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/`

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

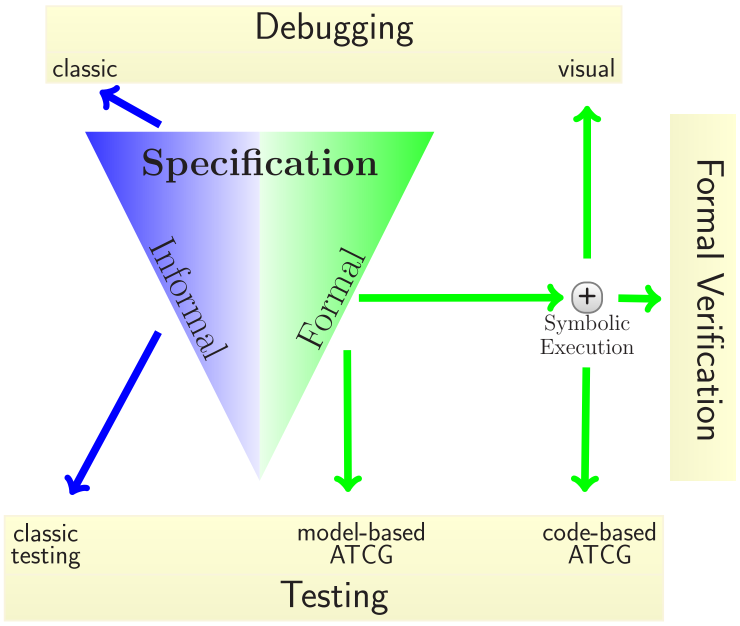


Fig. 1. Dependencies among TDV topics

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).

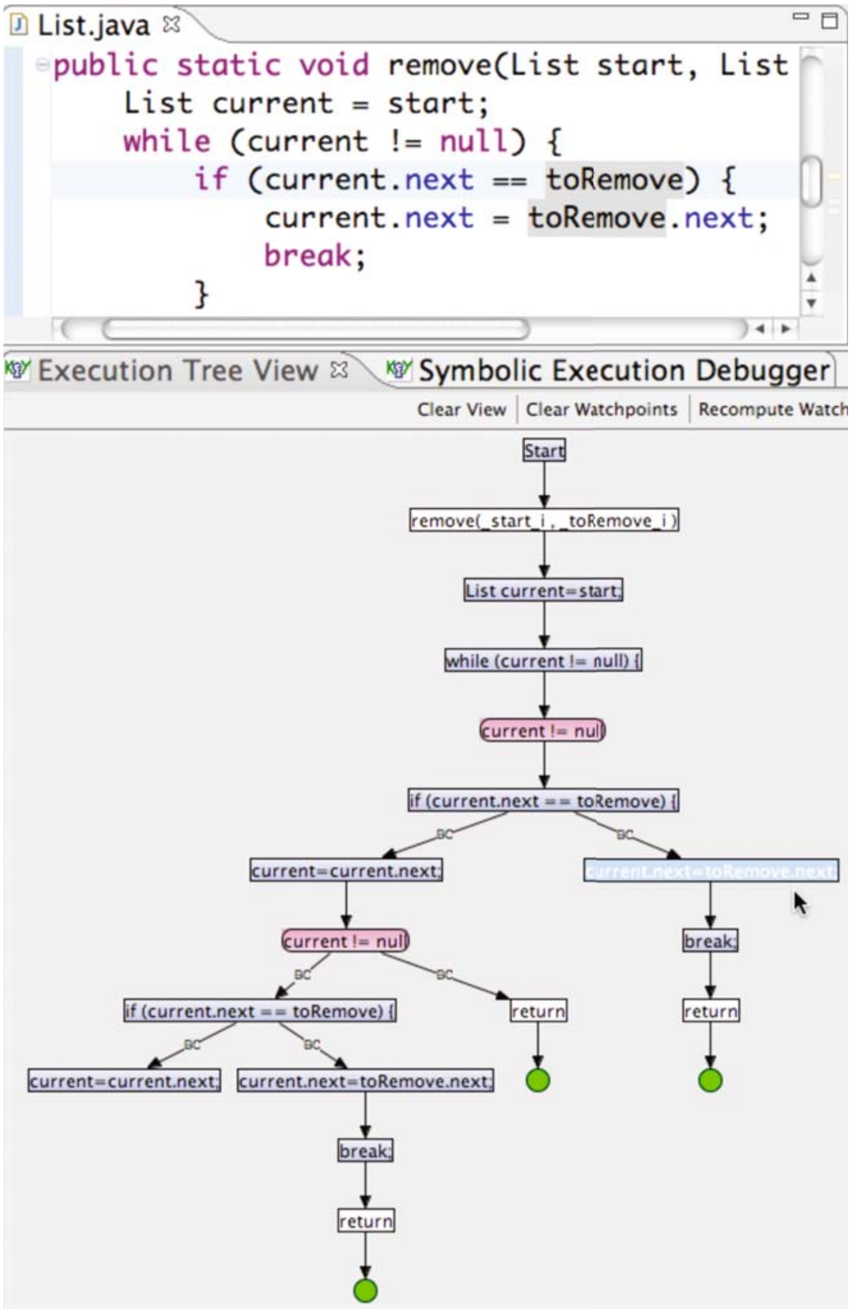
In contrast to a standard debugger, the visual debugger is based on symbolic execution. As described in Section 4.2 the idea of symbolic execution is to run a program not on concrete, but on symbolic input values. The set of the initial states to be considered can be restricted by adding constraints 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.

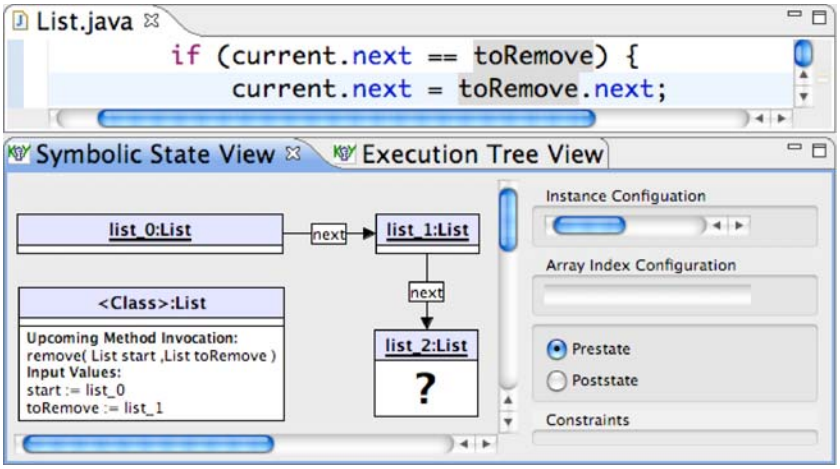
Besides the execution tree view (see Figure 2), the visual debugger provides also a visualisation of the symbolic state for any execution tree node. The symbolic state is visualised as a symbolic UML object diagram (see Figure 3). The user can browse through all possible configurations of the symbolic 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.



```
public static void remove(List start, List
    List current = start;
    while (current != null) {
        if (current.next == toRemove) {
            current.next = toRemove.next;
            break;
        }
```

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.

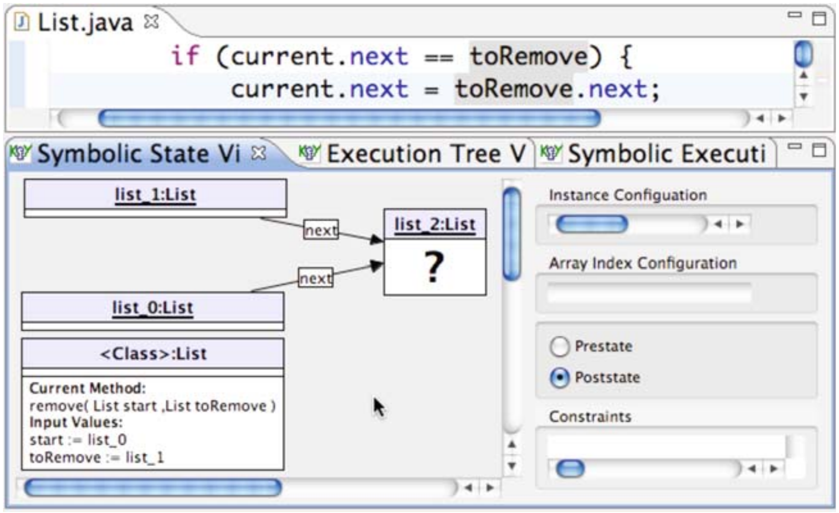


```
if (current.next == toRemove) {
    current.next = toRemove.next;
```

Upcoming Method Invocation:
remove(List start ,List toRemove)
Input Values:
start := list_0
toRemove := list_1

[✓ Prestate] Poststate

(a) Symbolic state before removal of list element `list_1`



```
if (current.next == toRemove) {
    current.next = toRemove.next;
```

Current Method:
remove(List start ,List toRemove)
Input Values:
start := list_0
toRemove := list_1

Prestate [✓ Poststate]

(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)