtained thus far is easy to understand. Ignoring the doors in the building, it is quite simple but already incorporates properties P3, P4, and P6. Its simplicity permits us to judge more readily whether the model is reasonable. We can inspect it or animate it and can expect to get a fairly complete picture of its behaviour. We may ask: Is it possible to achieve a state where some person can move around in the building? A simplified state-transition graph can summarise such information comprehensively.

Figure 5 shows a slice of the model constructed by ProB for two persons and two rooms $\boldsymbol{I}$ and $\boldsymbol{O}$ where only the ranges of loc and of arm are considered. The states have

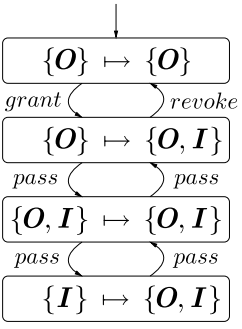


Fig. 5. Transitions on a simpler state space, showing $ran(loc) \mapsto ran(arm)$

been aggregated in sets according to the expression $ran(loc) \mapsto ran(arm)$. We can see how the two persons can pass between the rooms. It is not possible to tell which person is in which room but we can see that event $pass$ can occur and that locations change.

5 Concrete Model

We are satisfied with the abstract model of the secure building for now and turn to the refinement where doors are introduced into the model. A door will be represented as a pair consisting of two rooms. In the refined model we employ two variables adr for authorised doors and loc for the locations of persons in the building (as before). Variable adr is a function which indicates for every person the set of doors he is allowed to use. The intention is to keep the information contained in the abstract variable arm implicitly in the concrete variable adr. That is, in the refined model variable arm would be redundant. We specify

$$inv5 \ : \ adr \in Person \rightarrow (Room \leftrightarrow Room) \qquad \text{Property P5}$$

$$inv6 \ : \ \forall q \cdot \mathsf{ran}(adr(q)) \subseteq arm[\{q\}] \qquad \text{Property P4}$$

Any problem we can have with initialisation we can have with any other event, too; but in an initialisation they look less interesting because nothing can happen if an initialisation is wrong. (This is not an argument for ignoring initialisation when teaching modelling but for presenting interesting problems. And those usually do not appear in initialisations.) However, we need to know what the initialisation is in order to analyse other events. We reason: In the abstract model all persons can only be outside initially. This corresponds to them not being authorised to use any doors,

initialisation
  begin
    $act1 \ : \ adr := Person \times \{\varnothing\}$
    $act2 \ : \ loc := Person \times \{\boldsymbol{O}\}$
  end .

5.1 Moving between Rooms

Let us first look at event $pass$. Only a few changes are necessary to model property P8,

$pass$
  any p, r when
    $grd1 : loc(p) \mapsto r \in adr(p)$ *person p is authorised to enter room r from current location*
  then
    $act1 : loc := loc \lhd\!\!+ \{p \mapsto r\}$
  end .

We only have to show guard strengthening, because loc does not occur in $inv5$ and $inv6$. The abstract guard $grd1$ is strengthened by the concrete guards because $r \in \mathsf{ran}(adr(p))$ and by $inv6$, $\mathsf{ran}(adr(p)) \subseteq arm[\{p\}]$. The second guard strengthening proof obligation of event $pass$ is:

$$\begin{array}{ll} & loc \in Person \to Room & \textit{Invariant } inv3 \\ & loc(p) \mapsto r \in adr(p) & \textit{Concrete guard } grd1 \\ \vdash & & \\ & p \mapsto r \notin loc & \textit{Abstract guard } grd2 \end{array}$$

Using $inv3$ we can rephrase the goal,

$$\begin{array}{lll} & p \mapsto r \notin loc & \{\, inv3 \,\} \\ \Leftrightarrow & loc(p) \neq r & \end{array}$$

Neither concrete guard $grd1$ nor the invariants $inv1$ to $inv6$ imply this. If we animate the abstract and the concrete machine simultaneously, we find that in the concrete machine a person can pass from some room into the same room. In the abstract machine this is not possible as can be seen in Figure 6. We could add a guard $loc(p) \neq r$ to the concrete model but this would make event $pass$ express *a person can pass through a door if it connects two different rooms*. However, the model should not contain doors

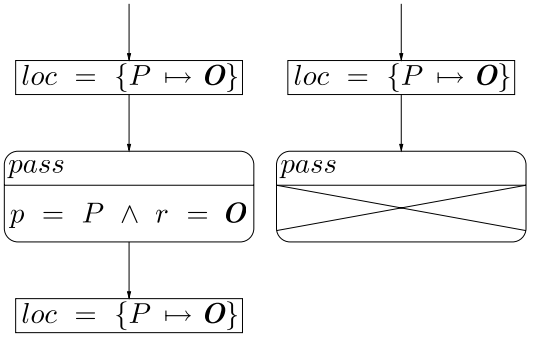


Fig. 6. Simultaneous animation of concrete (left) and abstract (right) machine

that connect rooms to themselves in the first place. The invariant is too weak. We do not specify that doors connect *different* rooms. In fact, our model of the building is rather weak. We decide to model the building by the doors that connect the rooms in it. They are modelled by a constant $Door$. We make the following two assumptions about doors:

$$axm1 \ : \ Door \ \in \ Room \leftrightarrow Room \qquad \textit{Each door connects two rooms.}$$
$$axm2 \ : \ Door \cap \mathsf{id}_{Room} \ = \ \varnothing \qquad \textit{No door connects a room to itself.}$$

A new invariant $inv7$ prevents doors connecting rooms to themselves. We realise that it captures much better property P5 than invariant $inv5$,

$$inv7 \ : \ \forall q \cdot adr(q) \ \subseteq \ Door \quad . \qquad \text{Property P5}$$

Using $inv7$ and $axm2$, we can prove $loc(p) \mapsto r \in adr(p) \Rightarrow loc(p) \neq r$ allowing to discharge the guard strengthening proof obligation above.

5.2 Leaving the Building

It may be necessary to pass though various rooms in order to leave the building. Hence, we need to specify a property about the transitive relationship of the doors. Property P7 is more involved.

A relation x is called *transitive* if $x ; x \ \subseteq \ x$. In other words, any composition of elements of x is in x. The transitive closure of a relation x is the least relation that contains x and is transitive. We define the *transitive closure* x^+ of a relation x by

$$\forall x \cdot x \ \subseteq \ x^+ \tag{6}$$
$$\forall x \cdot x^+ ; x \ \subseteq \ x^+ \tag{7}$$
$$\forall x, z \cdot x \ \subseteq \ z \ \wedge \ z ; x \ \subseteq \ z \ \Rightarrow \ x^+ \ \subseteq \ z \quad . \tag{8}$$

That is, x^+ is the least relation z satisfying $x \ \cup \ z ; x \subseteq \ z$.

Using the transitive closure of authorised rooms we can express that every person can at least reach the authorised rooms from the outside,

$$inv8 \ : \ \forall q \cdot arm[\{q\}] \ \subseteq \ adr(q)^+[\{\boldsymbol{O}\}] \cup \{\boldsymbol{O}\} \quad .$$

This invariant does not quite correspond to property P7. However, by the end of Section 5 we will be able to prove that all invariants jointly imply property P7 which we formalise as a theorem,

$$thm1 \ : \ \forall q \cdot (arm[\{q\}] \setminus \{\boldsymbol{O}\}) \times \{\boldsymbol{O}\} \ \subseteq \ adr(q)^+ \quad . \qquad \text{Property P7}$$

We proceed like this because we expect that proving $inv8$ to be preserved would be much easier than doing the same with $thm1$. Note, that being able to leave the building has little to do with moving between rooms but with granting and revoking authorisations. We do not formalise leaving the building only ability to do so. And this can appropriately done by means of an invariant or a theorem such as the above.

5.3 Granting Door Authorisations

A new door authorisation can be granted to a person if (a) it has not been granted yet and (b) authorisation for one of the connected rooms has been granted to the person. We introduce constraint (a) to focus on the interesting case and constraint (b) to satisfy invariant $inv8$. Thus,

$grant$
  any p, s, r when
    $grd1 : \{s \mapsto r, r \mapsto s\} \subseteq Door \setminus adr(p)$
    $grd2 : s \in \mathsf{dom}(adr(p)) \cup \{\boldsymbol{O}\}$
  then
    $act1 : adr := adr \Leftarrow \{p \mapsto adr(p) \cup \{s \mapsto r, r \mapsto s\}\}$ [6]
  end

In our model a door connecting a room r to another room s is modelled by the set of pairs $\{s \mapsto r, r \mapsto s\}$. Doors are modelled by their property of connecting two rooms in both directions. Each door D is a *symmetric* relation, that is, $D \subseteq D^{-1}$.

Unfortunately, we have introduced a deadlock. Figure 7 shows an example of a state trace leading to a deadlock. The guard of event $grant$ seems to be too strong. The

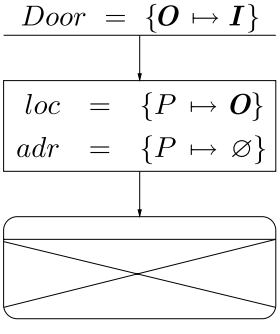


Fig. 7. State trace leading to a deadlock

problem is caused by the set of doors. It satisfies $Door \cap Door^{-1} = \varnothing$. We have not specified symmetry as a property of the set $Door$. Hence, there may not be any door $\{s \mapsto r, r \mapsto s\}$ contained in that set. Symmetry of the set $Door$ needs to be specified, too:

$axm3 : Door \subseteq Door^{-1}$ *Each door can be used in both directions*

[6] Event-B has the shorter (and more legible) notation $adr(p) := adr(p) \cup \{s \mapsto r, r \mapsto s\}$ for this. We do not use it because we can use the formula above directly in proof obligations. We also try as much as possible to avoid introducing more notation than necessary.

It is another assumption we have taken into account when modelling the building. It was not the guard of event $grant$ that was too strong, rather the assumptions about the building were too weak.

There is yet another problem. The guard of concrete event $grant$ is to weak to prove preservation of invariant $inv6$. (We could also look for problem with the action but this does not appear promising given all it does is to add the door $\{s \mapsto r, r \mapsto s\}$ to the authorisations of person p.) In fact, this cannot be spotted by animation or model-checking. The state of the system always satisfies the invariant and cannot reach an inconsistent state. However, the invariant does not provide enough information to prove this. We have a look at the corresponding proof obligation. For invariant $inv6$ we have to prove:

$$
\begin{array}{ll}
\forall q \cdot \mathsf{ran}(adr(q)) \subseteq arm[\{q\}] & \textit{Invariant } inv6 \\
\{s \mapsto r, r \mapsto s\} \subseteq Door \setminus adr(p) & \textit{Concrete guard } grd1 \\
s \in \mathsf{dom}(adr(p)) & \textit{Concrete guard } grd2 \\
\vdash & \\
\mathsf{ran}((adr \lhd\!\!+ \{p \mapsto adr(p) \cup \{s \mapsto r, r \mapsto s\}\})(q)) & \\
\quad \subseteq (arm \cup \{p \mapsto r\})[\{q\}] & \textit{Modified invariant } inv6
\end{array}
$$

for all q. For $q \neq p$ the proof is easy. For the other case $q = p$ we prove,

$$
\begin{array}{ll}
& \mathsf{ran}(adr(p) \cup \{s \mapsto r, r \mapsto s\}) \subseteq (arm \cup \{p \mapsto r\})[\{p\}] \\
\Leftarrow & \ldots \\
\Leftarrow & s \in \mathsf{ran}(adr(p))
\end{array}
$$

We would expect $s \in \mathsf{ran}(adr(p))$ to hold because doors are symmetric and because by concrete guard $grd2$ we have $s \in \mathsf{dom}(adr(p))$. Although only symmetric relations $\{s \mapsto r, r \mapsto s\}$ are added to $adr(p)$ it is recorded nowhere that $adr(p)$ itself is therefore a symmetric relation. We have to specify it explicitly,

$$
inv9 \ : \forall q \cdot adr(q) \subseteq adr(q)^{-1} \quad . \qquad \text{(see axiom } axm3\text{)}
$$

We can continue the proof where we left off

$$
\begin{array}{ll}
& s \in \mathsf{ran}(adr(p)) \qquad \{\, inv9 \text{ with } \text{“}q := p\text{”} \,\} \\
\Leftarrow & s \in \mathsf{dom}(adr(p))
\end{array}
$$

By adding invariant $inv9$ we have stated a property of the model that was already true. It just was not mentioned explicitly in the model. This property could only be discovered by proof [7].

5.4 Revoking Door Authorisations

We model revoking of door authorisations symmetrically to granting door authorisations. A door authorisation can be revoked if (a) there is an authorisation for the door, (b) the corresponding person is not in the room that could be removed, (c) the room is not the outside, and (d) all rooms except for the room to be removed must still be

reachable from the outside after revoking the authorisation for a door leading to that room. Condition (a) is just chosen symmetrically to $grd1$ of refined event $revoke$ (for the same reason). The other two conditions (b) and (c) are already present in the abstraction. The refined events $grant$ and $revoke$ together model property P9.

$$
\begin{array}{l}
revoke \\
\quad \mathsf{any} \quad p,\, s,\, r \quad \mathsf{when} \\
\quad\quad grd1 \;:\; s \mapsto r \in adr(p) \\
\quad\quad grd2 \;:\; p \mapsto r \notin loc \\
\quad\quad grd3 \;:\; r \neq \boldsymbol{O} \\
\quad\quad grd4 \;:\; \mathsf{ran}(adr(p)) \setminus \{r\} \subseteq (adr(p) \setminus \{s \mapsto r, r \mapsto s\})^{+}[\{\boldsymbol{O}\}] \cup \{\boldsymbol{O}\} \\
\quad \mathsf{then} \\
\quad\quad act1 \;:\; adr := adr \lhd\!\!\!\cdot \{p \mapsto adr(p) \setminus \{s \mapsto r, r \mapsto s\}\} \\
\quad \mathsf{end}
\end{array}
$$

We succeed proving guard strengthening of the abstract guards $grd1$ to $grd3$ and preservation of $inv5$, $inv6$, $inv7$, and $inv9$. But preservation of $inv6$ cannot be proved:

$$
\begin{array}{ll}
\forall q \cdot \mathsf{ran}(adr(q)) \subseteq arm[\{q\}] & \textit{Invariant } inv6 \\
s \mapsto r \in adr(p) & \textit{Concrete guard } grd1 \\
p \mapsto r \notin loc & \textit{Concrete guard } grd2 \\
r \neq \boldsymbol{O} & \textit{Concrete guard } grd3 \\
\vdash & \\
\quad \mathsf{ran}((adr \lhd\!\!\!\cdot \{p \mapsto adr(p) \setminus \{s \mapsto r, r \mapsto s\}\})(q)) & \\
\quad\quad \subseteq (arm \setminus \{p \mapsto r\})[\{q\}] & \textit{Modified invariant } inv6
\end{array}
$$

for all q. For $q = p$ we have to prove $\mathsf{ran}(adr(p) \setminus \{s \mapsto r, r \mapsto s\}) \subseteq arm[\{p\}] \setminus \{r\}$, thus, $r \notin \mathsf{ran}(adr(p) \setminus \{s \mapsto r, r \mapsto s\})$. This does not look right. Model-checking yields a counter example in form of a state trace; see Figure 8. (ProB permits to search directly for a violation of invariant $inv6$.) We find a counter example with one person P and three different rooms $\boldsymbol{H}, \boldsymbol{I}, \boldsymbol{O}$. We can reach the state:

$$
\begin{array}{l}
adr = \{P \mapsto \{\boldsymbol{O} \mapsto \boldsymbol{H}, \boldsymbol{H} \mapsto \boldsymbol{O}, \boldsymbol{O} \mapsto \boldsymbol{I}, \boldsymbol{I} \mapsto \boldsymbol{O}, \boldsymbol{I} \mapsto \boldsymbol{H}, \boldsymbol{H} \mapsto \boldsymbol{I}\}\} \\
arm = \{P \mapsto \boldsymbol{H}, P \mapsto \boldsymbol{I}, P \mapsto \boldsymbol{O}\} \\
loc = \{P \mapsto \boldsymbol{O}\} \quad .
\end{array}
$$

Revoking a door authorisation with parameters

$$p = P \quad s = \boldsymbol{I} \quad r = \boldsymbol{H}$$

leads to a state violating invariant $inv6$. In order to resolve this problem we could remove all doors connecting to r. But this seems not acceptable: we grant door authorisations one by one and we should revoke them one by one. We have to look at the problem from another angle. Figure 9 shows a continuation of the simultaneous trace of the abstract and the concrete machine. We cannot say anymore what happens in the abstract machine because the gluing invariant $inv6$ is violated. But we can still see

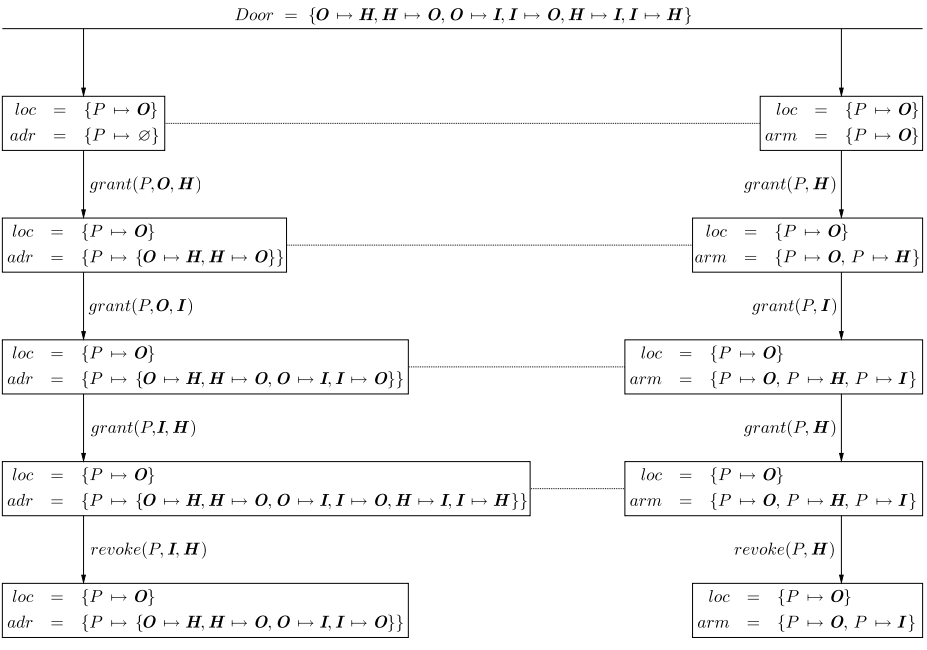


Fig. 8. Simultaneous state trace leading to an invariant violation

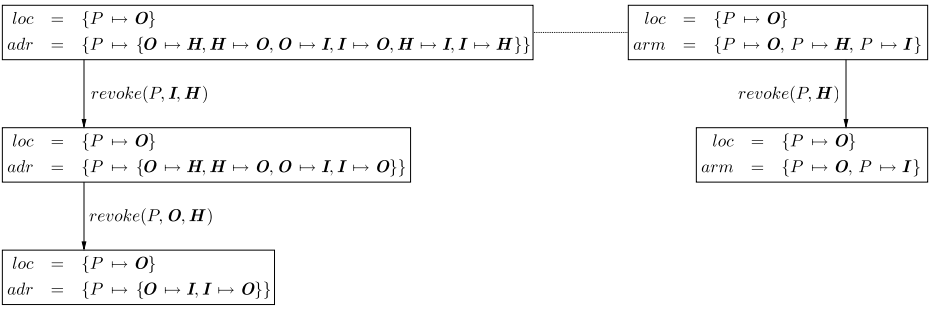


Fig. 9. The concrete trace continued

what the concrete machine could "do" next. Concrete revoke may occur again removing a door to $\boldsymbol{H}$. We could strengthen the guard of the concrete event requiring, say, $adr(p)[\{r\}] = \{s\}$. But then we would not be able to revoke authorisations once there are two or more doors for the same room. The problem is in the abstraction! We should allow abstract event $revoke$ to occur more often. It should not always remove r when it occurs. We weaken the guard of the abstract event using a set R of at most one room

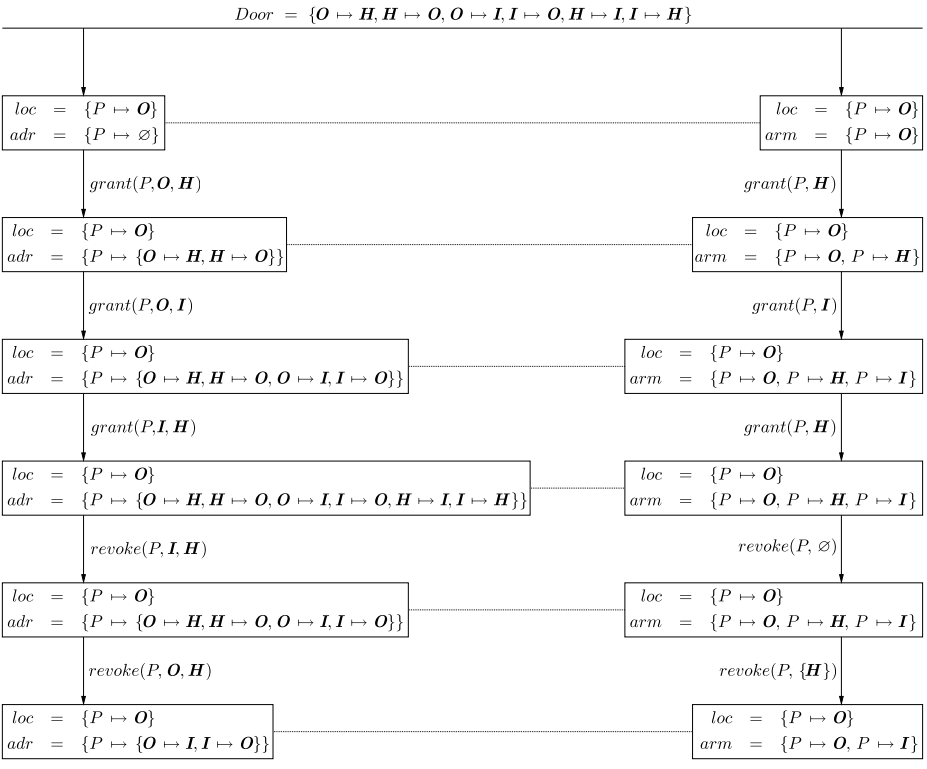


Fig. 10. Simultaneous trace of corrected model

instead of r. If $R = \varnothing$, then $\{p\} \times R = \varnothing$. So, for $R = \varnothing$ event $revoke$ does not change arm and for $R = \{r\}$ the effect of the event corresponds to the first attempt at abstract event $revoke$:

$$
\begin{array}{l}
revoke \\
\quad \mathsf{any} \quad p, R \quad \mathsf{when} \\
\qquad grd1 \;:\; p \in Person \\
\qquad grd2 \;:\; loc(p) \notin R \\
\qquad grd3 \;:\; R \in \mathbb{S}(Room \setminus \{\boldsymbol{O}\}) \\
\quad \mathsf{then} \\
\qquad act1 \;:\; arm := arm \setminus (\{p\} \times R) \\
\quad \mathsf{end} \quad ,
\end{array}
$$

where for a set X by $\mathbb{S}(X)$ we denote all subsets of X with at most one element:

$$Y \in \mathbb{S}(X) \quad \widehat{=} \quad Y \subseteq X \wedge (\forall x, y \cdot x \in Y \wedge y \in Y \Rightarrow x = y) \quad .$$

With this the proof obligation for invariant preservation of $inv6$ becomes:

$$
\begin{array}{ll}
\forall q \cdot \mathsf{ran}(adr(q)) \subseteq arm[\{q\}] & \textit{Invariant } inv6 \\
s \mapsto r \in adr(p) & \textit{Concrete guard } grd1 \\
p \mapsto r \notin loc & \textit{Concrete guard } grd2 \\
r \neq \boldsymbol{O} & \textit{Concrete guard } grd3 \\
\vdash & \\
\mathsf{ran}((adr \lhd\!\!\!\!- \{p \mapsto adr(p) \setminus \{s \mapsto r, r \mapsto s\}\})(q)) & \\
\quad \subseteq (arm \setminus (\{p\} \times R))[\{q\}] & \textit{Modified invariant } inv6
\end{array}
$$

for all q. For $q = p$ we have to prove,

$$\mathsf{ran}(adr(p) \setminus \{s \mapsto r, r \mapsto s\}) \subseteq arm[\{p\}] \setminus R \quad . \tag{9}$$

We need to make a connection between r and R. We need a witness for R. After some reflection we decide for

$$R = \{r\} \setminus \mathsf{ran}(adr(p) \setminus \{s \mapsto r, r \mapsto s\}) \quad . \tag{10}$$

Witness (10) explains how the concrete and the abstract event are related. If there is only one door s connecting to room r, then $R = \{r\}$ and the authorisation for room r is revoked. Otherwise, $R = \varnothing$ and the authorisation for room r is kept. The simultaneous trace (Figure 10) now confirms the correct behaviour and the proof succeeds too.

6 Conclusion

We have shown how different techniques of formal reasoning can be used jointly to understand and improve a formal model. In this article we have included formal proof, model-checking, and animation. This intended to be an open list. Sometimes we have used the result of model-checking or animation of a model to understand better problems that appeared in proof obligations. We have also seen cases where only model-checking showed that there was a problem. In the fragment of Event-B defined in Section 2 there is no mention of deadlock freedom, but the model-checker of ProB checks for it based on the operational interpretation. In all cases we have used the information gained as evidence from where to investigate and explain errors. We also saw that not all problems are found by model-checking or animation. Neither formal proof nor model-checking are complete in this sense.

A danger of using the operational interpretation is that students *only* think in terms of it, making it difficult to convey at the same time the usefulness of abstract reasoning by formal proof. But we think this is a very limited risk and the benefits outweigh it by far. In particular, comparing refinements by simultaneous traces helps greatly understanding particular refinements. An improved implementation of simultaneous model-checking and animation in ProB, using ideas of Brama [13], is under way.

Acknowledgement. This research was carried out as part of the EU research project DEPLOY (Industrial deployment of system engineering methods providing high dependability and productivity) http://www.deploy-project.eu/.

References

1. Abrial, J.-R. (ed.): The B-Book: Assigning Programs to Meanings. CUP, Cambridge (1996)
2. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2008) (to appear)
3. Abrial, J.-R., Butler, M., Hallerstede, S., Voisin, L.: An open extensible tool environment for Event-B. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 588–605. Springer, Heidelberg (2006)
4. Abrial, J.-R., Hallerstede, S.: Refinement, Decomposition and Instantiation of Discrete Models: Application to Event-B. Fundamentae Informatica 77(1-2) (2007)
5. Eclipse platform homepage, `http://www.eclipse.org/`
6. Hallerstede, S.: How to make mistakes. In: TFM B (to appear, 2009)
7. Leuschel, M., Butler, M.: ProB: A model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003)
8. Leuschel, M., Butler, M.: Automatic refinement checking for B. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 345–359. Springer, Heidelberg (2005)
9. Leuschel, M., Butler, M.J.: ProB: an automated analysis toolset for the B method. STTT 10(2), 185–203 (2008)
10. Leuschel, M., Turner, E.: Visualizing larger states spaces in ProB. In: Treharne, H., King, S., Henson, M.C., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 6–23. Springer, Heidelberg (2005)
11. Morgan, C.C. (ed.): Programming from Specifications, 2nd edn. Prentice Hall, Englewood Cliffs (1994)
12. Schneider, S.: The B-Method: An Introduction. Palgrave, Oxford (2002)
13. Servat, T.: BRAMA: A new graphic animation tool for B models. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 274–276. Springer, Heidelberg (2007)
14. Spermann, C., Leuschel, M.: ProB gets nauty: Effective symmetry reduction for B and Z models. In: Proceedings Symposium TASE 2008, Nanjing, China, June 2008, pp. 15–22. IEEE, Los Alamitos (2008)

Integrated and Tool-Supported Teaching of Testing, Debugging, and Verification

Wolfgang Ahrendt, Richard Bubel, and Reiner Hähnle

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
{ahrendt,bubel,reiner}@chalmers.se

Abstract. The course "Testing, Debugging, and Verification" is a non-traditional formal methods course that connects formal approaches to real-world development techniques in a novel way. A general theme in the course is that formalisation of specifications is the basis for debugging and test generation tools that go beyond what is possible with merely informal methods, and ultimately provides the opportunity of formal verification. Thereby, the course aims at integrating formal and informal methods as much as possible. The course is supposed to be accessible to participants without extensive mathematical training. We report about the design, implementation, and experiences with the course.

1 Introduction

The motivation and background for this paper is the design and implementation of a course called *Testing, Debugging, and Verification* (henceforth, called TDV) held at the Department of Computer Science and Engineering, Chalmers University of Technology.[1] This course is designed for third year students, i.e., the late Bachelor level. Consequently, it is not meant to provide a deep specialisation, like traditional introductions into formal methods, but it aims to inform systematically about a wide range of software validation methods that range from testing via debugging to formal verification. The course consists of thirteen lectures each of which takes 2×45mins, six exercise units, and three lab assignments. The students are credited 7.5 ETCS points.

In the TDV course, we aimed at integrating formal and informal methods as much as possible. We also attempted to make the course accessible to participants without extensive mathematical training. These are useful properties for many contemporary Bachelor programs. For this reason, we believe that the course concept as well as the lessons we learned from constructing and holding this course in its particular setting can be of general interest.

In the following section we explain the general teaching background and setting against which we developed the course, and we state the teaching goals and outcomes. In Section 3 we explain the conceptual choices we made to achieve the course goals. In Section 4 we describe the realisation of the course concepts, with

[1] Course Code TDA566, http://www.cse.chalmers.se/edu/course/TDA566/

J. Gibbons and J.N. Oliveira (Eds.): TFM 2009, LNCS 5846, pp. 125–143, 2009.
© Springer-Verlag Berlin Heidelberg 2009

an emphasis on the required technical and tool contributions. We also explain how the technical choices contribute to realizing the course goals. In Section 5 we summarise experiences, discuss alternative approaches, future development, and limitations.

2 Background and Goals

Engineering Tradition. Chalmers is a Technical University with a very strong Engineering tradition. The majority of Sweden's engineers have graduated from Chalmers. Since 2006, the education follows a 3-year Bachelor/2-year Masters system. The Bachelor programme is officially called the "base part of civil engineering education". Bachelor graduates from Chalmers are guaranteed a place in a suitable Masters programme and by default Masters graduates are also issued an engineering degree. As a consequence, there is still a strong flavour of a traditional engineering education.

Few Theoretical Prerequisites. For a course in formal methods this background has some important consequences: first, the Bachelor programmes are designed to provide broad knowledge in engineering. All programmes contain project-based courses that introduce into general engineering concepts. There is no Computer Science programme at the Bachelor level with strong theoretical foundations as can be found at many continental European universities. Mathematical courses concentrate on calculus, algebra, and statistics. There is a solid introduction into programming, including design and algorithms, but there is no room for courses dedicated to theoretical computer science, logic, or formal systems.

Hands-On Approach. A second consequence of the educational tradition at Chalmers is that most courses contain a "lab" component that may take up 20–50% of the time a student spends on a course. This practical part typically consists of 2–5 mini-projects done in teams where the students need to apply in practice what they have learned. In principle, it is a good thing to immediately reinforce by practice what has been learned, but it poses some challenges on the course design:

- the theoretical contents conveyed in the lectures must actually be relevant to solve the practical tasks, otherwise, the former may be dismissed as irrelevant;
- grading of the practical components must be reasonably economical (wrt. time spent), otherwise the course does not scale with the number of students;
- students have an increased expectation level with respect to practical applicability of course contents.

Transition to Masters Level. While the Bachelor education at Chalmers is relatively broad and practically geared, the picture changes at the Masters level: the Masters programmes are rather specialised and offer many advanced courses that

lead close to the frontiers of current research, and usually given by researchers which are active in the respective area.

All Masters programmes at Chalmers are held in English and more than one third of the Masters students in Computer Science-related programmes are non-Swedish. The influx of foreign students with widely differing backgrounds as well as the considerably more theoretical character of some advanced Masters courses can cause tension. To make the transition smoother the TDV course has been designated to be among the courses that provide a bridge between the Bachelor and Masters level. It can be taken both by 3rd year Bachelor students and 1st year Masters students, however, it is not obligatory for either.

Course Goals. Given the background described above we derived a number of goals that we wanted to achieve with the concept and the design of the TDV course:

Integration. Formal methods, that is, formal approaches for describing and analysing software systems should not come across as a more or less radical alternative to traditional software design techniques, but as an *integrated* aspect of software quality management. "Formal" and "informal" techniques are not be juxtaposed but, rather, formalisation is presented as a natural consequence of a systematic analysis and the desire to automate manual processes.

Diversity. Contemporary Software Engineering has a whole spectrum of validation and analysis methods to offer, some of which are informal though systematic (e.g., testing, debugging) and others make use of some kind of formal notation (e.g., automated test case generation (ATCG), formal specification, formal verification). Some methods are supposed to detect errors (testing, test generation), some to eliminate errors (debugging), and some to ensure that no errors w.r.t. a given specification are left (formal verification). This kind of diversity is essential to ensure efficient software construction with a high quality outcome. No method alone (e.g., *only* testing or *only* verification) is sufficient.

Applicability. Formal methods can be applied to real programs and problems, not only to toy languages. They can help to understand a program better (e.g., through visualisation of a symbolic execution tree or by verification of invariants), to detect problems (e.g., caused by insufficient specifications) or to save time during development (e.g., with automated test case generation). We show all these methods in action with actually executable JAVA programs. We choose JAVA, because all students are familiar with it.

Formalisation = Tool Support. Formalisation of software and its properties is not an end in itself, but it is a prerequisite for new and more far-reaching software analysis and design tools. Everyone uses compilers, and it is also very popular to illustrate software designs with diagrams. This is fine, but with a rigorous, formal description of the intended behaviour of a program, one can do much more.

Tools are essential. Without tools the potential of formalisation cannot be fully realised. Without tool support, formal specifications even of small programs inevitably are incomplete or wrong. Hand-written verification arguments are error-prone. In contrast to mathematical proofs, arguments about the correctness of programs must be formal *and mechanised*. Already with very lightweight usage of formalisation tools can save a lot of time (e.g., minimisation of test cases).

General Interest. We believe that the above list of desirable properties of an FM course is not unique to our situation at Chalmers: The Bachelor/Masters system led to a thinning-out of theory courses in many places. Likewise, one can observe an increased demand for "applicable" contents on the side of students. At the same time, academic programmes are opening up more and more for life-long education efforts, often in connection with industry. We see, therefore, the general need for "hands-on" formal methods courses that are accessible to experienced software designers who possess limited mathematical training. We are convinced that such a course must firmly place formalisation within the existing biosphere of available tools and methods. In addition, the prospect of saving time by using advanced tools that are enabled through formalisation, is a major motivation for increased rigour.

3 Concepts

In this section we explain the concepts that we chose to realise the course goals. The main message conveyed to students is that the course provides an overview of a broad range of software validation methods. To meet our diversity goal (see Section 2) we decided to address four essential activities that arise during software construction: *testing*, *specification*, *debugging*, and *verification*. These are covered in five teaching units as summarised in Table 1. Within each topic we selected a number of representative techniques. Obviously, it is not possible to be exhaustive, and other choices would have been possible. Alternatives are further discussed in Section 5. Additionally, we made two general design decisions:

1. Each teaching unit must involve practical exercises with at least one tool.
2. We do not introduce a formal, mathematical semantics of the specification languages we use (mainly JML [15] and, to some extent, first-order logic). Rather we teach them like a programming language: a systematic conceptual introduction backed up by examples. This decision is not only motivated by a lack of time, but also by our intention (See Section 2) to present formalisation as an immediately useful and readily applicable activity.[2]

In the following, we explain the approach that we took with respect to each of the activities testing, specification, debugging, and verification and how we achieve the goals laid down above. The close interdependencies between the various course topics are depicted in Figure 1.

[2] A similar approach has been recently taken by Ben-Ari's excellent introduction to model checking [5].

Table 1. Main characteristics of TDV teaching units

Teaching Unit	Content	Formal	Tools
Testing	Systematic testing, specification, assertions, black/white box, path/code coverage	no	JUnit
Debugging	Bug tracking, execution control, failure input minimisation, logging, slicing	no	`DDinput`, Eclipse, `log4j`
Formal Specification	Design-by-contract, formalisation, first-order logic, JML	yes	`jml` (type checker)
Automated Test Case Generation	Model-based TC generation, Symbolic execution, Code-based TC generation	yes	`jmlunit`, KeY VSD, KeY VBT
Formal Verification	Hoare triple, weakest precondition, formal verification, loop invariant	yes	KeY-Hoare

Testing. Testing is indispensable, even when formal techniques are in place, because of incomplete specifications or unavailable source code. Our take on testing includes two passes: in the first round, an overview over classic testing concepts is provided. Test cases and oracles are written and derived from the code and specification by hand, then executed automatically with JUnit (`http://www.junit.org`). After having introduced formal specification with JML we revisit testing and show that even a relatively simple model-based test generation tool such as `jmlunit` increases the degree of automation. Thereafter, we go one step further and introduce the fundamental technique of *symbolic execution* which is the basis of code-based test generation. We show that formalisation leads to automation (in model-based test generation) and that the dynamic analysis technique symbolic execution, which is practically and theoretically more difficult than static analysis, increases coverage and helps to understand programs.

Specification. The value of explicitly specifying requirements and properties of software is often underestimated by students. Therefore, we decided to make specification a permeating topic of the course that resurfaces as an essential prerequisite for nearly everything (see Figure 1). In the first pass on testing we stress that testing becomes arbitrary without a notion of what is being tested for. Informal specifications lead to test oracles that are crafted by hand. Later, specifications in JML partially automate test case and oracle generation. Obviously, specification is a prerequisite for formal verification, but also a source for useful initial states and the exclusion of unfeasible paths in debugging. For structuring

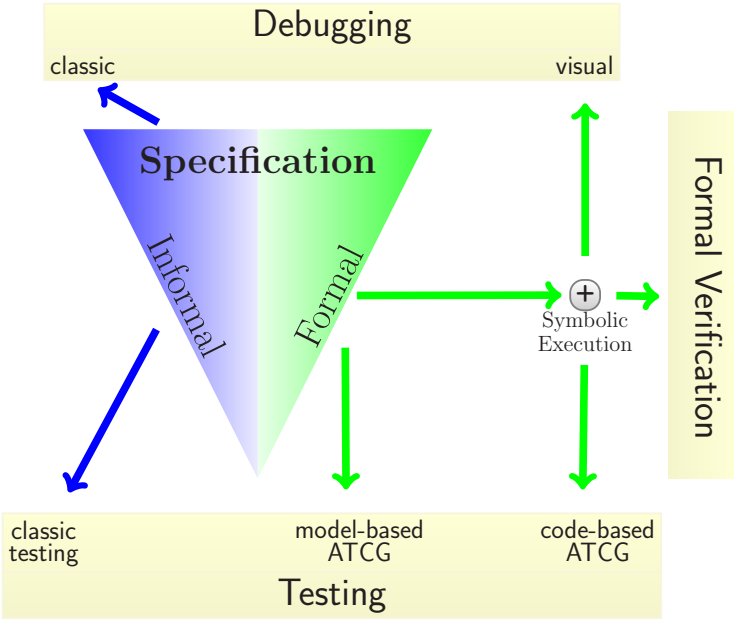


Fig. 1. Dependencies among TDV topics

specifications we follow the design-by-contract paradigm [18], starting with the observation that even in traditional JAVA development one implicitly programs with contracts.

During the course we gradually increase the rigour of contracts. This motivates the move from natural language to a dedicated specification language. The choice of JML is driven by the the fact that it is based on JAVA expressions, which makes the discussion of introductory examples very intuitive. Extending on such examples leads to the on-demand introduction of JML features that go beyond JAVA such as pre-state values and quantification.

Debugging. Given that debugging accounts for a substantial amount of developer's time it is surprising that Zeller's book [21] is the only systematic introduction to debugging on the market. Even in a comprehensive standard text on software engineering such as [20], only half a page of nearly 1000 pages is devoted to debugging. Debugging is also neglected by formal approaches. The most comprehensive textbook on Software Engineering based on a formal approach [6] does not mention debugging at all.

In the course, we present selected topics from [21]. We show that even a simple tool such as a minimisation algorithm for bug-inducing input can result in vast time savings. We also found that many students are unfamiliar with logging frameworks and modern interactive debuggers, so we include those as well. We stress the importance of bugs as a source for regression tests. We show

that tracing back an infected program state to the location of the bug that caused it can be systematised based on the fundamental notion of program slices. Altogether, we try to convey that the unavoidable activity of debugging is not a "black art", but a systematic craft that can be learned. It is also linked to testing as well as specification.

Verification. Throughout the course we stress that verification is a broad spectrum of informal as well as formal program analysis methods of which formal verification is the most rigorous. Informal verification methods include code reviews and metrics. More automation is provided by various static checking tools such as ESC/Java2 [9]. As informal verification is covered in other courses and we wanted the students to learn at least one important theoretical concept, we decided to concentrate on formal verification. By far the most popular approach to formal verification is Hoare logic which is what we align to as well. When we planned the course, we were very surprised that there was not a single verification tool on the market that can be used for teaching Hoare logic. All lecture notes on Hoare logic that we found were based on hand-written proofs. Not surprisingly, therefore, many exercises in lecture notes exhibited errors when we tried to machine-check them. As the TDV course is about verification, and not about first-order theorem proving, we wanted a tool with an oracle that can dispose of first-order verification conditions. In the end we created such a tool ourselves based on the Java verification system KeY. The KeY-Hoare tool is described in [7] and Section 4.4 below.

4 Realisation and Implementation

We describe in this section the realisation of the course concepts focusing in particular on the necessary technical and tool contributions.

4.1 Specification and the Java Modelling Language

Right from the beginning of the course, we put much emphasis on specification. At first, we teach how to write informal, but *precise* method specifications in natural language. A central goal of this exercise is to introduce concepts such as pre- and postconditions to document method behaviour. Also, we emphasise that programmers must be fully aware of the specification of code they use, and of the specification they implement. A central example is the consistency requirement on `equals()` and `hashcode()` as formulated in the inherited contract of `Object`. We demonstrate the common error to redefine `equals()` without redefining `hashcode()`. This usually violates the inherited contract and leads to unexpected results in the interaction with Java collections.

As an example of a formal specification language we introduce the Java Modelling Language (JML) [15,16]. JML is a so-called "one-tiered specification language" [13] whose expressions are a superset of Java's expressions and easy to master for programmers. JML specifications are attached to the implementations as structured comments in the source code. For the objectives of the

course, this technical integration supports the message of specification and implementation as being integrated activities.

Another reason for choosing JML as formal specification language is its sizable community among practitioners and readily available open source tool support such as ESC/Java2 [9] or the Common JML tools[3]. Out of those, we mostly use the `jml` syntax and type checker, without which most specifications written by students or even teachers are likely to not follow all restrictions imposed by the language. In particular, visibility rules are checked, as well as "purity" (side effect-freeness) of methods used in specifications. A further advantage of JML is the on-line availability of specifications for many standard library classes, following the pattern of Javadoc pages. (Library classes are, however, often too complex to serve as introductory examples.)

The close relation of Java and JML allows to introduce the latter entirely example-driven. Starting from monolithic natural language specifications as found in typical specification documents of APIs, we identify corresponding pre-/postcondition pairs. First examples are chosen such that informal pre- and postconditions can be turned into `boolean` Java expressions, thereby manifesting JML specifications already. When later examples exceed the expressiveness of Java, further JML features are introduced on demand such as access to pre-state values, or quantification. Finally, unsatisfactory attempts to express (i) unchanged program locations and (ii) consistency conditions on fields motivate the introduction of the concepts of (i) `assignable` clauses and (ii) class invariants that allow to express these requirements much more neatly.

4.2 Verification-Based Testing

In the TDV course we give a brief introduction on conventional testing theory and introduce common notions such as black-/white-box testing, coverage criteria, etc. (see Table 1). In accompanying demonstrations and exercises the students are encouraged to write their own unit tests by hand using the JUnit[4] framework.

Once formal specifications have been introduced, we revisit testing under the aspect of *automated test case generation* (ATCG).

We start with a black-box testing approach to ATCG, namely, model-based test generation. Model-based testing typically tries to logically cover specifications and select boundary cases by analysing the specification for e.g. implicit disjunctions. The generated test suite is supposed to contain at least one boundary test for each logical disjunct. Besides teaching the students a basic algorithm to generate such test cases, we let them experiment with the model-based test generation tool `jmlunit` [8] from the Common JML tools suite.

Finally, we present code-based test generation as a white-box approach to ATCG. In code-based test generation the control-flow of the code under test is analysed to generate test suites, normally achieving a higher branch and statement coverage than with model-based testing approaches. We focus on a recent

[3] `http://www.eecs.ucf.edu/~leavens/JML/download.shtml` (GPL)
[4] `http://www.junit.org` (CPL)

variant of code-based test generation that uses symbolic program execution as the underlying method to analyse the code under test [10,12].

Symbolic execution is a general dynamic analysis technique where program execution is performed with symbolic values rather than with concrete values for input data, and program outputs are expressed as logical or mathematical expressions involving those symbols [14]. Consequently, when statements are executed that cause the control-flow to branch, all possible continuations have to be considered. Thus a symbolic program run does not result in a single execution path (trace), but in an execution tree covering every possible concrete execution trace.

Formal specifications are used to prune infeasible branches of the symbolic execution tree. After some JAVA code has been symbolically executed up to some depth one attempts to generate a test case for each feasible branch of the resulting symbolic execution tree. It is possible to show that this ensures feasible branch coverage provided that the code under test is symbolically executed to sufficient depth.

In the course we use our own fully automatic verification-based test generation tool called KeY VBT[5] described in detail in [11,12] .

4.3 Debugging

To remedy the lack of systematic teaching of debugging observed in Section 3 we present a number of approaches to debugging in TDV. The students learn how to

log events systematically: instead of distributing print statements all over the code to narrow down code fragments that contain a bug, they are taught to log events using the `log4j`[6] framework developed by the Apache Software Foundation. It allows to log events driven by orthogonal criteria such as emitting component and event type/severity.

use debuggers to comprehend code and to locate erroneous program code: the students are taught stepwise execution of programs, breakpoints and variable inspection hands-on using the Eclipse debugger.

use delta debugging to automatically minimise the input revealing a certain error. The delta debugging algorithm [22,21] is taught together with usage of the `DDinput`[7] framework.

All these techniques belong to the informal spectrum of the software engineering field and do not require any *formal* methods. The educational objective is here to teach useful systematic debugging techniques and to let the students actively explore reach and limits of those tools.

[5] `http://www.key-project.org/download/testgen.html` (GPL)
[6] `http://logging.apache.org/log4j/1.2/index.html` (Apache License)
[7] `http://www.phbouillon.de/index.php?class=Calimero_Webpage\&id=23680`

Later in the course we connect debugging more tightly to the formal world. While we explain white-box automated test case generation we introduce a prototype of a visual symbolic-state debugger [3,19] integrating debugging, visualisation, and automatic test case generation (see Figures 2 and 3).

In contrast to a standard debugger, the visual debugger is based on symbolic execution. As described in Section 4.2 the idea of symbolic execution is to run a program not on concrete, but on symbolic input values. The set of the initial states to be considered can be restricted by adding constraints on the symbolic values in form of JML preconditions. The benefits of a visual symbolic debugger are three-fold:

1. A standard debugger executing a program on concrete input can only inspect a single run of a program at a time. The visual debugger in contrast inspects *all* possible runs of a program for any possible input simultaneously. This is feasible, as the symbolic execution engine generates a symbolic execution tree (Figure 2) rather than a single path.
2. Using symbolic values allows to start debugging at any given position in the source code. It is in particular not necessary to execute complex initialisation code establishing the initial state from where to start the actual debugging.
3. The symbolic execution tree represents a memory-efficient data structure of the complete symbolic execution history. As debugging can be started immediately at any source code position, it is possible to keep the length of the execution history small without losing information. Hence, the visual debugger is also an omniscient debugger [17] enabling the developer to jump back and forth through the execution history as well as to inspect and to compare different states without having to rerun the program. Omniscient debuggers based on standard debugging technology are limited by excessive memory consumption requirements for storing the program execution history which tends to be very long in the absence of symbolic initial states.

Further features of the visual debugger include stepwise symbolic program execution, breakpoints and watchpoints. Like standard debuggers stepwise symbolic execution permits to control the granularity of execution steps, letting the user decide to step into or over statements and methods. Watchpoints are floating breakpoints interrupting symbolic execution at nodes that satisfy a user-specified condition.

Besides the execution tree view (see Figure 2), the visual debugger provides also a visualisation of the symbolic state for any execution tree node. The symbolic state is visualised as a symbolic UML object diagram (see Figure 3). The user can browse through all possible configurations of the symbolic state.

Finally, the visual debugger integrates the automatic test case generation tool described in Section 4.2. This integration allows a convenient generation of regression tests. Whenever a test case fails, then the corresponding path in the symbolic execution tree is highlighted.

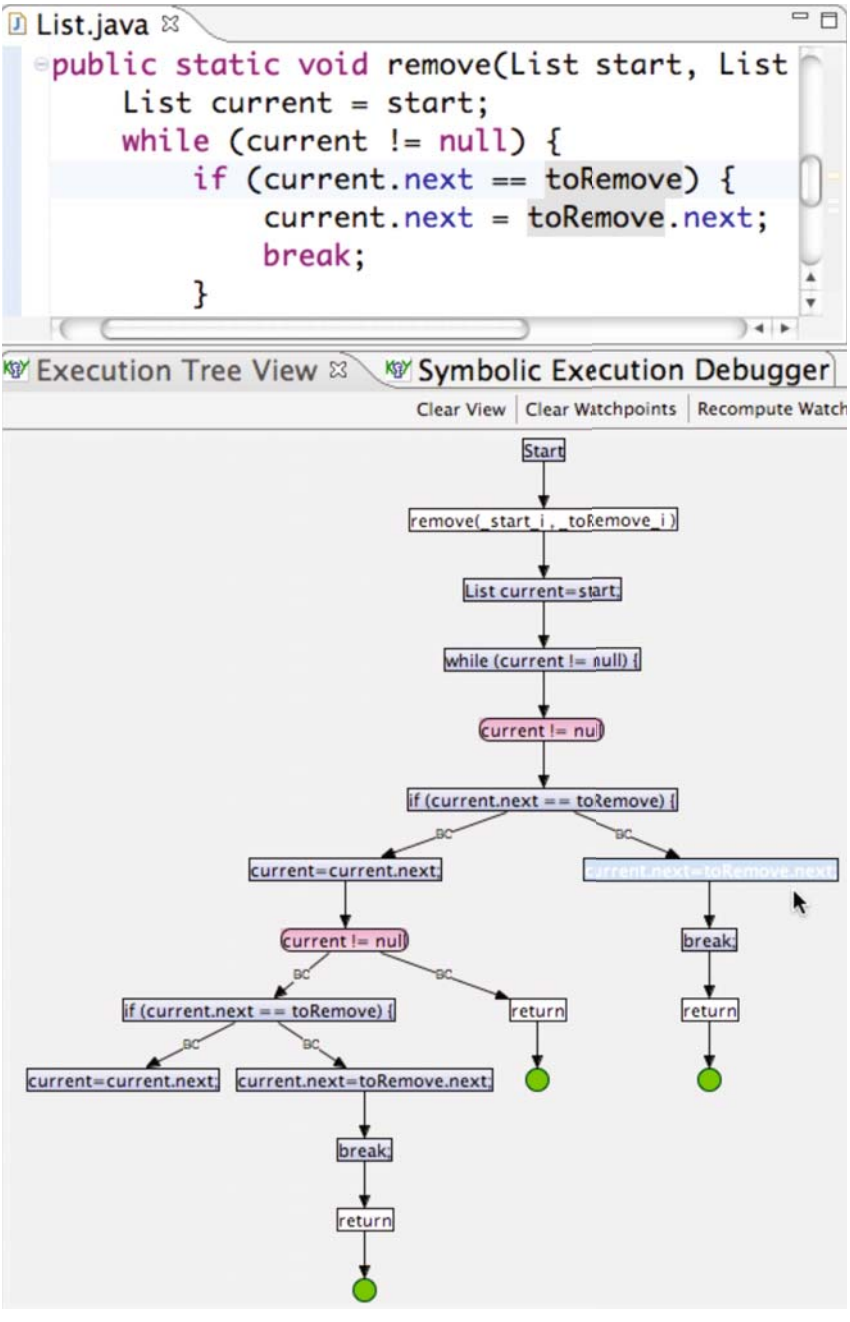
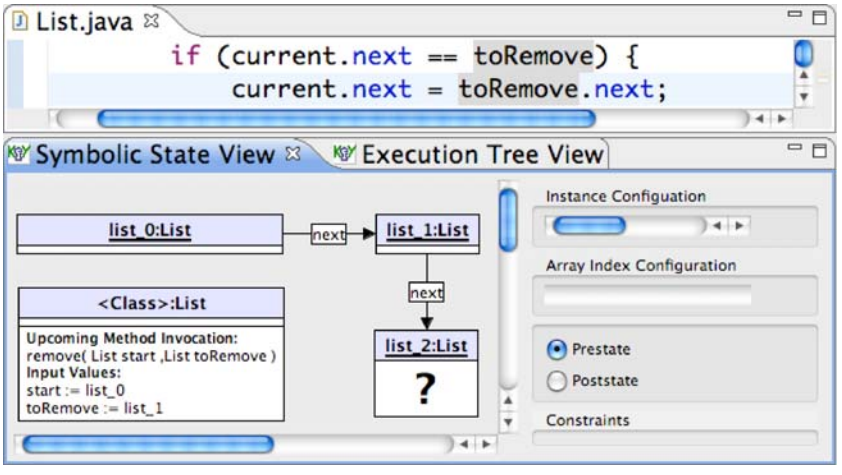
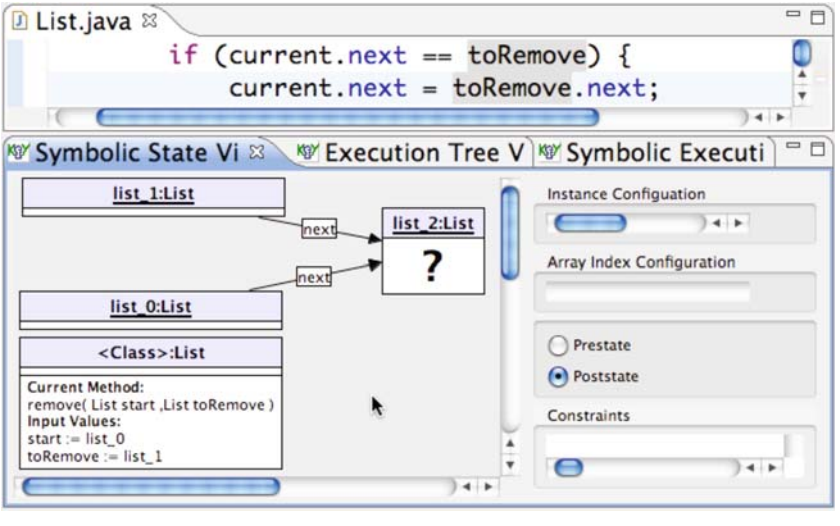


Fig. 2. Execution tree view of a prototypical Visual Debugger exemplified by a removal method for elements in a linked list. The currently executed statement is highlighted both in the editor and in the symbolic execution tree.



(a) Symbolic state before removal of list element `list_1`



(b) Symbolic state after removal of list element `list_1` (`list_1` is no longer reachable from the list head `list_0`)

Fig. 3. Object diagram view of a prototypical Visual Debugger

4.4 The KeY-Hoare Tool

As detailed in Section 3 the final teaching unit focuses on the most rigorous approach to software validation, namely, formal verification of programs.

For many students this is actually the first time that they are introduced to the idea that programs can be proved correct. This is both a chance and a risk at the same time. If done well, one has the rare opportunity to meet open, unprejudiced minds. But if not done with care, formal verification ends up being stowed away as a useless academic pastime.

The authors have experienced more than once that even active researchers in other fields of computer science are unaware of the progress achieved in formal verification during the last 15 years, but still hanging on the impression of the area from the early eighties. Instead we claim that:

Formal methods are applicable in the real world.

Formal methods are part of the software development tool chain.

These are the (not so) subliminal messages we want to pass on to the students. Consequently, a hands-on part where students verify their own programs is crucial.

In the lectures we present formal verification as the natural next step coming at an acceptable cost after a partial formal specification is already in place. At this point in the course students have had already seen (in the lecture) and written (exercises and lab courses) a number of small programs annotated in a formal specification language. In particular, they have experienced the advantages of formal specifications as a prerequisite for

- automated test case generation: in contrast to hand-written unit test cases the resulting tests are guaranteed to satisfy formal criteria such as branch coverage, coverage of logical conditions, etc.
- systematic debugging using a symbolic debugger with extended functionality not achievable with standard debuggers.

In this context, presenting a Hoare-style calculus as a mere pen-and-paper version is unconvincing and obscures the message to be transported. Instead it is important to provide access to an easily usable (in particular, easily installable and documented) verification tool that provides a reasonable level of automation. We would even go as far as to say that *formal verification without mechanisation and automation is pointless.*

We had the following requirements on the tool that would be used in the lectures:

1. The calculus implemented by the tool should be a Hoare- or Dijkstra-like calculus. This eases the orientation for interested students who want to read additional material in advanced textbooks.
2. The supported programming language should be
   - *simple*, because teaching a calculus for a programming language like JAVA is obviously not feasible in an introductory course such as TDV;

   - *imperative*, because most students are more familiar with imperative languages and imperative languages are the most commonly used.
3. The generation of verification conditions, respectively, weakest precondition computation should be transparent and presented in detail to the students, while first-order reasoning on program-free expressions should be treated as black-box.
4. Reasonable automation: writing a correct specification, in particular, finding the right loop invariants, is already hard for students. If a valid proof obligation cannot be proven automatically, it is frustrating and not very convincing. Of course, we cannot overcome theoretical limitations, but for the kind of programs given to the students, the prover must be able to close virtually all occurring valid first-order verification conditions. In other words, if a proof obligation cannot be shown, then this should indicate that the program is erroneous, or the specification wrong or too weak. For complex programs it is already hard to find out whether the error is in the specification or in the program.

We were surprised that no tool existed which satisfied our requirements. Tools tailored towards program verification were designed for complex target languages like JAVA going beyond the scope of the lecture, others required extensive introduction into the underlying proof assistant or completely lacked automation. Finally, we decided to adapt our own program verification tool KeY [4]. We decided against using unaltered KeY in the specific context of the TDV course, because it is targeted towards full JAVA and uses a sequent calculus in dynamic logic which we thought is not "mainstream" enough.

The developed variant of KeY, called KeY-Hoare[8] [7], is implemented on top of the standard KeY-tool. It has a simple imperative programming language with arrays as target language. The specification language is a standard sorted first-order logic with arithmetic. The latter's concrete syntax is almost identical to JML with which the students are already familiar. The implemented Hoare-style calculus [7] has some other characteristics which make it suitable for an introductory course:

- The calculus is based on symbolic execution following the control flow of the program. Approaching verification condition generation from the viewpoint of a (symbolic) interpreter eases the topic considerably for the students providing a bridge between new ideas and existing knowledge.
- The calculus is computational and resembles in this aspect Dijkstra's weakest precondition calculus. The only non-deterministic verification condition generation rule is the loop invariant rule where user interaction is needed.
- First-order reasoning can be treated as black-box. As we built on top of the standard KeY-tool, we could reuse its state-of-art first-order prover with support for linear and non-linear arithmetic. For the kind of problems encountered by students in exercises and lab courses the prover works in almost every case fully automatic, i.e., typically a goal that cannot be closed indicates an invalid proof obligation.

[8] `http://www.key-project.org/download/hoare` (GPL)

The prover provides a user-friendly, self-explaining point-and-click GUI. It can be installed using JAVA webstart technology with one click on all standard architectures (Linux, Windows, MacOS).

Finally, using a tool to teach formal program verification helps also the teacher as it avoids common mistakes found in typical lecture notes where given specifications (in particular, loop invariants) are too weak to be actually proven. Tool support saves also time spend on supervision as students need less supervision than for pen-and-paper proofs. Mechanisation is also important in grading, because soundness of the verification tool ensures that completed proofs are correct. KeY-Hoare produces proof certificates that can be loaded, inspected, and verified by the teacher to avoid fraud.

5 Experiences and Discussion

History. The TDV course has been taught in its present form two times, starting in Fall of 2007. The course goes back to a course called *Program Verification* that was supposed to be a lightweight introduction to formal verification techniques (without tool support). The course had attracted relatively few students and used to be on the borderline of being economically feasible. In an attempt to make it more attractive to a broader audience, we expanded significantly on the testing topic, and included debugging for the first time, thereby redesigning it radically using the concepts outlined in this paper. As this happened only during Summer 2007, the course information given to students still referred to the old structure, and interest was not very high (18 registrations). We were encouraged, however, by the fact that not a single student dropped out. In 2008, the course was announced with the new title and content. We also took great care to explain the intended goals and concept. Nevertheless, we were surprised that registration jumped up to ca. 80 participants. This shows that the title and content description of a course, as well as the available information, can have considerable impact on registration figures.

Course Evaluation. It is fair to say that the two rounds which the course had so far got very positive reactions from the students. This is documented by course evaluation protocols and a web questionnaire. In particular, the students valued highly the relevance of the overall course, and highlighted the impact of the hand-in assignments for their learning. In 2008, 90% of all students who started the course tried to complete it successfully (i.e., participated in all compulsory exercises and sat the exam). As the course is not compulsory in any programme, this points to a relatively high degree of motivation that students take from this course.

Limitations. We do not want to conceal limitations of the current course concept. Trying to find the right balance, it constitutes in its current form a compromise between available time, range and depth of topics. Regarding the available time, we cover already relatively many topics and it is not possible to treat some of

them in the depth we would like to. On the other hand, we had already to cut down on the number of topics we would like to treat as, for example, code reviews or software certification. One important topic that needs to be still addressed is to integrate the taught techniques into a software development process.

The hands-on approach using mainly JML and JAVA means that we do not deal with abstraction which is obviously a very important topic. Partly, this stems from our conviction that currently available formalised abstraction techniques are not suitable for our goals: refinement-driven approaches such as B [1] are simply too heavyweight for what we have in mind; model-driven approaches based on UML, on the other hand, are too far removed from popular implementation languages.

Given the variety of topics, we could not live up to the ambition to let the students practise the acquired knowledge in *all* the topics covered by the course. Even if we have weekly exercises covering all the material, the three extensive hand-in assignments cover only the topics testing, specification and verification, not the topics debugging or test generation. Naturally, the students performed better in those parts of the exam which related to either of the hand-in assignments.

In future courses we plan to address some of the mentioned limitations. We consider an integration of formal methods into a viable development process as elementary. Therefore we will define an agile software process that makes use of formal techniques: agile processes have explicit test generation and debugging phases in each increment. Their cyclic nature can potentially benefit a lot from automation. The challenge is to integrate formal specification in the right way. As a follow-up to the TDV course one could then run a project course based on an agile process with formal techniques.

Despite stressing automation throughout the course, grading of practical exercises should use more automation. We did not yet fully realise the potential of formalisation there.

Relation to Dedicated Formal Methods and Logic Courses. The TDV course covers a variation of topics, among them some lightweight formal methods. Towards the latter, we take an extremely pragmatic stance. Perhaps our most controversial design decision is to dispense with any form of rigorous mathematical semantics. There are several potential problems with this: seduction to cut-and-paste without real understanding, fostering misconceptions caused by ambiguity, frustration for theoretically interested students who want to know the "nuts and bolts". We believe that the advantages (accessibility for mathematically untrained people, applicability of learnt methods) outweigh the problems, given the course is designed for the (late) Bachelor level.

Still, students should be given the opportunity to acquire more in depth knowledge and understanding in logical methods for computer science, in software verification, and furthermore in hardware verification. At Chalmers, these three areas are covered by the Masters-level courses "*Logic in Computer Science*", "*Software Engineering using Formal Methods*" (SEFM), and "*Hardware Description and*

Verification" [2]. The SEFM course[9] is also given by the authors of this paper, and covers software model checking (with Spin/Promela, based on [5]) as well as deductive software verification (of JAVA with KeY).

Research-Driven Course Development. The authors of this paper constitute the senior staff of the research group "Software Engineering using Formal Methods" at the Department of Computer Science and Engineering, Chalmers University. The group is carrying out research in formal modeling and verification of software as well as verification-based testing and debugging. Together with groups at the universities of Karlsruhe and Koblenz, we have developed the KeY approach and system for JAVA source code verification. This research and the corresponding tool development were the basis for developing both the TDV course and the SEFM course. The close relation to research does *not* contradict the fact that TDV targets an audience of Bachelor students with little theoretical prerequisites. On the contrary, this is a good match, as the objective of our research is precisely the increased accessibility of formal methods to software developers. We believe that the students profit from this overall objective. In the opposite direction, our research profits very much from these courses. The usage of cutting edge research tools such as code-based ATCG, formal verification tools, or the symbolic visual debugger, in the course context increases the pressure on usability. The concrete feedback from students and course assistants drives the further design and development of these tools.

Adaptation. While the basic development of the course as outlined in this paper would not have been possible without relying on the research in the group, we claim that the resulting course can be run in any other context. We actively support adaptation of this course (or individual modules of it) and provide the complete sources for slides, examples, and assignments to interested teachers. The course has been adapted (or is planned to) at several European universities, including Technical University of Madrid, University of Innsbruck, and University of Freiburg.

Of these, we would like to mention in particular the adaptation of the KeY-Hoare tool done by Joanna Chimiak-Opoka at University of Innsbruck/Austria. She gave ample feedback and contributed a well-organised collection of additional examples. Her comments and suggestions lead to the integration of several originally not considered features of KeY-Hoare, for example worst-case execution time analysis of programs. This strengthen our opinion that adaptation works in two directions and is not a one-way street.

Acknowledgements

We want to express our gratitude to the people without whom a course such as TDV would not have been possible. We thank Fredrik Lindblad for his commitment in co-teaching the TDV course, Christian Engel for implementing the

[9] Course Code TDA292, `http://www.cse.chalmers.se/edu/course/TDA292/`

KeY VBT tool and Marcus Baum as well as Marcel Rothe for implementing the visual debugger. Special thanks go to Joanna Chimiak-Opoka for using KeY-Hoare in her lecture as well as providing examples and continuous feedback that was an invaluable help and motivation to improve the KeY-Hoare tool. A teaching grant from the Chalmers Masters programme in *Software Engineering and Technology* is gratefully acknowledged. Finally, we thank all our students for their active participation making the course a pleasure to teach. We thank also the anonymous reviewers for their valuable comments.

References

1. Abrial, J.-R.: The B Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Axelsson, E., Björk, M., Sheeran, M.: Teaching hardware description and verification. In: International Conference on Microelectronics Systems Education, Anaheim, CA, USA, pp. 119–120. IEEE Computer Society, Los Alamitos (2005)
3. Baum, M.: Debugging by visualizing of symbolic execution. Master's thesis, Department of Computer Science, Institute for Theoretical Computer Science (June 2007)
4. Beckert, B., Hähnle, R., Schmitt, P.: Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
5. Ben-Ari, M.: Principles of the Spin Model Checker. Springer, Heidelberg (2008)
6. Bjørner, D.: Software Engineering, vol. 3. Springer, Heidelberg (2006)
7. Bubel, R., Hähnle, R.: A Hoare-style calculus with explicit state updates. In: Instenes, Z. (ed.) Proc. Formal Methods in Computer Science Education (FORMED). ENTCS, pp. 49–60. Elsevier, Amsterdam (2008)
8. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: The JML and JUnit way. In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 231–255. Springer, Heidelberg (2002)
9. Cok, D.R., Kiniry, J.: ESC/Java2: Uniting ESC/Java and JML. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 108–128. Springer, Heidelberg (2005)
10. de Halleux, J., Tillmann, N.: Parameterized unit testing with Pex. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 171–181. Springer, Heidelberg (2008)
11. Engel, C.: Verification based test case generation. Master's thesis, Department of Computer Science, University of Karlsruhe (August 2006)
12. Engel, C., Hähnle, R.: Generating unit tests from formal proofs. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 169–188. Springer, Heidelberg (2007)
13. Guttag, J.V., Horning, J.J., Garland, S.J., Jones, K.D., Modet, A., Wing, J.M.: Larch: Languages and Tools for Formal Specification. Springer, New York (1993)
14. King, J.C.: A program verifier. PhD thesis, Carnegie-Mellon University (1969)
15. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06y, Iowa State University, Department of Computer Science (2003) (revised, June 2004)

16. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P.: JML Reference Manual (February 2007); Draft revision 1.200
17. Lewis, B.: Debugging backwards in time. In: Ronsse, M. (ed.) Proc. Fifth Int. Workshop on Automated and Algorithmic Debugging, AADEBUG (September 2003)
18. Meyer, B.: Applying "design by contract". IEEE Computer 25(10), 40–51 (1992)
19. Rothe, M.: Assisting the understanding of program behavior by using symbolic execution. Master's thesis, Department of Computer Science and Engineering (July 2008)
20. Sommerville, I.: Software Engineering, 8th edn. Addison-Wesley, Reading (2006)
21. Zeller, A.: Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann, San Francisco (2005)
22. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering 28 (2002)

On Teaching Formal Methods: Behavior Models and Code Analysis*

Jan Kofroň[1,2], Pavel Parízek[1], and Ondřej Šerý[1]

[1] Charles University in Prague,
Department of Software Engineering Malostranské náměstí 25,
118 00 Prague 1, Czech Republic
{kofron,parizek,sery}@dsrg.mff.cuni.cz
http://dsrg.mff.cuni.cz
[2] Academy of Sciences of the Czech Republic,
Institute of Computer Science Pod Vodárenskou věží 2,
182 07 Prague 8, Czech Republic
kofron@cs.cas.cz
http://www.cs.cas.cz

Abstract. Teaching formal methods is a challenging task for several reasons. First, both the state-of-the-art knowledge and the tools are rapidly evolving. Second, there are no comprehensive textbooks covering certain topics, especially code analysis. In this paper, we share our experience with teaching two courses. The first is focused on classics of modeling and verification of software and hardware systems (LTS, LTL, equivalences, etc.), while the other one involves topics related to automated analysis of program code. We hope that other lecturers can benefit from our experience to improve their courses.

1 Introduction

For a developer of a system with high demand on reliability (e.g., safety-critical systems and device drivers), at least a basic insight into formal methods is essential. In particular, familiarity with model checking and code verification techniques and tools is an important asset. First, it is useful when actually dealing with the tools, which often communicate in specialized formalisms (e.g., various temporal logics). Second, it helps developers to decide whether they can benefit from a concrete technique or tool and also to choose among different implementations to suit their specific needs. The latter point is especially important with the increasing number of available code analysis tools that are ready for industrial use, at least in specialized domains (e.g., Slam [8]). The underlying techniques have different strengths and limitations, which is very hard to assess without a deeper insight.

Well targeted formal methods education of the future software developers is very important, but also very intricate. In order to fully understand the topics, quite deep mathematical background (e.g., in logics, algebra, and automata

* This work was partially supported by the Ministry of Education of the Czech Republic (grant MSM0021620838) and by the Czech Academy of Sciences project 1ET400300504.

J. Gibbons and J.N. Oliveira (Eds.): TFM 2009, LNCS 5846, pp. 144–157, 2009.
© Springer-Verlag Berlin Heidelberg 2009

theory) is required. This is in contrast to the current trend of a slow decrease in the amount of mathematical theory taught in favor of software development practice. As a result, building a formal methods course based on the limited foundations is very challenging.

Another obstacle is lack of literature. When it comes to general modeling of systems and model checking, *Model checking* by Clarke et al. [24] forms an excellent basis for a course. Unfortunately, the situation is not so good in the case of code analysis (as applied in tools like Java PathFinder [4] and Blast [2]). This topic is relatively new and still rapidly evolving. To the best of our knowledge, there is no comprehensive publication summarizing and comparing the different techniques so far. This means that a course has to be based mainly on various journal and conference publications and technical reports. These publications are rather brief, as the page range is typically limited. They also often differ in the level of abstraction and the underlying formalization and notation. Except for the additional preparation overhead, it is not an issue for the lecturers; however, such a form of study materials constitutes a major obstacle for the students.

In this paper, we share our experience with teaching formal methods in the scope of a new master study plan, *Dependable Systems* (Sect. 2). Among other courses, the plan contains two one-semester courses on formal methods that together cover modeling systems, model checking, code verification, deductive methods, and static analysis. The common goal of the courses (and the study plan in general) is balancing the theoretical and practical skills of the students. In Sections 3 and 4, the topics covered by the two courses are summarized in more detail along with the main references to study materials and the list of tools the students get acquainted with. We believe that other lecturers preparing similar courses will benefit from our experience. Sect. 5 contains observations and points to be discussed by the formal methods teaching community.

2 Our Vision and Realization

In 2008, a new study plan for master studies, *Dependable Systems*, emerged in our department as a reaction to both increasing industrial demand for highly specialized software experts and differentiation in their expected knowledge. The motivation of the new study plan is to provide industry with graduates familiar with techniques necessary to develop dependable systems (e.g., embedded, safety-critical, and real-time systems).

On one hand, this includes rather low-level knowledge of system architectures, operating systems, middleware, real-time systems, embedded systems, and parallel computing. On the other hand, the graduates get insight into software architectures, component systems, and services. Of course, courses on formal methods are a natural part of this study plan as well.

In general, the Dependable Systems study plan provides students with both the theoretical foundations and the practical hands-on experience with different tools and development techniques for more exotic platforms (e.g., embedded devices) and development under specific conditions (e.g., real-time, limited memory).

Considering formal methods, there are two specialized lectures: *Behavior Models and Verification* (Sect. 3) and *Program Analysis and Code Verification* (Sect. 4). The former covers modeling and model checking of software and hardware systems. The latter specializes on techniques for direct code analysis. Originally, there was only the former course covering also few code model checking topics. However, the amount of information associated with recent advances in code model checking and success of projects like SLAM[8] in practice simply did not fit into a single course and motivated us to separate the course into the two specialized courses.

3 Behavior Models and Verification (NSWI101)

The course *Behavior Models and Verification* [14] aims at providing basics of the behavior modeling of systems and their consequent verification. The attendees of the course, future developers, should learn about the principles of formal specification and verification as well as work with state-of-the-art tools that are used in industry for verification of hardware and software models nowadays. Since this is an introductory course for master's level students, we do not assume any special knowledge in this area in addition to what they have learned during their bachelor's level studies. In particular, this includes propositional and predicate logics, and the automata theory. After passing the course, the students should be able to construct a formal behavior specification of a simple hardware/software system, think of and specify properties of interest, and, eventually, verify these properties using available tools. As to the organization, there are a lecture and a lab every week.

3.1 Lectures

In Fig. 1, the topics covered by the lectures are depicted; the main body includes the following:

- **Basic concepts.** LTS, Kripke structure, and different preorder and equivalence relations
- **Temporal logics.** Syntax, semantics, and expressive power of LTL, CTL, and CTL*
- **Model checking algorithms.** Both explicit (for LTL and CTL) and symbolic (for CTL) based on OBBDs
- **Partial order reduction.** The Ample set algorithm

This part of the course is motivated mainly by the comprehensive book *Model checking* by Clarke et al. [24]. A suitable level of abstraction is maintained throughout the book, which makes it also a useful study material for the attendees.

The rest of the lectures introduce timed automata and process algebras. The overview of timed automata is motivated by [16] and presents the basic properties

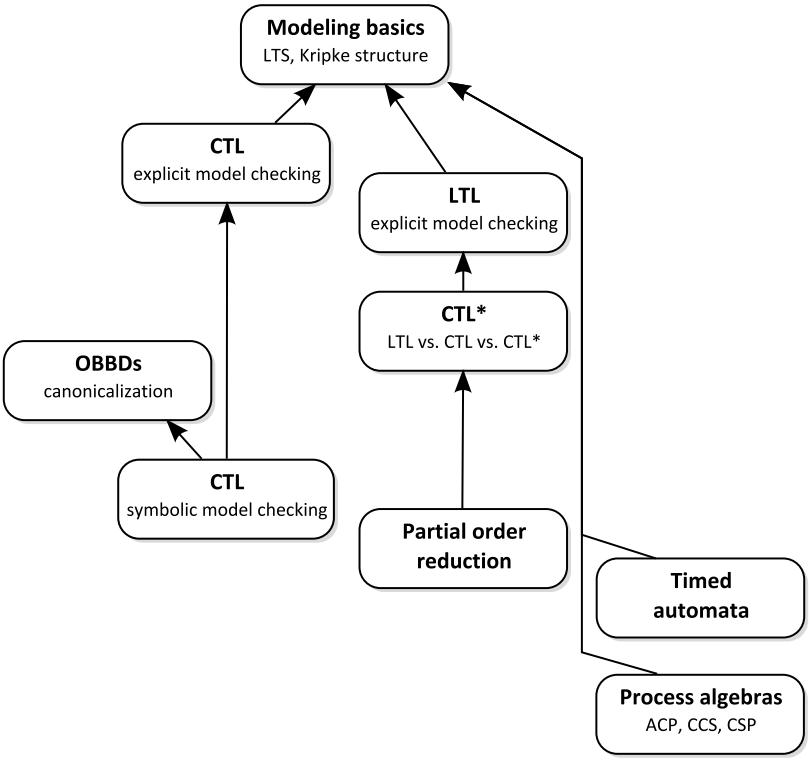


Fig. 1. Topics covered by the NSWI101 course and their dependencies. The corresponding lectures are held in the top-down order.

of the class of timed regular languages, emptiness check algorithm, parallel composition, and references the Uppaal integrated environment [12]. The lectures on process algebras focuses on *Algebra of Communicating Processes* (ACP) [21] and its content is highly inspired by the book *Introduction to Process Algebra* by Fokkink [26]. As an example of a relatively recent application, the formalism for behavior specification of software components, *Behavior Protocols*, is also presented based on [15].

3.2 Lab

There are two major aims of the lab of the NSWI101 course—first, the students should practically exercise the algorithms and techniques presented during the lectures, and, second, the model checking tools are presented and their input languages are discussed in detail.

The first three labs are devoted to the Spin model checker [11] and its input language Promela. We use the slides from the official Spin website [38,39], which turned out to be very good for first understanding of the basic modeling

concepts. The presentation of the language is divided into two parts. After each part, the students solve simple assignments during the lab, such as the modeling of the producer–consumer problem. The goal of these labs is to cover almost the entire language. The principles of model simulation and verification are also presented as well as the majority of the options (command-line switches), however, the details on implementation of the algorithms inside Spin are mentioned just briefly or entirely skipped. The tool is demonstrated using both command-line and graphical user interfaces, whose options and settings are briefly explained. For more information, the students are pointed to the complete slide sets and the Holzmann book on Spin [30].

The subsequent three labs are devoted to exercises of the techniques and algorithms presented in the lectures. This includes in particular modeling simple systems via LTS, deciding on equivalences of temporal logics formulae, representing formulae, sets, and Kripke structures via OBDDs, and LTL and CTL model checking algorithms. After the lectures introducing OBDDs and algorithms for symbolic CTL model checking, a lab focusing on SMV model checker, in particular NuSMV [5], is held. The tool along with the parallel assignment language is introduced and the students again try to model simple systems (e.g., dining philosophers and the producer-consumer problem) to become familiar with the tool.

The rest of the labs is again devoted to exercises related to the theory presented in the lectures. To provide an opportunity to work on homework assignments, there are two to three labs left out at the end of the semester.

There are two graded homeworks; the grading forms 55% of the final grade, while the rest, i.e., 45%, is formed by the grade a student gets from the final test. The first homework is assigned at the end of the fourth lab. The assignment is articulated in a very general way, such as "model a railway station" and "model an airport". This way turned out to be beneficial from several points of view. First, it is easy to reveal a potential disallowed collaboration of the students—such a general assignment is very unlikely to be "implemented" similarly in multiple cases. Second, the students are forced to think of suitable abstractions to be used to create the models. Third, they have to think of properties to check—this is very important according to us, since in most papers and text books, usually only deadlocks and in better cases also response patterns are considered. We believe, however, that it is important to verify specific properties that can be a matter of interest in particular cases. Nevertheless, the students are provided with examples of entities that can be modeled, the properties that can be checked, and ways to make their models simpler if they reach the limits of Spin, usually in the sense of the size of their state spaces. The maximum amount of points a student can normally get for the first homework is 40. However, if a student creates an exceptional model, he or she can get up to 5 extra points.

At the end of the seventh lab (slightly after the middle of the semester), the second homework is assigned. It is aimed at practicing the NuSMV tool [5]. Since the parallel assignment language is rather low level in comparison with Promela, we decided for a simpler assignment, in particular, modeling and verification of

properties of a well-known algorithm, e.g., the Dekker's algorithm for mutual exclusion [25]. Because of the lower complexity of the second assignment, the maximum amount of points in the case of the second homework is 15.

3.3 Grading

The grade for the course is based on points. We award 0–40 and 0–15 points for the first and second homework, respectively, and 0–45 points for the written exam; the total number of points is therefore 100. The grading scale is defined as follows:

- Score of 80–100 points corresponds to the *excellent* grade.
- Score of 71–79 points corresponds to the *very good* grade.
- Score of 62–70 points corresponds to the *good* grade.
- Score of 0–61 points corresponds to the failure, i.e., to an unsuccessful attempt to complete the course.

The grading scale is defined to force students to do both homeworks and the written exam that is devoted to theoretical background, principles, and important algorithms; it is not possible to do just the homeworks or the exam to complete the course.

There are soft deadlines for both homeworks—after a deadline passes, for each day of delay, there is a penalty of 10% of the points awarded.

3.4 Experience

Originally, there was a lecture every week, while the lab was held every other week only, i.e., there were six labs during the semester. They were entirely focused on the tools and their input languages (Spin, NuSMV, Bandera). After two years, we realized that the students are quite able to construct models in the sense of using a specification/modeling language and corresponding tools to verify their properties, however, they did not capture the algorithms and underlying theories well. This is mainly due to the rather demanding amount of theory. Therefore, we decided to extend the course and have the lab every week to practice it.

The aforementioned fact that the students were able to create models and use the tools deserves more explanation here. Since for most of the students, this was the very first experience with behavior modeling, which differs from ordinary implementation, indeed, several problems occurred. According to the complexity of the models they submitted as solutions to the homework assignment (taking just the first Promela assignment into account), the students could be divided into two groups. The students of the first group submitted a sort of simplistic models which can be verified by Spin in a reasonable amount of time (in the order of minutes on a decent machine), while the others ran into difficulties with verification due to complexity of their models. The unfortunate conclusion of some of them was then that Promela is not a suitable modeling language, and

that behavior modeling in general does not make much sense. After getting this kind of feedback, we have started to emphasize the goals and success stories of modeling in general and focused more on guidelines on how the models should be constructed, especially choosing an appropriate level of abstraction.

As to the organization of the course, after first two years, we have decided to teach this course in English, which is not a native language of the students. There were several motivations for doing so. First, the majority of the terms have just an English form and there are no widely accepted translations. Second, since the English-language skills are generally not on a proper level in our country, the students can benefit from knowledge of the English language at conferences and workshops in the future. Even though there was a significant drop in the number of students attending the course after switching to English, all the students attending the courses so far have given us a positive feedback on this issue.

4 Program Analysis and Code Verification (NSWI132)

While the course *Behavior Models and Verification* (Sect. 3) focuses on general behavior modeling of software and hardware systems and checking of various properties of the models, in the course *Program Analysis and Code Verification* [34] we focus on analysis and verification of programs in mainstream languages like Java and C. The goals of the course are twofold:

1. to show the students, future software developers, that there exist tools for formal verification and analysis of programs that can discover real bugs (errors) in non-trivial programs and/or verify many interesting properties in the programs, and to let students gain experience with usage of the tools;
2. to provide the students with basic knowledge of key approaches to program analysis and verification, and of advantages, challenges, and limitations associated with each approach.

Our vision is that students attending the course should be able to use the appropriate methods and tools during the software development process.

We do not expect any specific prior knowledge from the students—we only assume that all students have basic knowledge of the automata theory and predicate logic. Since the course aims at master's level students, and courses on the automata theory and logic are taught at the bachelor's level, all students should have the required knowledge. Moreover, we do not strictly require that students complete the course on *Behavior Models and Verification* before participating in this course.

As for organization, the course run in the winter of 2008 for the first time. There was a lecture every week and a lab every other week. We give the students three homework assignments as a part of the course.

4.1 Lectures

The goal of the lectures is to introduce and describe the main approaches to verification and analysis of programs (code). We divided all the lectures (Fig. 2)

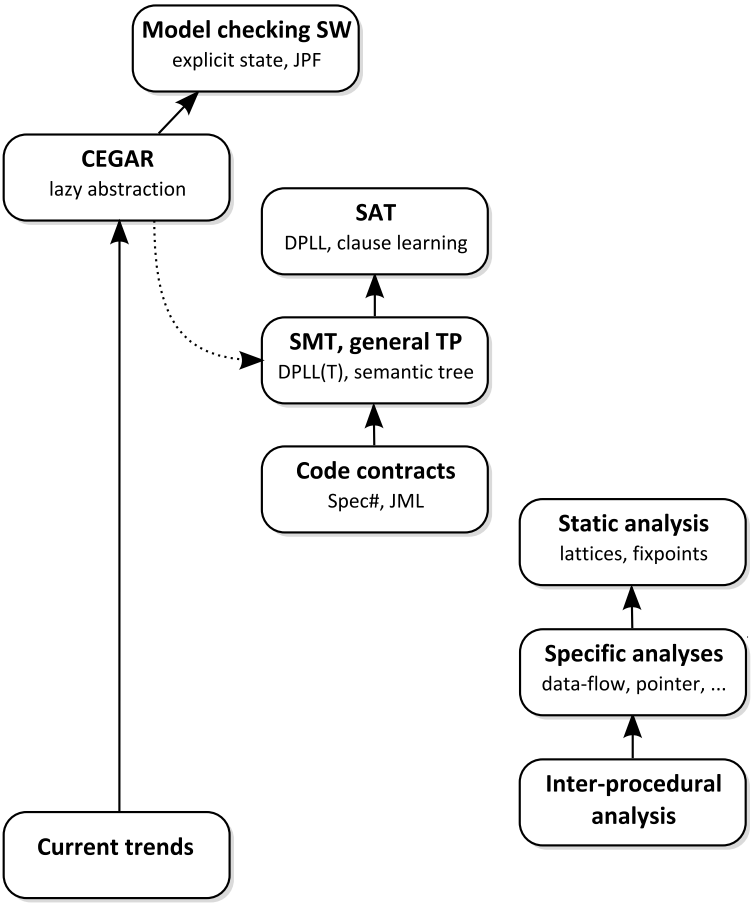


Fig. 2. Topics covered by the NSWI132 course and their dependencies. The corresponding lectures are held in the top-down order.

into four blocks: *Program Model Checking*, *Deductive Methods*, *Static Analysis*, and *Current Trends*. In the lectures forming each block, we describe theoretical background (e.g., lattices and fixed points for static analysis) of the approach, the basic principles and concepts of the approach (e.g., state space traversal in case of program model checking), and main limitations and challenges associated with use of the approach (e.g., state explosion) together with some solutions to the challenges (e.g., POR and symmetries for state explosion).

Program Model Checking comprises lectures on both explicit-state program model checking and CEGAR-based algorithms. The explicite-state program model checkers are explained on the example of Java PathFinder [4].

The lectures are based on a related paper [41] describing the algorithms Java PathFinder uses for POR, efficient state caching, etc.

The CEGAR-based algorithms are explained on the examples of Slam [8], SatAbs [7], and Blast [2]. The lectures cover predicate abstraction, abstraction refinement loop, and lazy abstraction; they are based on the papers [18,22,37] and nice tutorial slides on lazy abstraction [29]. Note that, before diving into the theorem proving details in the next lectures, the theorem prover is used here as a black-box with emphasis on the type of questions it is able to answer.

Deductive Methods and their application in program verification. In this block, we provide an overview of the main techniques used in SAT solvers, SMT solvers, and general theorem provers. The lectures are inspired mainly by the books *Decision Procedures: An Algorithmic Point of View* by Kroening and Strichman [31], and *Automated Theorem Proving: Theory and Practice* by Newborn [32]. Additional information on SAT solvers was taken from a nice survey [42].

In the subsequent lectures, we present application of solvers in the contract-verification frameworks like ESC/Java2 for JML [3] and Boogie for Spec# programs [10]. Here, the most useful information sources are the papers that describe algorithms used in Boogie [20,19].

Static Analysis block contains methods that are based on the lattice theory and computation of fixed points. This includes traditional data-flow analyses, points-to and shape analysis, and also a brief introduction to a control-flow analysis. The structure and content of this block of lectures is to a great extent based on the lecture notes [40]. A more thorough source is the book *Principles of Program Analysis* by Nielson et al. [33], however, we found it a bit too formal for use in an overview course.

Current Trends in formal analysis and verification of programs are summarized in the last block. Here, we provide an overview of the very recent topics based on various conference papers. The topics include compositional verification using assume-guarantee reasoning [27], symbolic execution in Java PathFinder [36,17], and a combination of testing with predicate abstraction [35,28,23].

4.2 Lab

The main goal of the labs is to provide the students with a hands-on experience with selected tools for verification and analysis of programs (code).

Each lab is devoted to a specific tool—for example, Java PathFinder and Soot framework [9]. First we explain how a tool works, how it can be configured and executed, and how to prepare its input and interpret its output—all this on simple demo programs and examples. Then, in the second part of a lab, we assign some simple tasks to the students, so that they can get their own experience with using the tools (and "playing" with them).

We present a single tool for each main technique (method) that is described in the lectures. To be more specific, we present the following tools:

- Java PathFinder [4], an explicit-state model checker for Java programs,
- Blast model checker [2], an implementation of the CEGAR-based model checking algorithm,
- SatAbs [7], model checker for C programs that uses CEGAR and a SAT solver,
- PicoSAT [6], a state-of-the-art SAT solver,
- Yices [13], a state-of-the-art SMT solver,
- ESC/Java2 [3], a tool for verification of Java programs against JML specifications [1], and
- Soot [9], a framework for static analysis and transformation of Java programs.

We have selected these tools due to their maturity and stability, moreover they are widely used, and open source.

The homework assignments directly follow the labs. Our motivation behind the homeworks is to let students try the tools on a larger example (program) than it is possible during a lab. There are three homeworks together. The theme of the first homework is Java PathFinder—the students are required to create a reasonable abstract environment for an open system and also to create a custom property. In case of the second homework, students are required to create a JML specification for several Java classes and to verify the classes' implementation against the specification. Finally, the third homework consists of creating custom analysis and transformation of Java source code on top of the Soot framework. The time needed for each homework is between 8 and 16 hours, depending on student's skills.

4.3 Grading

The grade for the course is based on points. We award 0–10 points for each homework and 0–30 points for the oral exam; the total number of points is therefore 60. The grading scale is defined as follows:

- Score of 49–60 points corresponds to the *excellent* grade.
- Score of 40–48 points corresponds to the *very good* grade.
- Score of 31–39 points corresponds to the *good* grade.
- Score of 0–30 points corresponds to the failure, i.e., to an unsuccessful attempt to complete the course.

We have defined such a grading scale in order to force students to do both homeworks, which are about practical use of the tools, and oral exams that is devoted to theoretical background, basic principles of the approaches and important algorithms. In particular, it is not possible to do solely the homeworks or solely the exam to complete the course.

4.4 Experience

After the first year, our experience with the course is somewhat mixed. On the one hand, the students were interested in the discussed topics and we were very satisfied with the quality of students' solutions to the homework assignments.

On the other hand, we found that having a lab only once per two weeks is not enough, similarly to the other course. For the upcoming years, we plan to have a lab every week. Some of the labs will focus on manual computation of the key algorithms using paper and blackboard, while the others focus on practical experience with the tools.

5 Evaluation and Discussion

The common issue of both courses is a low number of students attending the courses. We believe that the low attendance (enrollment) of students has the following two main causes:

- The usefulness of formal methods in industrial software development is not obvious to the students. They probably do not see the benefit of formal verification and analysis in comparison to testing. Moreover, formal methods are rarely used in software companies and therefore the students are not forced to learn about them. The students prefer to attend those courses, which they see as useful for their employment (XML, software engineering, web development, etc.).
- Courses on formal methods typically require significant mathematical background (logics, automata theory, formal languages, etc.) and they are also typically more demanding than the courses on XML and web development. Since most students prefer to choose the easier way to get the degree, they tend to avoid mathematics as much as possible.

While we see these two causes as general, they may be specific to our university to a certain extent.

Another problem is the lack of literature about formal methods at the level of master's level studies. There are many books and research papers that could be used, however they are often aiming at PhD students and researchers. This is especially problematic in the course *Program Analysis and Code Verification*, since we are not aware of any comprehensive book on code analysis and verification—e.g., with an extent and coverage similar to [24], which we use in the course *Behavior Models and Verification*.

As for the structure and syllabus of the courses, as the greatest benefit for students, we see the possibility to get hands-on experience with the tools and see their advantages and limitations, since they will not have such an opportunity in industry. Nevertheless, they are not always able to assess the complexity of models (or programs) with respect to formal analysis and verification, even after completing the courses—this requires years of experience with the practical application of formal methods.

6 Conclusion

In this paper, we have shared our experience with teaching formal methods in the scope of a new study plan, *Dependable Systems*. We have presented the content of two courses—"Behavior Models and Verification" and "Program Analysis and Code Verification". While the former one is probably similar to courses at other universities regarding its structure and content, we believe that a reader will benefit from our experience with the latter one. We have structured the code analysis course in a way to provide the students with experience with more tools rather than introducing few tools in greater depth.

Interestingly enough, the number of students attending the two courses is rather low in comparison to practically-oriented software engineering courses. Having a positive feedback from our students on the content and quality of the courses, we believe that the low attendance is caused by the fact that most students are interested in different topics.

As for the future, we plan to improve the first course by making it more comprehensible for students via including of more examples during lectures. As to the code analysis course, we definitely plan to stay up-to-date and update the content according to result of recent research in the area.

References

1. Java modeling language (JML), http://www.eecs.ucf.edu/~leavens/JML/
2. Blast project, http://mtc.epfl.ch/software-tools/blast/
3. ESC/Java2, http://kind.ucd.ie/products/opensource/ESCJava2/
4. Java PathFinder, http://javapathfinder.sourceforge.net/
5. NuSMV, http://nusmv.irst.itc.it/
6. PicoSAT, http://fmv.jku.at/picosat/
7. SatAbs tool, http://www.verify.ethz.ch/satabs/
8. Slam project, http://research.microsoft.com/en-us/projects/slam/
9. Soot framework, http://www.sable.mcgill.ca/soot/
10. Spec#, http://research.microsoft.com/en-us/projects/specsharp/
11. Spin, http://spinroot.com/spin/whatispin.html
12. Uppaal integrated environment, http://www.uppaal.com/
13. Yices, http://yices.csl.sri.com/
14. Adámek, J., Kofroň, J., Plášil, F.: NSWI101: Behavior models and verification, http://dsrg.mff.cuni.cz/teaching/nswi101/
15. Adamek, J., Plasil, F.: Component composition errors and update atomicity: static analysis: Research articles. Journal of Software Maintenance and Evolution: Research and Practice 17(5), 363–377 (2005)
16. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
17. Anand, S., Pasareanu, C.S., Visser, W.: JPF-SE: A symbolic execution extension to Java PathFinder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 134–138. Springer, Heidelberg (2007)
18. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. SIGOPS Oper. Syst. Rev. 40(4), 73–85 (2006)

19. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
20. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE 2005, pp. 82–87. ACM, New York (2005)
21. Bergstra, J., Klop, J.: Process algebra for synchronous communication. Information and Control 60(1-3), 109–137 (1984)
22. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R., Beyer, D.: The software model checker blast: Applications to software engineering. Int. J. Softw. Tools Technol. Transfer, 505–525 (2007)
23. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), pp. 29–38. IEEE Computer Society Press, Los Alamitos (2008)
24. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
25. Dijkstra, E.W.: Cooperating sequential processes. In: Programming Languages: NATO Advanced Study Institute, pp. 43–112. Academic Press, London (1968)
26. Fokkink, W.: Introduction to Process Algebra. Springer-Verlag New York, Inc., Secaucus (2000)
27. Giannakopoulou, D., Pasareanu, C.S., Cobleigh, J.M.: Assume-guarantee verification of source code with design-level assumptions. In: 26th International Conference on Software Engineering (ICSE 2004), pp. 211–220. IEEE Computer Society, Los Alamitos (2004)
28. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Synergy: a new algorithm for property checking. In: SIGSOFT 2006/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, pp. 117–127. ACM, New York (2006)
29. Henzinger, T. A., Jhala, R., Majumdar, R.: SPIN Workshop 2005 – BLAST tutorial slides, `http://www.cs.ucla.edu/~rupak/Powerpoint/BlastTutorial/SPIN2005.ppt`
30. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional, Reading (2003)
31. Kroening, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View. Springer, Heidelberg (2008)
32. Newborn, M.: Automated Theorem Proving: Theory and Practice. Springer, Heidelberg (2001)
33. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus (1999)
34. Parízek, P., Šerý, O.: NSWI132: Program analysis and code verification, `http://dsrg.mff.cuni.cz/~parizek/teaching/proganalysis/`
35. Pasareanu, C.S., Pelanek, R., Visser, W.: Predicate abstraction with under-approximation refinement. Logical Methods in Computer Science 3(1) (2007)
36. Pasareanu, C.S., Visser, W.: Verification of java programs using symbolic execution and invariant generation. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 164–181. Springer, Heidelberg (2004)

37. Ranjit, T.H., Henzinger, T.A., Jhala, R., Majumdar, R.: Lazy abstraction. In: POPL, pp. 58–70. ACM Press, New York (2002)
38. Ruys, T.C.: SPIN Workshop 2002 – SPIN beginners' tutorial, http://spinroot.com/spin/Doc/SpinTutorial.pdf
39. Ruys, T.C., Holzmann, G.J.: SPIN Workshop 2004 – advanced SPIN tutorial, http://spinroot.com/spin/Doc/Spin_tutorial_2004.pdf
40. Schwartzbach, M.: Lecture notes on static analysis, http://www.brics.dk/~mis/static.html
41. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering 10(2), 203–232 (2003)
42. Zhang, L., Malik, S.: The quest for efficient boolean satisfiability solvers. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 17–36. Springer, Heidelberg (2002)

Teaching Concurrency: Theory in Practice*

Luca Aceto[1], Anna Ingolfsdottir[1], Kim G. Larsen[2], and Jiří Srba[2]

[1] School of Computer Science, Reykjavik University, Kringlan 1, 103 Reykjavik, Iceland
[2] Department of Computer Science, Aalborg University, Selma Lagerlöfs Vej 300, 9220 Aalborg Ø, Denmark

Abstract. Teaching courses that rely on sound mathematical principles is nowadays a challenging task at many universities. On the one hand there is an increased demand for educating students in these areas, on the other hand there are more and more students being accepted with less adequate skills in mathematics. We report here on our experiences in teaching concurrency theory over the last twenty years or so to students ranging from mathsphobic bachelor students to sophisticated doctoral students. The contents of the courses, the material on which they are based and the pedagogical philosophy underlying them are described, as well as some of the lessons that we have learned over the years.

1 Introduction and Background

We report on our experiences in teaching concurrency theory, as well as principles of modelling and verification for reactive systems [18]. Some of us have been teaching such courses for about twenty years now and the underlying philosophy in our teaching has not changed much over this time span. However, the structure of the courses we shall discuss in what follows has naturally evolved over time to reflect the scientific developments in our subject matter and has been adopted in lecture series that have mostly been held at Aalborg University and Reykjavík University over the last seven to eight years. Our teaching experience and the structure, contents and pedagogical philosophy of our courses form the basis for a textbook we published in 2007 [1]. The experiences we report in this article are based on the teaching material accessible to a wide variety of students at various stages of their academic development and with differing levels of mathematical background and maturity.

1.1 Philosophy behind Our Courses

The aim of the above-mentioned semester-long courses was to introduce students at the early stage of their M.Sc. degrees, or late in their B.Sc. degree studies,

* The work of Aceto and Ingolfsdottir has been partially supported by the projects "The Equational Logic of Parallel Processes" (nr. 060013021) and "New Developments in Operational Semantics" (nr. 080039021) of the Icelandic Research Fund. Srba was partially supported by Ministry of Education of the Czech Republic, project No. MSM 0021622419.

J. Gibbons and J.N. Oliveira (Eds.): TFM 2009, LNCS 5846, pp. 158–175, 2009.
© Springer-Verlag Berlin Heidelberg 2009

in computer science to the theory of concurrency, and to its applications in the modelling and analysis of reactive systems. This is an area of formal methods that is finding increasing application outside academic circles and allows the students to appreciate how techniques and software tools based on sound theoretical principles are very useful in the design and analysis of non-trivial reactive computing systems. (As we shall discuss later in Section 2, we have also taught intensive three-week courses based on our course material as well as given lectures series intended for Ph.D. students.)

In order to carry this message across to the students in the most effective way, the courses present

- some of the prime models used in the theory of concurrency (with special emphasis on state-transition models of computation like labelled transition systems [24] and timed automata [2]),
- languages for describing actual systems and their specifications (with focus on classic algebraic process calculi like Milner's Calculus of Communicating Systems [31] and logics like modal and temporal logics [11,20,35]), and
- their embodiment in tools for the automatic verification of computing systems.

The use of the theory and the associated software tools in the modelling and analysis of computing systems is a rather important component in our courses since it gives the students hands-on experience in the application of what they have learned, and reinforces their belief that the theory they are studying is indeed useful and worth mastering. Once we have succeeded in awakening an interest in the theory of concurrency and its applications amongst our students, it will be more likely that at least some of them will decide to pursue a more in-depth study of the more advanced, and mathematically sophisticated, aspects of our field—for instance, during their M.Sc. thesis work or at a doctoral level.

It has been very satisfying for us to witness a change of attitudes in the students taking our courses over the years. Indeed, we have gone from a state in which most of the students saw very little point in taking the course on which this material is based, to one in which the relevance of the material we cover is uncontroversial to most of them! At the time when an early version of our course was elective at Aalborg University, and taken only by a few mathematically inclined individuals, one of our students remarked in his course evaluation form that 'This course ought to be mandatory for computer science students.' Now the course *is* mandatory, it is attended by all of the M.Sc. students in computer science at Aalborg University, and most of them happily play with the theory and tools we introduce in the course.

How did this change in attitude come about? And why do we believe that this is an important change? In order to answer these questions, it might be best to describe first the general area of computer science to which our courses and textbook aim at contributing. This description is also based on what we tell our students at the beginning, and during, our courses to provide them with the context within which to place the material they learn in the course and with the initial motivation to work with the theory we set about teaching them.

1.2 The Correctness Problem and Its Importance

Computer scientists build artifacts (implemented in hardware, software or, as is the case in the fast-growing area of embedded and interactive systems, using a combination of both) that are supposed to offer some well defined services to their users. Since these computing systems are deployed in very large numbers, and often control crucial, if not safety critical, industrial processes, it is vital that they correctly implement the specification of their intended behaviour. The problem of ascertaining whether a computing system does indeed offer the behaviour described by its specification is called the *correctness problem*, and is one of the most fundamental problems in computer science. The field of computer science that studies languages for the description of (models of) computer systems and their specifications, and (possibly automated) methods for establishing the correctness of systems with respect to their specifications is called *algorithmic verification*.

Despite their fundamental scientific and practical importance, however, twentieth century computer and communication technology has not paid sufficient attention to issues related to correctness and dependability of systems in its drive toward faster and cheaper products. (See the editorial [34] by David Patterson, former president of the ACM, for forceful arguments to this effect.) As a result, system crashes are commonplace, sometimes leading to very costly, when not altogether spectacular, system failures like Intel's Pentium-II bug in the floating-point division unit [36] and the crash of the Ariane-5 rocket due to a conversion of a 64-bit real number to a 16-bit integer [28].

Classic engineering disciplines have a time-honoured and effective approach to building artifacts that meet their intended specifications: before actually constructing the artifacts, engineers develop models of the design to be built and subject them to a thorough analysis. Surprisingly, such an approach has only recently been used extensively in the development of computing systems.

1.3 The Use of Tools

The courses we have given over the years stem from our deep conviction that each well educated twenty-first century computer scientist should be well versed in the technology of algorithmic, model-based verification. Indeed, as recent advances in algorithmic verification and applications of model checking [12] have shown, the tools and ideas developed within these fields can be used to analyze designs of considerable complexity that, until a few years ago, were thought to be intractable using formal analysis and modelling tools. (Companies such as AT&T, Cadence, Fujitsu, HP, IBM, Intel, Microsoft, Motorola, NEC, Siemens and Sun—to mention but a few—are using these tools increasingly on their own designs to reduce time to market and ensure product quality.)

We believe that the availability of automatic software tools for model-based analysis of systems is one of the two main factors behind the increasing interest amongst students and practitioners alike in model-based verification technology. Another is the realization that even small reactive systems—for instance, relatively short concurrent algorithms—exhibit very complex behaviours due to their

interactive nature. Unlike in the setting of sequential software, it is therefore not hard for the students to realize that systematic and formal analysis techniques *are useful*, when not altogether necessary, to obtain some level of confidence in the correctness of our designs. The tool support that is now available to explore the behaviour of models of systems expressed as collections of interacting state machines of some sort makes the theory presented in our courses appealing for many students at several levels of their studies.

It is our firmly held belief that only by teaching the theory of concurrent systems, together with its applications and associated verification tools, to our students, we shall be able to transfer the available technology to industry, and improve the reliability of embedded software and other reactive systems. We hope that our textbook and the teaching resources that accompany it, as well as the experience report provided in this article will offer a small contribution to this pedagogical endeavour.

1.4 Historical Remarks and Acknowledgments

As already stated earlier, we have used the material covered in [1] and discussed in this article in the present form for courses given at several institutions during the last seven to eight years. However, the story of its developments is much older, and goes back at least to 1986. During that year, the third author (Kim G. Larsen, then a freshly minted Ph.D. graduate from Edinburgh University) took up an academic position at Aalborg University. He immediately began designing a course on the theory of concurrency—the branch of theoretical computer science that he had worked on during his doctoral studies under the supervision of Robin Milner. His aim was to use the course, and the accompanying set of notes and slides, to attract students to his research area by conveying his enthusiasm for it, as well as his belief that the theory of concurrency is important in applications. That material and the pedagogical style he adopted have stood the 'lecture room test' well, and still form the basis for the way we organize the teaching of the part on classic reactive systems in our courses.

The development of those early courses was strongly influenced by Robin Milner's teaching and supervision that Kim G. Larsen enjoyed during his doctoral studies in Edinburgh, and would not have been possible without them. Even though the other three authors were not students of Milner's themselves, the strong intellectual influence of his work and writings on their view of concurrency theory will be evident to the readers of this book. Indeed, the 'Edinburgh concurrency theory school' features prominently in the academic genealogy of each of the authors. For example, Rocco De Nicola and Matthew Hennessy had a strong influence on the view of concurrency theory and the work of Luca Aceto and Anna Ingolfsdottir; Jiri Srba enjoyed the liberal supervision of Mogens Nielsen.

The material upon which the courses we have held at Aalborg University and elsewhere since the late 1980s were based has undergone gradual changes before reaching the present form. Over the years, the part of the course devoted

to Milner's Calculus of Communicating Systems and its underlying theory has decreased, and so has the emphasis on some topics of mostly theoretical interest. At the same time, the course material has grown to include models and specification languages for real-time systems. The courses we deliver now aim at offering a good balance between classic and real-time systems, and between the theory and its applications.

Overall, as already stated above, the students' appreciation of the theoretical material covered here has been greatly increased by the availability of software tools based on it. We thank all of the developers of the tools we use in our teaching; their work has made our subject matter come alive for our students, and has been instrumental in achieving whatever level of success we might have in our teaching.

Road map of the paper. The paper is organized as follows. We begin by describing the material we teach students in our courses as well as the types of students we have taught over the years (Section 2). We then present the organization of our courses and what role each component plays in the understanding of students (Section 3). Section 4 is devoted to the role that software tools play in our teaching. Next we introduce a possible syllabus and exam form for a one-semester course (Section 5). The proposed course and exam skeletons have served us well in our teaching over the years. Some concluding remarks are offered in Section 6.

2 What Do We Teach and to Whom?

When planning a course on a topic that is closely related to one's research interests, one is greatly tempted to cover a substantial body of fairly advanced material. Indeed, earlier editions of our courses presented rather challenging results and mathematically sophisticated techniques from books such as Milner's classic monograph [31]. Our teaching experience over the years, however, has taught us that, when presenting topics from concurrency theory to today's students at bachelor and master level, we achieve best results by following the golden rule that

> Less is more!

This is particularly true when we address bachelor students or teach intensive versions of our courses. In those situations, it becomes imperative to present a tightly knit body of theoretical material, exercises and projects that are designed in order to convey repeatedly a few main messages. Therefore, our courses do *not* aim at giving broad overviews of many of the available formalisms for describing and reasoning about reactive systems. We have instead selected a basic line of narrative that is repeated for classic reactive systems and for real-time systems. Our story line has been developed over the years with the specific aim to introduce in an accessible, yet suitably formal way, three notions that we use to describe, specify and analyze reactive systems, namely

– languages for the description of reactive systems as well as their underlying semantic models,
– behavioural relations giving the formal yardstick for arguing about correctness in the single-language approach, and
– logics for expressing properties of reactive systems following the model-checking approach to the correctness problem and their connections with behavioural relations.

2.1 Details of the Course Material

Of course, when planning a course based on the above-mentioned narrative, we are faced with a large array of possible choices of languages, models and logics. Our personal choice, which is based on our own background, personal tastes and research interests, is reflected in the topics covered in our courses and in the textbook [1].

As mentioned above, we typically divide our courses into two closely knit parts. The first part of the course deals with classic models for reactive systems, while the second presents a theory of real-time systems. In both parts we focus on the three main themes mentioned above and stress the importance of formal models of computing devices, the different approaches that one can use to specify their intended behaviour and the techniques and software tools that are available for the (automatic) verification of their correctness.

In the setting of classic reactive systems, we typically present

– Milner's Calculus of Communicating Systems (CCS) [31] and its operational semantics in terms of the model of Labelled Transition Systems (LTSs) [24],
– the crucial concept of bisimilarity [33,31] and
– Hennessy-Milner Logic (HML) [20] and its extension with recursive definitions of formulae [27].

In the second part of the course, we usually introduce a similar trinity of basic notions that allows us to describe, specify and analyze real-time systems—that is, systems whose behaviour depends crucially on timing constraints. There we present

– the formalisms of timed automata [2] and Timed CCS [45,46,47] to describe real-time systems and their semantics in terms of the model of timed labelled transition systems,
– notions of timed and untimed bisimilarity, and
– a real-time version of Hennessy-Milner Logic [26].

After having worked through the material in our courses, our students will be able to describe non-trivial reactive systems and their specifications using the aforementioned models, and verify the correctness of a model of a system with respect to given specifications either manually or by using automatic verification tools like the Edinburgh Concurrency Workbench (CWB)[1] [13], the Concurrency

[1] `http://www.dcs.ed.ac.uk/home/cwb/`

Workbench of the New Century (CWB-NC)[2] [14] and the model checker for real-time systems UPPAAL[3] [8]. These tools are integrated in the course material and are demonstrated during the lectures.

Our, somewhat ambitious, aim is to present a model of reactive systems that supports their design, specification and verification. Moreover, one of the messages that we reiterate throughout the course is that, since many real-life systems are hard to analyze manually, we should like to have computer support for our verification tasks. This means that all the models and languages that we introduce in our courses need to have a *formal* syntax and semantics.

The level of detail and formality that we adopt in teaching courses based on the above-mentioned material depends both on the level of mathematical maturity of the students and on the length of the course. However, we feel that our experience strongly indicates that the contents of our courses lends itself to presentations at widely different levels of 'mathematical depth'. The resulting types of courses are all based on the following main messages that serve as refrains in our narrative.

1. Formal models of computing systems can be developed using very expressive and flexible, but mathematically rather simple, formalisms.
2. These models are executable and can serve at the same time both as descriptions of actual implementations of systems and as their specifications. Therefore a fundamental component of their theory and practice is a notion of equivalence or approximation between objects in the model. Such a notion of equivalence or approximation may be used as a formal yardstick for establishing the correctness of systems by comparing a particular implementation of a system with its given specification.
3. Modal and temporal logics play a fundamental role in the specification of (un)desirable properties of reactive systems and are the cornerstone of the model-checking approach to the correctness problem.
4. All of the above ingredients are embodied in software tools for the automatic verification of computing systems.

2.2 Focus on Students' Understanding

In an intensive three-week course[4] taken jointly by bachelor and master students in computer science and software engineering, we cannot hope to cover much of the material presented in [1]. We therefore focus on the very basic topics that we believe the students must understand in order to use the available models, techniques and tools in a conscious way when working on the projects we set them. In such courses, when introducing, for instance, the theory of classic reactive systems, we eschew many of the mathematical details presented in the book and limit

[2] `http://www.cs.sunysb.edu/~cwb/`

[3] `http://www.uppaal.com/`

[4] For the sake of completeness, we remark here that, when we teach the course in three weeks, the students taking it are following *only* our course and are expected to devote all their time to it.

ourselves to describing the model of labelled transition systems, CCS and its operational semantics, trace equivalence, the definition of strong and weak bisimilarity and the proof technique it supports, HML and its extension with very simple recursive definitions as well as its connection with bisimilarity and with branching-time temporal logics. Knowing the syntax and basic semantics of CCS allows the students to describe their models in a way that can be used as input for tools such as the CWB and the CWB-NC. Familiarity with trace equivalence and bisimilarity gives the students enough knowledge to formulate and establish correctness requirements using equivalence checking. Moreover, the basic knowledge of HML and its recursive extension they develop is sufficient to interpret the debugging information they receive from the tools when equivalence checking tests fail and to formulate fairly sophisticated correctness requirements in logical terms.

At the other end of the spectrum are courses delivered to M.Sc. or Ph.D. students who specialize in topics related to concurrency theory. Those students can instead be taught the unified theory underlying the material presented in [1] by complementing the applications of the theory and practical assignments with a careful presentation of the main theorems and their proofs, as well as of the mathematics that underlies the algorithmics of concurrency. When teaching a course to such students, we typically cover the theory of fixed-points of endofunctions over complete lattices (unless the students are already familiar with these notions), culminating in a proof of Tarski's fixed-point theorem [42], the view of bisimilarity as a largest fixed-point and characteristic formula constructions for bisimilarity [23,39].

The teaching of the semantics of recursive extensions of HML to either kind of students is a challenging pedagogical exercise. Rather than presenting the general theory underlying the semantics of fixed-point logics in all its glory and details, we prefer to pay heed to the following advice, quoted in [25]:

> Only wimps do the general case. Real teachers tackle examples.

Following the prime role played by examples in our teaching, we essentially teach students how to compute the collection of states in a finite labelled transition system that satisfy least or largest fixed-point formulae on a variety of examples. At the same time, we make the students define recursive formulae that intuitively express some properties of interest and have them check whether the formulae are correct by performing the iterative computations to calculate their semantics over well chosen labelled transition systems. This is essentially an 'experimental' approach to teaching students how to make sense of fixed points in the definition of temporal properties of concurrent systems. In our experience, after having worked through a good number of examples, and having solved and discussed the solutions to the exercises we give them, the students are able to express basic, but important, temporal properties such as

the system can deadlock

or

it is always the case that a message that is sent will eventually be delivered.

Note that a specification of the latter property involves the use of both largest and least fixed-point formulae. We consider it a great success that students are able to define such formulae and to convince themselves that they have the expected meaning.

2.3 Explaining Formal Theories

Overall, the story line in our courses lends itself to delivery at different levels of formality, provided we stress throughout the course the connections between the mathematical theories we present and their practical applications and meaning. In our experience, if the lecturers provide sufficient context for the formal material, the students do respond by accepting the necessary mathematics within the limits of their abilities (and during the examination we take this into account as we put more focus on the application of the mathematical framework rather than on explaining the formal theories behind it). Having said so, teaching formally based techniques to present-day bachelor and master students in computer science does present challenges that any working university lecturer knows very well. The material we cover does have a strong relevance for the practice of computing, but it involves the 'M' word, viz. Mathematics. This means that many students have the preconceived idea that the material we plan to teach them is beyond their abilities. In order to make the material more palatable to students and overcome this psychological obstacle to its acceptance by our audience, we have developed a narrative that uses anthropomorphic examples and, e.g., fairly recent theoretical results on process equivalences based on games that, at least according to our own extensive but admittedly biased experience, does help the students in understanding difficult notions like bisimilarity. Our lecturing style and the structure of our courses will be described in slightly more detail in Section 3. Here we limit ourselves to mentioning that, in our experience, the game characterization of bisimilarity (see, e.g., [40,43]) helps students at all levels understand the fundamental concept of bisimilarity and its difference from the simulation preorder and the equivalence induced by the latter. All our students like to play games, be they computer games or board games, and they all seem to understand the basic rules of the bisimulation game and to play the game without too many problems. Using the game, the students can often argue convincingly when two processes are not bisimilar, also in cases when they would have trouble using the relational definition of bisimilarity. In fact, we have even had students who can use the game characterization of bisimilarity, but who do not know precisely what a relation, an equivalence relation or relation composition are. We are not sure about this lack of background says about the teaching of discrete mathematics, but it does seem to call for providing more context in those courses for the introduction of these fundamental mathematical concepts and their uses in computer science.

A similar game characterization is also used for arguing about the meaning of recursive HML formulae. This offers an alternative approach to the example-based explanation of recursive formulae and provides a different view on understanding largest and least fixed-point properties.

3 How Do We Teach Concurrency Theory?

As we mentioned in the previous section, we have used much of the material presented in the textbook [1] in several one semester courses at Aalborg University and at Reykjavík University, amongst others. These courses usually consist of about thirty hours of lectures and a similar number of hours of exercise sessions, where the students solve exercises and work on projects related to the material in the course. As we stated earlier, we strongly believe that these practical sessions play a very important role in making the students appreciate the importance of the theory they are learning, and understand it in depth. The importance that we attach to practical sessions is also reflected by the fact that we devote just as much time to them as to the lectures themselves. Indeed, we usually devote *more* time to hands-on tutorial sessions than to the lectures since two or more lecture slots are typically devoted to work on the mini-projects we set the students.

3.1 Focus on Exercises

During the exercise sessions and the mini-projects that we usually set during each installment of our courses, the students work in groups that typically consist of two or three members. The groups of students are supposed to work independently on the solutions to the exercises and to engage in peer instruction [30], at least 'in the small'. The teaching assistants and we discuss the solutions with the students, ask them further 'what if' questions that arise from their purported answers to the exercises, spur the students to question their proposed answers to the exercises and use the results of the exercise sessions to find out what topics need to be further clarified in the lecture room. We always post solutions to selected exercises after each exercise session. This allows the students to check whether the answers they proposed are in agreement with ours and whether they are convinced by the model solutions we provide. Bone fide instructors can obtain the material we typically use for the exercise sessions, as well as the model solutions we distribute, by emailing us at `rsbook@cs.aau.dk`.

As mentioned above, apart from the standard exercise sessions, students taking our courses usually work on two group projects. For each such project, the students receive six supervised work hours. The aim of the projects is to give the students experience in modelling a reasonably non-trivial scenario and in analyzing their models using one of the software tools for computer-aided verification introduced during the course.

The first project usually deals with a modelling and verification task in 'classic concurrency theory'; the students use Milner's CCS as a modelling language and Hennessy-Milner logic with recursive definitions and/or CCS itself as a specification language. They then verify their models employing either the CWB or the CWB-NC to perform equivalence checking and/or model checking as appropriate. In fact, we often ask our students to verify properties of their model using *both* equivalence checking and model checking. The rationale for this choice is that we believe that students should be familiar with both approaches to

verification since this trains them in selecting the approach that is best suited for the task at hand.

Examples of projects that we have repeatedly used over the years include modelling and analysis of

- basic communication protocols such as the classic Alternating Bit Protocol [7] or the CSMA (Carrier Sense Multi Access) Protocol,
- various mutual exclusion algorithms using CCS and Hennessy-Milner logic with recursion [44],
- the solitaire game presented in [5, Chapter 6].

A useful source for many interesting student projects is the book [15], which presents semaphore-based solutions to many concurrency problems, ranging from classic ones (like the barbershop) to more exotic problems (like the Sushi bar). Indeed, as cogently argued in [17], not only topics in classic concurrency control from courses in, say, operating systems can be fruitfully used as student projects to provide context for the material covered in concurrency-theory courses, but model checkers and other software tools developed within the concurrency-theory community can be employed to make the material typically taught in operating systems course come alive for the students.

The second project usually deals with a modelling and verification task involving real-time aspects, at least in part; the students use networks of timed automata [2] as a modelling language and the query language supported by the tool Uppaal as a specification language. They then verify their models employing Uppaal to perform model checking, synthesize schedules or strategies to win puzzles or games as appropriate. Examples of projects that we have repeatedly used over the years include modelling and analysis of

- the board game Rush Hour [48],
- the gossiping girls puzzle [22],
- real-time mutual exclusion algorithms like those presented in, e.g., [3], and
- more problems listed on the web page at `http://rsbook.cs.aau.dk/index.php/Projects`.

Examples of recent courses may be found at the URL

`http://www.cs.aau.dk/rsbook/`.

There the instructor will find suggested schedules for his/her courses, more exercises, links to other useful teaching resources available on the web, further suggestions for student projects and electronic slides that can be used for the lectures. (As an example, we usually supplement the lectures with a series of four to six 45 minute lectures on Binary Decision Diagrams [10] and their use in verification based on Henrik Reif Andersen's excellent lecture notes [4] that are freely available on the web and on Randel Bryant's survey paper [10].)

3.2 The Textbook We Use

Our pedagogical style in teaching concurrency theory can be gleaned by looking at our own recent textbook on the subject [1]. This book is by no means the

first one devoted to aspects of the theory of reactive systems. Some of the books that have been published in this area over the last twenty years or so are the references [6,16,19,21,29,31,37,38,41] to mention but a few. However, unlike all the aforementioned books but [16,29,38], ours was explicitly written to serve as a *textbook*, and offers a distinctive pedagogical approach to its subject matter that derives from our extensive use of the material presented there in book form in the classroom. In writing that textbook we have striven to transfer on paper the spirit of the lectures on which that text is based. Our readers will find that the style in which that book is written is often colloquial, and attempts to mimic the Socratic dialogue with which we try to entice our student audience to take active part in the lectures and associated exercise sessions. Explanations of the material presented in the textbook are interspersed with questions to our readers and exercises that invite the readers to check straight away whether they understand the material as it is being presented. This is precisely how we present the material in the classroom both during the lectures and the tutorial sessions. We engage the students in continuous intellectual table tennis so that they are enticed to work through the course material as it unfolds during the course sessions, in some cases feeling that they are 'discovering things themselves' as the course progresses.

As we mentioned earlier, we have developed a collection of anthropomorphic examples that, we believe, make the introduction of key notions come alive for many of our students. By way of example, we mention here that we introduce the whole syntax of Milner's CCS by telling the story of a computer scientist who wants to maximize her chances of obtaining tenure at a research university by gaining exclusive access to a coffee machine, which she needs to continue producing publications after having published her first one straight out of her thesis.

As another example of this approach, we describe the iterative algorithm for computing the set of processes satisfying largest fixed-point formulae in Hennessy-Milner logic with recursion by drawing a parallel with the workings of a court of law. Each process satisfies the formula (or 'is innocent') unless we can find a reason why it should not (that is, 'unless it is proven to be guilty'). Apart from leading to memorable scientific theatre, we think that this analogy helps students appreciate and remember the main ideas behind the iterative algorithms better. Dually, when introducing the iterative algorithm for computing the set of processes satisfying least fixed-point formulae in Hennessy-Milner logic with recursion, we say that the key intuition behind the algorithms is that no process is 'good' (satisfies the formula) unless it is proven to be so.

These are simple, but we believe telling, examples of the efficiency of story telling in the teaching of computer science and mathematics, as put forward by Papadimitriou in [32]. The readers of our book [1] and of the further material available from the book's web site will find other examples of the use of this pedagogical approach in our teaching and educational writings.

4 The Role of Software Tools in Our Teaching

We strongly recommend that the teaching of concurrency theory be accompanied by the use of software tools for verification and validation. In our courses, we usually employ the Edinburgh Concurrency Workbench [13] and/or the Concurrency Workbench of the New Century [14] for the part of the course devoted to classic reactive systems and, not surprisingly, Uppaal [8] for the lectures on real-time systems. All these tools are freely available, and their use makes the theoretical material covered during the lectures come alive for the students. Using the tools, the students will be able to analyze systems of considerable complexity, and we suggest that courses based upon our book and the teaching philosophy described in this article be accompanied by two practical projects involving the use of these, or similar, tools for verification and validation.

We moreover recommend that the aforementioned tools be introduced as early as possible during the courses, preferably already at the moment when the students hear for the first time about the language CCS, and that the use of tools be integrated into the exercise sessions. For example, the students might be asked about the existence of a particular transition between two CCS expressions and then proceed with verifying their answers by using a tool. This will build their confidence in their understanding of the theory as well as motivate them to learn more about the principles behind these tools. On the other hand, it is important that the students are faced also with exercises that are solved without the tool support in order to avoid them getting the impression that tools can substitute for theory.

In several of our recent courses, we have also used blogs as a way to entice the students to put their thoughts about the course material and their solutions to the exercises in writing. The rationale for this choice is that we feel that many of our students underestimate the importance of writing down their thoughts clearly and concisely, and of explaining them to others in writing. The use of blogs allows students to comment on each other's posts, thoughts and solutions. Moreover, in so doing, they learn how to exercise restraint in their criticisms and how to address their peers in a proper scientific debate. In order to encourage students to make use of course blogs, we have sometimes reserved a tiny part of the final mark of the course, say 5%, for activity on the course blog. Overall, students have made good use of the course blogs whenever we have asked them to do so. We think that what the students learn by using this medium justifies our decision to award a tiny part of the final grade for the course based on blog-based activities and assignments. Despite the lack of conclusive data, we believe that the use of a blog or of similar software is beneficial in university level courses.

Finally, let us remark that we encourage the students taking our courses to experiment with 'The Bisimulation-Game Game'[5]. This tool, which has been developed by Martin Mosegaard and Claus Brabrand, allows our students to play the bisimulation game on their laptops and to experiment with the behaviour of

[5] `http://www.brics.dk/bisim/`

processes written in Milner's CCS using the included graphical CCS visualizer and simulator. It would be interesting to obtain hard data measuring whether the use of such a graphical tool increases the students' understanding of the bisimulation game. We leave this topic for future investigations.

5 A Possible Syllabus and Exam Form

In this section we shall present a possible syllabus of the course which integrates two mini-projects. The assumption is that the course is given in 15 lectures, each lecture consisting of two 45 minute blocks.

- **Lecture 1: Labelled Transition Systems.** Introduction to the course; reactive systems and their modelling via labelled transition systems; CCS informally.
- **Lecture 2: CCS.** Formal definition of CCS syntax; SOS rules; number of examples; value passing CCS.
- **Lecture 3: Strong Bisimilarity.** Trace equivalence; motivation for behavioral equivalences; definition of strong bisimilarity; game characterization; examples and further properties of bisimilarity.
- **Lecture 4: Weak Bisimilarity.** Internal action τ; definition of weak bisimilarity; game characterization; properties of weak bisimilarity; a small example of a communication protocol modelled in CCS and verified in CWB.
- **Lecture 5: Hennessy-Milner Logic.** Motivation; syntax and semantics of HML; examples in CWB; correspondence between strong bisimilarity and HML on image-finite transition systems.
- **Lecture 6: Tarski's Fixed Point Theorem.** Motivation via showing the need for introducing temporal properties into HML; complete lattices; Tarski's fixed point theorem and its proof; computing fixed points on finite lattices.
- **Lecture 7: Hennessy-Milner Logic with Recursion.** Bisimulation as a fixed-point; one recursively defined variable in HML; game characterization; several recursively defined variables in HML.
- **Lecture 8: First Mini-Project.** Modelling and verification of Alternating Bit Protocol in CWB.
- **Lecture 9: Timed CCS.** Timed labelled transition systems; syntax and semantics of timed CCS; introduction to timed automata.
- **Lecture 10: Timed Automata.** Timed automata formally; networks of timed automata; timed and untimed bisimilarity; region construction.
- **Lecture 11: Timed Automata in UPPAAL.** UPPAAL essentials; practical examples; algorithms behind UPPAAL; zones.
- **Lecture 12: Second Mini-Project.** Modelling and verification of Rush Hour puzzle.
- **Lecture 13: Binary Decision Diagrams.** Boolean expressions; normal forms; Shannon's expansion law; ordered and reduction binary decision diagrams; canonicity lemma; algorithms for manipulating binary decision diagrams.

- **Lecture 14: Applications of Binary Decision Diagrams.** Constraint solving; Boolean encoding of transition systems; bisimulation model checking; tool IBEN.
- **Lecture 15: Round-Up of the Course.** Overview of key concepts covered during the course; exam information.

We recommend that the lectures be accompanied by two hours of exercises, with a possible placement before the lectures so that exercises related to Lecture 1 are solved right before Lecture 2 and so on. We find the students' active involvement in the exercise sessions crucial for the success of the course. To further motivate the students we use the technique of *constructive alignment* [9] so that the course objectives, the information communicated with the students during the lectures and exercises, as well as the exam content and form are aligned. This practically means that in each lecture we explicitly identify two or three main points that are essential for the understanding of the particular course topic, we exercise those points during the exercise sessions and then examine them at the exam. In order to be more explicit about the essential elements of the course, we mark in each exercise session one or two problems with a star to indicate their importance. During the exam (which is typically oral in our case, but can be easily adapted to a written one) the students pick up one of the star exercises (or rather an instance of the corresponding type of the exercise, unique to each student) and are then given 20 minutes to find a solution, while at the same time another student is being examined. At the start of the oral exam we ask the respective student to first present the solution to the chosen exercise and we proceed with the standard examination (presentation of one randomly selected topic) only if the exercise was answered at a satisfactory level.

After introducing the alignment technique described above we can report on a remarkable increase in students' involvement in the exercise sessions as well as in their understanding of the most essential notions covered during the course.

6 Concluding Remarks

In this article, we have reported on our experiences in teaching concurrency theory over the last twenty years or so to a wide variety of students, ranging from mathsphobic bachelor students to sophisticated doctoral students. We have described the contents of the courses, the material on which they are based and the pedagogical philosophy underlying them, as well as some of the lessons that we have learned over the years. Our main message is that concurrency theory *can* be taught with a, perhaps surprisingly, high degree of success to many different types of students provided that courses present a closely knit body of theoretical material, exercise and practical sessions, coupled with the introduction of software tools that the students use in modelling and verification projects. Our teaching experience over the years forms the basis for our textbook [1] and for the material that is available from its associated web site.

Of course, it is not up to us to determine how successful our pedagogical approach to teaching concurrency theory to many different types of students

is. We can say, however, that several of the students taking our courses, who will not specialize within formal methods, appreciate the course as they often have the opportunity of applying the methods and tools we introduce in their projects on the practical construction of various kinds of distributed systems—for instance, in order to better understand a particular communication network or embedded controller. We believe that the ubiquitous nature of distributed and embedded systems, together with the fact that most students consider the crucial importance of their proper functioning uncontroversial, has been a crucial factor in the change of attitudes in the students taking our courses over the years mentioned in Section 1.1. Another factor contributing to an increase in our students' interest in the material we teach them in our courses is their realization that it helps them understand better the material that is taught in follow-up courses, such us those in concurrent programming and distributed algorithms.

Also, several neighbouring departments—in particular, the Control Theory Department and the research group on hardware-software co-design at Aalborg University—have adopted the model-based approach and tool support advocated by our course, both in research and in teaching at several levels (including doctoral education).

Finally, let us mention that, to the best of our knowledge, our textbook and the teaching approach we have described in this paper have so far been adopted in at least 15 courses at universities in several European countries and in Israel. (See `http://rsbook.cs.aau.dk/index.php/Lectures`.)

Overall, the positive feedback that we have received from the colleagues of ours who have used our material and pedagogical approach in their teaching and from the students following our courses gives us some hope that we may be offering a small contribution to making students appreciate the beauty and usefulness of concurrency theory and of algorithmic, model-based verification.

References

1. Aceto, L., Ingólfsdóttir, A., Larsen, K.G., Srba, J.: Reactive Systems: Modelling, Specification and Verification. Cambridge University Press, Cambridge (2007)
2. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Comput. Sci. 126(2), 183–235 (1994); Fundamental Study
3. Alur, R., Taubenfeld, G.: Fast timing-based algorithms. Distributed Computing 10(1), 1–10 (1996)
4. Andersen, H.R.: An introduction to binary decision diagrams (1998), Version of October 1997 with minor revisions April 1998, p. 36, `http://www.itu.dk/people/hra/notes-index.html`
5. Arnold, A., Bégay, D., Crubillé, P.: Construction and Analysis of Transition Systems Using MEC. AMAST Series in Computing, vol. 3. World Scientific, Singapore (1994)
6. Baeten, J.C., Weijland, P.: Process Algebra. Cambridge Tracts in Theoretical Computer Science, vol. 18. Cambridge University Press, Cambridge (1990)
7. Bartlett, K., Scantlebury, R., Wilkinson, P.: A note on reliable full–duplex transmission over half–duplex links. Commun. ACM 12, 260–261 (1969)

8. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
9. Biggs, J.: Teaching for quality learning at University. Open University Press, Stony Stratford (1999)
10. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. 24(3), 293–318 (1992)
11. Clarke, E., Emerson, E.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)
12. Clarke, E., Gruemberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
13. Cleaveland, R., Parrow, J., Steffen, B.: The concurrency workbench: A semantics-based tool for the verification of concurrent systems. ACM Trans. Prog. Lang. Syst. 15(1), 36–72 (1993)
14. Cleaveland, R., Sims, S.: The NCSU Concurrency Workbench. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 394–397. Springer, Heidelberg (1996)
15. Downey, A.B.: The Little Book of Semaphores, 2nd edn. Green Tea Press (2008), http://www.greenteapress.com/semaphores/
16. Fokkink, W.: Introduction to Process Algebra. Texts in Theoretical Computer Science. An EATCS Series. Springer, Berlin (2000)
17. Hamberg, R., Vaandrager, F.: Using model checkers in an introductory course on operating systems. Operating Systems Review 42(6), 101–111 (2008)
18. Harel, D., Pnueli, A.: On the development of reactive systems. In: Logics and models of concurrent systems (La Colle-sur-Loup, 1984). NATO Adv. Sci. Inst. Ser. F Comput. Systems Sci., vol. 13, pp. 477–498. Springer, Berlin (1985)
19. Hennessy, M.: Algebraic Theory of Processes. MIT Press, Cambridge (1988)
20. Hennessy, M., Milner, R.: Algebraic laws for nondeterminism and concurrency. J. ACM 32(1), 137–161 (1985)
21. Hoare, C.: Communicating Sequential Processes. Prentice-Hall International, Englewood Cliffs (1985)
22. Hurkens, C.: Spreading gossip efficiently. Nieuw Archief voor Wiskunde 5/1(2), 208–210 (2000)
23. Ingolfsdottir, A., Godskesen, J.C., Zeeberg, M.: Fra Hennessy-Milner logik til CCS-processer. Master's thesis, Department of Computer Science, Aalborg University (1987) (in Danish)
24. Keller, R.: Formal verification of parallel programs. Commun. ACM 19(7), 371–384 (1976)
25. Krantz, S.G.: How to Teach Mathematics (a personal pespective). American Mathematical Society, Providence (1993)
26. Laroussinie, F., Larsen, K.G., Weise, C.: From timed automata to logic - and back. In: Hájek, P., Wiedermann, J. (eds.) MFCS 1995. LNCS, vol. 969, pp. 529–539. Springer, Heidelberg (1995)
27. Larsen, K.G.: Proof systems for satisfiability in Hennessy–Milner logic with recursion. Theoretical Comput. Sci. 72(2–3), 265–288 (1990)
28. Lions, J.L.: ARIANE 5 flight 501 failure: Report by the inquiry board (July 1996), http://www.cs.aau.dk/~luca/SV/ariane.pdf
29. Magee, J., Kramer, J.: Concurrency: State Models and Java Programs. John Wiley, Chichester (1999)

30. Mazur, E.: Peer Instruction: A User's Manual. Series in Educational Innovation. Prentice-Hall International, Upper Saddle River (1997)
31. Milner, R.: Communication and Concurrency. Prentice-Hall International, Englewood Cliffs (1989)
32. Papadimitriou, C.H.: Mythematics: storytelling in the teaching of computer science and mathematics. In: Dagdilelis, V., Satratzemi, M., Finkel, D., Boyle, R.D., Evangelidis, G. (eds.) Proceedings of the 8th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2003, Thessaloniki, Greece, June 30–July 2, p. 1. ACM, New York (2003)
33. Park, D.: Concurrency and automata on infinite sequences. In: Deussen, P. (ed.) GI-TCS 1981. LNCS, vol. 104, pp. 167–183. Springer, Heidelberg (1981)
34. Patterson, D.A.: 20^{th} century vs. 21^{st} century C&C: The SPUR manifesto. Commun. ACM 48(3), 15–16 (2005)
35. Pnueli, A.: The temporal logic of programs. In: Proceedings 18^{th} Annual Symposium on Foundations of Computer Science, pp. 46–57. IEEE, Los Alamitos (1977)
36. Pratt, V.R.: Anatomy of the Pentium bug. In: Mosses, P.D., Schwartzbach, M.I., Nielsen, M. (eds.) CAAP 1995, FASE 1995, and TAPSOFT 1995. LNCS, vol. 915, pp. 97–107. Springer, Heidelberg (1995)
37. Roscoe, B.: The Theory and Practice of Concurrency. Prentice-Hall International, Englewood Cliffs (1999)
38. Schneider, S.: Concurrent and Real-time Systems: The CSP Approach. John Wiley, Chichester (1999)
39. Steffen, B., Ingolfsdottir, A.: Characteristic formulae for processes with divergence. Information and Computation 110(1), 149–163 (1994)
40. Stirling, C.: Local model checking games. In: Lee, I., Smolka, S.A. (eds.) CONCUR 1995. LNCS, vol. 962, pp. 1–11. Springer, Heidelberg (1995)
41. Stirling, C.: Modal and Temporal Properties of Processes. Springer, Heidelberg (2001)
42. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics 5, 285–309 (1955)
43. Thomas, W.: On the Ehrenfeucht-Fraïssé game in theoretical computer science (extended abstract). In: Gaudel, M.-C., Jouannaud, J.-P. (eds.) CAAP 1993, FASE 1993, and TAPSOFT 1993. LNCS, vol. 668, pp. 559–568. Springer, Heidelberg (1993)
44. Walker, D.: Automated analysis of mutual exclusion algorithms using CCS. Journal of Formal Aspects of Computing Science 1, 273–292 (1989)
45. Yi, W.: Real-time behaviour of asynchronous agents. In: Baeten, J.C.M., Klop, J.W. (eds.) CONCUR 1990. LNCS, vol. 458, pp. 502–520. Springer, Heidelberg (1990)
46. Yi, W.: A Calculus of Real Time Systems. PhD thesis, Chalmers University of Technology, Göteborg, Sweden (1991)
47. Yi, W.: CCS + time = an interleaving model for real time systems. In: Leach Albert, J., Monien, B., Rodríguez-Artalejo, M. (eds.) ICALP 1991. LNCS, vol. 510, pp. 217–228. Springer, Heidelberg (1991)
48. Yoshigahara, N.: Rush Hour, Traffic Jam Puzzle, `http://www.puzzles.com/products/rushhour.htm` by Puzzles.com (accessed on July 10, 2009)

Author Index