A comparison of tools for teaching formal software verification

Ingo Feinerer, Gernot Salzer

▶ To cite this version:

Ingo Feinerer, Gernot Salzer. A comparison of tools for teaching formal software verification. Formal Aspects of Computing, Springer Verlag, 2008, 21 (3), pp.293-301. 10.1007/s00165-008-0084-5 . hal-00477907

HAL Id: hal-00477907
https://hal.archives-ouvertes.fr/hal-00477907

Submitted on 30 Apr 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

DOI 10.1007/s00165-008-0084-5
BCS © 2008
Formal Aspects of Computing (2009) 21: 293–301

Formal Aspects of Computing

A comparison of tools for teaching formal software verification

Ingo Feinerer and Gernot Salzer

Institut für Computersprachen, Technische Universität Wien, Favoritenstr. 9/E185, 1040 Vienna, Austria.
E-mail: salzer@logic.at

Abstract. We compare four tools regarding their suitability for teaching formal software verification, namely the Frege Program Prover, the KEY system, Perfect Developer, and the Prototype Verification System (PVS). We evaluate them on a suite of small programs, which are typical of courses dealing with Hoare-style verification, weakest preconditions, or dynamic logic. Finally we report our experiences with using Perfect Developer in class.

Keywords: Formal software verification; Frege Program Prover; KEY system; Perfect developer; Prototype verification system

1. Introduction

Courses on the formal verification of software are part of many curricula worldwide, at least as elective subjects. Typically they cover topics like the specification of program properties using formal logic, the systematic development of programs guided by loop invariants, and formal semantics based on the Hoare calculus or weakest preconditions. Assignments are usually done on paper. Contrary to hardware verification, where fully automated model checkers are regularly used by industry, there are hardly any automated tools supporting formal software verification due to its inherent complexity.

The course *Formal Verification of Software* at our university was no exception; the students had to verify numerous one-loop-programs by hand, guided by the books by Dijkstra [Dij76] and Gries [Gri87]. The slow but steady progress in the automation of program verification led us in 2004 to evaluate several tools in order to select one for use in the course [Fei05]. Our aim was twofold. On the one hand an automated verification system lets the students concentrate on formal specification, with immediate feedback if the program does not match the specification. On the other hand we think that automated program verification has become mature enough to spread the word in industry, via the students.

This paper is not intended as a comprehensive overview of tools for formal verification, nor does it offer a general methodology for evaluating such tools. Instead, we present four candidate systems selected with respect to the goals of our particular course (Sect. 3), and compare them on a small set of examples typical of the assignments there (Sect. 4). Section 5 summarises the results and argues why we decided in favour of Perfect Developer. Finally, Sect. 6 reports our experiences with using it in class. To put the tool evaluation into context, we start with a section describing the situation of formal methods at our university.

Correspondence and offprint requests to: G. Salzer, E-mail: salzer@logic.at

Table 1. Comparison of old and new CS curricula

	Duration	Credits	Math and statistics	Theory and logic
Diploma study (up to 2001)	5 years	324 Ects	34.5 Ects (10.6%)	19.5 Ects (6.0%)
Bac.+Mast. (2001 onwards)	3+2 years	180 + 120 Ects	21.0 Ects (7.0%)	12.0 Ects (4.0%)

Ects ECTS credit, where ECTS means European Credit Transfer System; 1 year corresponds to 60 Ects, 1 Ects equals 25 working hours in Austria.

2. Educational background

Curricula of studies in computer science and software engineering currently undergo (or recently underwent) more or less dramatic changes at many universities. Among the reasons are: making the studies more attractive to counter the decrease in enrolment [DM05, Pat05] or to address the underrepresentation of women and minorities [SL03]; reflecting the growth of the field by replacing old-fashioned topics by new ones; or adapting the studies to new regulations like the Bologna declaration of the EU. No matter what the reasons are, the changes usually lead to a reduction of mathematical and formal training; e.g., traditional topics like formal language and recursion theory, formal specification and verification, or the concept of mathematical proof are dropped or moved to elective courses.

As an example consider the situation at the *Technische Universität Wien* (TUW, Vienna University of Technology). Its computer science department is the biggest in Austria: About 120 full, associate, and assistant professors (5 % of TUW's teaching personnel) teach approximately 6,000 students (30 % of TUW's students). Until 2001, there were only two monolithic studies: *Informatik* (informatics) taking 5 years and *Wirtschaftsinformatik* (business informatics) taking 4.5 years. In 2001, triggered by the Bologna declaration of the European Union, the two studies were replaced by six bachelor[1] and nine master[2] degrees taking three and two years, respectively.

The change of the curricula had a significant impact on the kind and amount of formal methods taught. First, specialisation and diversification now start as early as the second year; the required room was partially gained by reducing mathematics, theory and logic by one third (Table 1). Second, traditional theory and logic was compressed into a single course on bachelor level; a second course on master level now focuses on applied logics following by and large the book *Logic in Computer Science* by Huth and Ryan [HR03]. Apart from these compulsory courses there are only a few elective ones dealing with formal methods, most notably *Computer Aided Verification* emphasising model checking [CGP00], and *Formal Verification of Software* concentrating on Hoare calculus and dynamic logic; each has 6.0 Ects and consists of a lecture and a lab. In summary, the change led to a modernisation concerning theory and logic, but at the same time reduced mathematics to a minimum, leaving hardly any room to train the ability to understand and construct rigorous proofs.

Consequently, our master course on software verification had to cope with a new situation. First, students no longer have a uniform background in programming, theory, and mathematics, since they are admitted to master studies on the basis of many different bachelor studies of computer science. Moreover, students increasingly come from abroad funded by exchange programs, doing the bachelor in one place and the master in another. Second, the focus of the subjects taught in bachelor studies has shifted. Nowadays students are trained in object-oriented modeling, networked computing, component-oriented software construction, and techniques specific to application areas, whereas the design and implementation of algorithms and data structures based on (semi-)formal principles, abstract specifications and formal reasoning has become less important.

The aim of our course is to give students of varied mathematical and computing backgrounds a common grounding in verification concepts and practice by introducing formal semantics of typical programming constructs using the Hoare calculus and weakest preconditions, by showing how to specify the behaviour of programs and derive correct programs from formal specifications, and by training the ability to argue rigorously about program correctness. Books like the one by Gries [Gri87] are old regarding the time scale of computer science, but in our opinion are still of high didactic value. Based on this and similar books, our course illustrates the concepts and techniques on small examples involving mainly integers, arrays of integers, and simple loops.

[1] Business Informatics, Computer Engineering, Data Engineering and Statistics, Media Informatics, Medical Informatics, Software and Information Engineering.

[2] Business Engineering and Informatics (*Wirtschaftsingenieurwesen Informatik*), Business Informatics, Computer Engineering, Computational Intelligence, Information and Knowledge Management, Media Informatics, Medical Informatics, Software Engineering and Internet Computing, Visual Computing.

3. Tools for formal program verification

Tools for formal program verification roughly fall into three categories: interactive or semi-automatic proof construction environments like ISABELLE [Isa] or the Prototype Verification System [PVS, ORS92], tools for model checking (using abstract interpretation and similar techniques) like the Symbolic Model Verifier [SMV], and systems based on the Hoare calculus or related approaches (weakest precondition, dynamic logic) like the Frege Program Prover [FPP, Win97], the KEY system [KEY, ABB+05, BHS07], or Perfect Developer [PD, Cro03]. We selected the latter three systems for evaluation, since the theory part of the course concentrates on structured program development using invariants, pre- and postconditions. Moreover, we included PVS to see whether systems for verifying algorithms (instead of programs) could be an alternative for teaching formal methods. We give short descriptions of these four tools.

Frege program prover The Frege Program Prover (FPP) is an experimental system developed by a group at *Friedrich-Schiller-Universität Jena* (Germany) for proving the correctness of simple annotated programs. FPP is implemented as a web application: the annotated program is pasted into a web form, the verification is done at the home site of FPP, and the result is again presented as a web page.

The program syntax is a subset of Ada and offers assignments, compound statements, if- and case-conditionals, and for- and while-loops. The only supported data types are Boolean and integer. Annotations can be used to specify pre- and postconditions, loop invariants, and termination functions. The system computes weakest preconditions or proves the correctness of Hoare triples. Verification conditions are discharged using an extended version of Analytica [CZ92], an automated theorem prover for theorems in elementary analysis. Analytica is written in the script language of Mathematica [Mat].

The key project The KeY project is developed jointly by groups at *Universität Karlsruhe* (Germany), *Universität Koblenz-Landau* (Germany), and *Chalmers University of Technology* (Sweden). It is distributed under the GNU General Public License; additionally one of the supported commercial CASE tools and the JAVA runtime environment is required.

KEY aims at integrating formal specification and verification of software into the software development process. This is achieved by coupling the KEY system tightly with a CASE tool like Borland's *Together Control Center*, by supporting JAVACARD—a single-threaded subset of JAVA intended for programming mobile and embedded devices like smart cards—as programming language, and by accepting the Unified Modeling Language (UML) [Obj05], the Object Constraint Language (OCL) [Obj06], and since recently also the Java Modeling Language (JML) as specification language.

KEY is able to generate OCL constraints automatically by instantiating design patterns. Additionally the user may specify any valid OCL statement, may annotate the program with JML clauses, or may use the logic internally used by KEY, dynamic logic for JAVACARD. KEY then checks the specification for consistency. If the user provides a refinement of the specification in the form of a program written in JAVACARD, KEY generates proof obligations in first-order dynamic logic and tries to discharge them automatically; alternatively, proofs can be constructed interactively by specifying explicitly the rules to be applied.

Perfect developer Perfect Developer (PD) is a commercial tool for the development of safety-critical software developed by the British company *Escher Technologies Limited*. Binaries for Windows and Linux are available free of charge to universities and schools.

PD consists of a separate object-oriented programming language called Perfect, of an automated theorem prover, and of a code generator translating programs from Perfect to JAVA or C++. A rich collection of built-in data types, classes, functions, and theories allows the user to write concise specifications on a fairly abstract level. The advantage of using Perfect, a newly designed language, instead of an existing one is that the specification language, the programming language, and the prover match each other concerning expressiveness.

PD comes with a small project manager that helps to circle between editing source files (specifications or programs) with one of several supported language-sensitive editors (Crimson, Text Pad, Multi-Edit, XEmacs, Kate, Vim), verifying specifications and refinements, generating JAVA or C++ code, and compiling this code. UML class diagrams can be imported to create the skeleton of classes automatically. As feedback to the user, the verification phase provides proofs in natural deduction style and a description of failed proof attempts, in HTML- or TEX format.

Table 2. Problems for evaluating the systems

Problem	Description
Conditional	Check pre- and postconditions for an if-statement
Cubic sum	Given a positive integer n, compute $\sum_{i=1}^{n} i^3$
Multiplication	Given two positive integers m and n, compute the product $m \cdot n$
Division	Given two positive integers m and n, compute the quotient and remainder for the division $\frac{m}{n}$
Factorial	Given a positive integer n, compute the factorial $\prod_{i=1}^{n} i$
Prime	Given a nonnegative integer n, determine whether n is prime
Index of maximum	Given a list of integers, determine the index of a maximal element
Card. of intersection	Given two sorted lists of integers, compute the cardinality of their intersection
Count inversions	Given a list, A, of n integers, count the pairs (i, j) of indices satisfying $i < j$ and $A[i] > A[j]$
Quicksort	Given a sequence of integers, perform Hoare's Quicksort

Prototype verification system The Prototype Verification System (PVS) was developed at the *Stanford Research Institute* (SRI). It is an open source Lisp program under the GNU General Public License with pre-built binaries for Linux, SunOS and MacOS, and uses the Emacs editor as user interface.

PVS is a specification language integrated with support tools and a semi-automatic theorem prover. It is used for academic purposes to test new methods, but has also been successfully applied in safety-critical projects, e.g., at NASA. In contrast to the systems presented above it does not generate verified program code, but proves properties of algorithms. The specification language is strongly typed and supports recursive definitions as well as parametrised theories that encapsulate abstract data types and their verified properties.

4. Evaluation

We evaluated the tools with respect to the following criteria.

1. *Expressiveness* The specification languages of the tools have to be sufficiently expressive to support the examples covered in the course. As benchmark we used the problems listed in Table 2, which are representative of those in the course (Tables 5, 6), but which in general do not demonstrate the full power of the systems.
2. *Ease of use* Bearing in mind that the main objective of the course is to teach general principles of program verification, the students should not be forced to spend too much time on learning a new specification/programming language and on understanding its idiosyncrasies. Moreover, once a student has obtained a correct program by systematically refining the specification (as advocated in the course), the system should be able to discharge the proof obligations as automatically as possible.
3. *Feedback on errors* Error messages and reports on proof failures should provide feedback that can be readily understood without deep knowledge of the tool's specific underpinning formalism, in order to keep the focus on the verification problems itself. Note that specifications, like programs, are hardly ever correct and complete on the first try, so support in locating conflicts between specification and refinement is essential.
4. *Adequate documentation* There should be sufficient documentation comprehensible to students at masters level. The better the documentation the less help the teaching staff has to provide on basic language issues.
5. *Ease of installation* The system should be easy to install on any of the platforms students typically use, i.e., on Windows, Linux, and MacOS. It should not be necessary to install further software components from other sources.

In the remaining section we describe our experiences with each of the four systems. Table 3 summarises their evaluation on our set of benchmark problems. To convey an idea what the input to the systems looks like, Table 4 shows implementations of the factorial function for all four systems.

Frege program prover The evaluated version dates from May 22, 2001. It seems that there was a minor update in September 2006, which does not affect the result of the evaluation.

The first thing one notes about FPP is its simplicity. A few web pages tell everything one needs to know, there is no installation required since it is implemented as a web service hosted in Jena, and the functionality covers exactly the topics discussed in introductory books on formal software verification: given a program and a postcondition, it computes the weakest precondition; given additionally a precondition, the system checks whether the program is correct with respect to its specification. Proofs are fully automatic, no human intervention is needed.

Table 3. Summary of evaluation on benchmark problems

Problem	FPP	KEY	PD	PVS
Conditional	✓	✓	✓	.
Cubic sum	✓	✓	✓	✓
Multiplication	✓	✓	✓	✓
Division	✓	✓	✓	.
Factorial	✓	?	✓	✓
Prime	?	?	✓	.
Index of maximum	–	?	✓	.
Cardinality of intersection	–	?	✓	.
Count inversions	–	?	✓	✓
Quicksort	–	?	?	✓

✓ algorithm can be specified and verified
? algorithm can be specified, but not properly verified
– algorithm can be neither specified nor verified
. not evaluated

Table 4. Computing the factorial of a number

. . . using FPP

```
--!pre:  n >= 0;
result := 1;
--!pre:  result = 1 AND n >= 0;
--!post: result = factorial(n);
--!inv:  result = factorial(i);
FOR i IN 1..n LOOP
   result := result * i;
END LOOP;
```

. . . using KEY

```
public class Fac {
   /* @preconditions n >= 0
      @postconditions result > 0
   */
   public static int fac(int n) {
      if (n == 0) return 1;
      else return (n * fac(n - 1));
}  }
```

. . . using PD

```
function factorial(n: nat): nat
decrease n
^= ([n = 0]: 1, []: n * factorial(n - 1))
via
   var tot: nat! = 1;
   loop
      var nn: nat! = 0;
      change tot
      keep tot’ = factorial(nn’)
      until nn’ = n
      decrease n - nn’;
      nn! = nn + 1;
      tot! = tot * nn
   end;
   value tot
end;
```

. . . using PVS

```
fac: THEORY
BEGIN
   n, x, y, z: VAR nat

   fac(n): RECURSIVE nat =
      (IF n = 0 THEN 1
       ELSE n * fac(n-1) ENDIF)
   MEASURE (LAMBDA n: n)

   mul_mon: LEMMA FORALL x, y, z:
      (x > 0 AND y > z) IMPLIES x * y > x * z
   fac_pos: LEMMA FORALL x: fac(x) > 0
   fac_inc: LEMMA FORALL x: fac(x + 1) < fac(x + 2)
END fac
```

This simplicity comes at a price. The supported subset of ADA comprises basic constructs only (assignments, if- and loop-statements), the only data types are Boolean and integer, no aggregate data types like arrays are allowed. Moreover, the language for specifying verification conditions is restricted to formulas over a collection of built-in functions; it is difficult or impossible to specify program behaviour beyond the functions supported by the prover.

These limitations lead to the results in Table 3: FPP handles well all integer problems whose behaviour can be specified. This includes factorials and even Fibonacci numbers since they are built in; see e.g. the specification in Table 4, where the result of the program is simply specified as `result = factorial(n)`. Problems involving arrays, however, can be neither specified nor implemented.

The work with FPP exhibited some practical problems. FPP assumes the program to be syntactically correct and well-typed; errors are silently ignored and may lead to unexpected results like meaningless preconditions. Formulas constructed by FPP are not simplified, hence the output (pure ASCII) can be more difficult to read than necessary. In the case of failed proof attempts—either because the specification is incomplete or because of prover limitations—the system gives no hints regarding the source of the problem. Finally, implementing FPP as a web service also has drawbacks: it is not always available, like at the time of preparing this article.

The key project The main evaluation of the system took place in 2004 and 2005. At that time, KeY was in alpha stage at version 0.95. Contrary to the other systems it evolved considerably since then, so some of the limitations may no longer apply.

The KEY system needed some effort to get it working. One reason was that Borland's Together Control Center—the commercial CASE tool serving as user interface—requires a license and has to be set up separately. Our evaluation license expired after 30 days and set tight time constraints on the testing schedule.

Writing programs in JAVACARD for the evaluation problems is easy for anyone familiar with JAVA. The bottleneck is OCL for specifying pre- and postconditions; it is not expressive enough to capture the behaviour of the programs. Constructs like `allInstances` or `Set{1..n}` are not yet supported. Hence the specification and verification of the more complex examples involving arrays failed. For example, the postcondition of the KEY implementation in Table 4 just states that the result of the program is positive; a complete specification of the factorial function against which the program can be verified is not possible at the moment.[3]

While the front-end of KEY is quite familiar to software engineers, its back-end is not. The KEY prover is an interactive tool with support for tactics and strategies which automate part of the reasoning. Directing the prover and interpreting the voluminous output requires a thorough understanding of the underlying dynamic logic and the proof rules.

The situation has recently changed with KEY version 1.0, which supports the Java Modeling Language (JML). On the one hand KEY now no longer depends on the CASE tool which is required to handle UML and OCL, on the other hand JML is more expressive than OCL. However, the current support of JML is not yet sufficient for full program verification, but rather allows the user to check light-weight properties of classes and methods. JML annotations in the program (like loop invariants) are also useful for guiding the prover and thus constitute one more step towards automation of proofs.

Perfect developer The evaluation was initially performed with version 2.00 of Perfect Developer and later partially repeated with version 3.03. Installation under Windows and under recent Linux distributions like Redhat or Debian is straightforward. The changes since version 2.00 mostly concern the theorem prover. It has become more powerful in the sense that it is now able to discharge more proof obligations, and also has become more user friendly in the case of failed proofs as it now gives hints how to correct the specification.

PD comes with its own object-oriented programming language called Perfect, which on the one hand excludes features of known languages that cause problems for verification, and on the other hand includes mathematical data types like unbounded integers. A rich collection of built-in data types, classes, functions, and theories allows the user to write concise specifications on a fairly abstract level. UML class diagrams can be imported to generate the skeleton of classes automatically. People acquainted with JAVA or C++ will learn Perfect in a short time; the web site of PD offers an online tutorial.

As an example, consider the PD program for computing the factorial (Table 4). The recursive, functional specification in the third line is refined by an iterative program following the keyword `via`. The loop construct states that `nn` is a variable local to the loop (initialised with zero) and that `tot` is the only other variable changed by the loop. Moreover, the keyword `keep` introduces the loop invariant, `until` the loop condition, and `decrease` the termination function. The loop body increments `nn` by one and multiplies `tot` by `nn`.

Using PD it is possible to specify and verify (automatically) all benchmark problems except the last one (see the corresponding column in Table 3). The verification of the last problem requires induction, which is not supported by the prover. The prover output is quite intuitive, mixing natural language explanations with formal logic. However, at least a basic knowledge of formal logic is required to be able to interpret the prover output and to use it for detecting errors in the specification or in the program.

[3] As a workaround one could specify the postcondition as a formula of the dynamic logic that is used internally by KEY. This is no option, however, for an introductory course and its tight time constraints.

Although Perfect Developer is already quite perfect, there are still some things that could be better. As mentioned above, the prover currently does not support induction. Consequently certain recursive functions and loops cannot be verified by the system; e.g., if multiplication is specified recursively by iterating over the first argument but is implemented as a loop over the second argument, the loop invariant cannot be proved automatically since the proof involves induction. As a workaround one could specify the inductive theorem as an axiom to the system, and prove the theorem outside the system. In practice this limitation is not as severe as it may seem. The principle of proof-by-contract generates dozens of proof obligations even for toy programs, most of which are rather trivial and therefore are verified by PD's prover automatically; the user can focus his attention on a few unproven assertions resulting from the core algorithms.

Another weakness, at least from an academic point of view, is the lack of information concerning the internals of the prover. Ideally the logical rules used in correctness proofs should be open for inspection such that independent proof checkers can establish additional trust in the system. Finally, the prover does not support all specification methods equally well. It may happen that a natural and obvious specification of an algorithm leads to unprovable assertions, whereas a less obvious one using different built-ins succeeds. Ideally, the user should be able to formalise the informal specification of a problem as directly as possible without paying attention to the prover.

Prototype verification system We used PVS version 3.2 for our evaluation; the current version is 4.0. Since our problems scratch only the surface of what PVS can do the precise version does not matter.

PVS is a powerful interactive proof assistant, which is versatile and offers many possibilities. It is automatic to a certain degree, but usually requires frequent user interactions. We tried only some of the problems since it soon turned out that PVS takes too long to learn to be used by the students in our course. By developing appropriate theories (many are already provided by the PVS community) one can construct proofs for any problem. For example, Table 4 shows a PVS version of factorials. It contains a recursive definition of the factorial function and states two properties of factorials, namely that factorials are always positive (`fac_pos`) and that factorials increase with their arguments (`fac_inc`). To prove the latter we need the monotonicity of multiplication with respect to its second argument (`mul_mon`). The lemmas `fac_pos` and `mul_mon` both can be verified by induction regarding variable `x` and two simplification steps:

```
(induct "x") (grind) (grind)
```

Note that hardly any human intervention is needed. The proof of lemma `fac_inc` is more involved and uses the other two lemmas:

```
(skolem!) (expand "fac" :occurrence 2)
(lemma mul_mon) (inst -1 "fac(1+x!1)" "x!1+2" "1") (prop) (grind)
(lemma fac_pos) (grind) (grind)
```

Lemma `fac_inc` is skolemised and the second occurrence of `fac` is expanded by its definition. Then a particular instance of lemma `mul_mon` is added and the result is propositionally simplified. Finally the lemma `fac_pos` is used to establish the precondition of lemma `mulmon`.

5. The verdict

FPP is a valuable tool for illustrating the basic ideas of formal program verification in courses, where Hoare-style verification is just one of many topics and where concepts like postconditions and invariants are introduced for the first time. Programs are limited to those involving just integers, but no arrays. Before using FPP in class we recommend to install the system locally to be safe from server failures in Jena; this probably requires negotiations since the tool is not offered for download by default. FPP is too simple, however, for advanced courses based on books like [Gri87], which discuss many examples using arrays and various types of quantifiers.

For such advanced courses, PD is the best choice among the four tools at the moment. It offers a high-level language, which can be learned within the time constraints of a course. The high degree of proof automation lets the students concentrate on the primary goals of the course, namely writing correct specifications and refining them. The notoriously difficult problem of providing feedback on failed proof attempts is solved sufficiently well to give at least some help in tracking down errors.

Table 5. Simple assignments

Problem	Description
Card. of intersection	Given two sorted lists of integers, compute the cardinality of their intersection in linear time
Merging of lists	Given two sorted lists of integers, compute a sorted list containing the elements of the original lists in linear time
Minimal distance	Given two sorted lists of integers, A and B, compute the minimum of abs($A[i]$, $B[i]$) for all i, j in linear time
Longest plateau	Given a sorted list of integers, determine the length of the longest sublist of identical elements in linear time
Index of maximum	Given a list of integers, determine the index of a maximal element in linear time
Sortedness	Check in linear time whether a list of integers is sorted

Table 6. More difficult assignments

Problem	Description
Count inversions	Given a list, A, of n integers, count the pairs (i, j) of indices satisfying $i < j$ and $A[i] > A[j]$ in time $O(n \log n)$
Count Boolean inversions	Given a list, A, of Booleans, count the pairs (i, j) of indices satisfying $i < j$, $A[i]$ and $\neg A[j]$, in linear time
Length of longest monotone sublist	Given a list of integers, compute the length of the longest non-increasing or non-decreasing sublist in linear time
Length of longest left-minimal sublist	Given a list of integers, compute in linear time the length of the longest sublist, where the first element of the sublist is minimal among the elements of the sublist

The KEY system makes rapid progress and will be a serious alternative as soon as the prover offers a similar degree of automation and user friendliness as PD's prover. The use of languages like JAVACARD, UML, OCL, and JML, which are familiar to most students, saves time that can be spent on more advanced examples.

PVS requires a long learning phase due to its many basic inference rules and proof tactics, and users of PVS need a firm background in mathematics and formal logic to guide the prover. This makes PVS unsuitable for any regular course with its tight time constraints and the limited mathematical skills of average students. It might be the right tool for selected students at master or Ph.D. level.

6. Perfect developer in class

So far we have used PD twice in a lab on formal software verification, for which the students are supposed to spend 75 working hours on average. The course can be chosen by students of informatics and business informatics as elective subject.

The first assignment consisted in installing Perfect Developer under Windows or Linux, and in getting acquainted with the system by working through the online tutorial offered on the web site of *Escher Technologies*, the company behind PD. As a check we asked the students to write a report listing the errors in the tutorial; there are a few, which can be easily corrected provided the assignment is taken seriously.

The second, third and fourth assignment consisted in selecting two problems from a list of six easy and one problem from a list of four harder problems (Tables 5, 6). The students had to write a formal specification in Perfect, refine it to an executable function and verify the correctness. The algorithms were required to run in time $O(n)$ or $O(n \log n)$, since in many cases PD is able to generate code automatically from the specification which for these examples has quadratic or cubic runtime. The students had to write a report including their errors, solutions, and the time they had spent on the problem. In case they were not able to get PD to verify their solution completely they had to provide formal arguments explaining why their specification and implementation was correct nevertheless.

We used the collected data to answer two questions: (1) Are 75 h sufficient for learning to use the system and doing the assignments? (2) How relevant are algorithmic, formal, and mathematical skills as prerequisite for doing the assignments? We separated the students into two groups: those doing a bachelor or master program in informatics, and those doing a bachelor or master in business informatics. In the curriculum valid until 2006, the latter had a few hours less on math, algorithmics and programming than their colleagues in pure informatics.

The statistical analysis is summarised in Figs. 1 and 2. Students in informatics needed less time on average and got better grades than their colleagues in business informatics. All students finished the assignments in time;

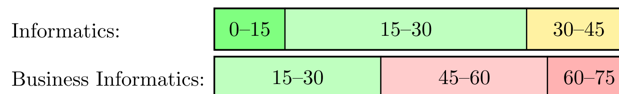

Fig. 1. Time needed for assignments (hours)

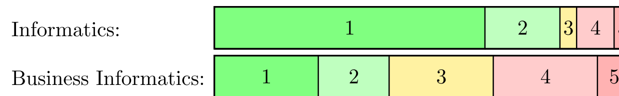

Fig. 2. Final grades (*1* excellent, *5* failed)

those studying informatics needed only half of the allotted time. This shows that even though Perfect Developer is new to the students, there is sufficient time to get acquainted with the system to solve the assignments. In fact, there is even time for more assignments. Moreover, the intuition is confirmed that the ability of applying formal methods correlates with the amount of algorithmic and mathematical training on bachelor level.

Acknowledgments

We are grateful to two anonymous referees for their detailed and constructive criticism. Their advice lead to substantial improvements.

References

[ABB+05] Ahrendt W, Baar T, Beckert B, Bubel R, Giese M, Hähnle R, Menzel W, Mostowski W, Roth A, Schlager S, Schmitt PH (2005) The KeY tool. Softw Syst Model 4(1):32–54
[BHS07] Beckert B, Hähnle R, Schmitt PH (eds) (2007) Verification of object-oriented software. The KeY approach. In: Lecture notes in artificial intelligence, vol 4334. Springer, Heidelberg
[CGP00] Clarke EM, Grumberg O, Peled DA (2000) Model checking. MIT Press, Cambridge
[Cro03] Crocker D (2003) Perfect developer: a tool for object-oriented formal specification and refinement. Tools exhibition notes at formal methods Europe
[CZ92] Clarke EM, Zhao X (1992) Analytica—a theorem prover in Mathematica. In: Kapur D (ed) Proceedings of 11th international conference on automated deduction (CADE'92). LNCS, vol 607. Springer, Heidelberg, pp 761–765
[Dij76] Dijkstra E (1976) A discipline of programming. Prentice-Hall, Englewood Cliffs
[DM05] Denning PJ, McGettrick A (2005) Recentering computer science. Commun ACM 48(11):15–19
[Fei05] Feinerer I (2005) Formal program verification: a comparison of selected tools and their theoretical foundations. Master's thesis, Technische Universität Wien, Vienna, Austria, January
[FPP] Frege Program Prover. http://psc.informatik.uni-jena.de/Fpp/fpp-intr.htm
[Gri87] Gries D (1987) The science of programming. Springer, Heidelberg
[HR03] Huth MRA, Ryan MD (2003) Logic in computer science—modeling and reasoning about systems. Cambridge University Press, London
[Isa] Isabelle. http://www.cl.cam.ac.uk/research/hvg/Isabelle/
[KEY] The KEY Project. http://www.key-project.org/
[Mat] Mathematica. http://www.wolfram.com/
[Obj05] Object Management Group (2005) Unified modeling language 2.0 Superstructure specification. http://www.omg.org/cgi-bin/apps/doc?formal/05-07-04.pdf
[Obj06] Object Management Group (2006) UML 2.0 Object constraint language specification. http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf
[ORS92] Owre S, Rushby J, Shankar N (1992) PVS: a prototype verification system. In: Kapur D (ed) 11th International conference on automated deduction (CADE). Lecture notes in artificial intelligence, vol 607. Springer, Saratoga, pp 748–752
[Pat05] Patterson DA (2005) Restoring the popularity of computer science. Commun ACM 48(9):25–28
[PD] Perfect Developer. http://www.eschertech.com/products/
[PVS] Prototype Verification System. http://pvs.csl.sri.com/
[SL03] Stiller E, LeBlanc C (2003) Creating new computer science curricula for the new millenium. J Comput Small Coll 18(5):198–209
[SMV] Symbolic Model Verifier. http://www.cs.cmu.edu/~modelcheck/smv.html
[Win97] Winkler J (1997) The Frege program prover, vol 42. Internationales Wissenschaftliches Kolloquium, Technische Universität Ilmenau, pp 116–121

Received 26 March 2007
Accepted in revised form 23 April 2008 by D. A. Duce, J. Oliveira, P. Boca and R. Boute
Published online 11 June 2008