Automated Tools for Teaching Formal Software Verification

Ingo Feinerer Gernot Salzer
Institut für Computersprachen
Technische Universität Wien
Karlsplatz 13
A-1040 Vienna, Austria
{*feinerer*|*salzer*}*@logic.at*

Abstract

We present the status of formal methods at our university, and describe our course on formal software verification in more detail. We report our experiences in using Perfect Developer for the course assignments.

Keywords: formal verification of software, Perfect Developer

1. INTRODUCTION

As the automation of formal methods makes progress, they become more and more relevant for industry. Unfortunately, other areas are growing faster and are more popular with students, university teachers, and professionals – at least in Austria.

Computer Science @ TU Wien The computer science department at *Technische Universität Wien* (TUW, Vienna University of Technology) is by far the biggest in Austria. About 120 full, associate, and assistant professors (5 % of TUW's teaching personnel) teach approximately 6000 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 6 bachelor[1] and 9 master[2] degrees taking three and two years, respectively.

	Duration	Math & Statistics	Theory & Logic
Diploma study (up to 2001)	5 yrs/324 Ects[3]	34.5 Ects (10.6%)	19.5 Ects (6.0%)
Bac.+Mast. (2001 onwards)	3+2 yrs/300 Ects	21.0 Ects (7.0%)	12.0 Ects (4.0%)

TABLE 1: Comparison of old and new CS curricula

Formal Methods @ TU Wien 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; a second course is now focussed on applied logics along the lines of the book *Logic in Computer Science* by Huth and Ryan [7] (augmented by recent topics like description logics and the semantic web). Apart from these compulsory courses there are only a few elective ones dealing with formal methods, most notably *Computer Aided Verification* focusing on model checking [2], 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

[1]Business Informatics, Computer Engineering, Data Engineering & Statistics, Media Informatics, Medical Informatics, Software & Information Engineering
[2]Business Engineering and Informatics (*Wirtschaftsingenieurwesen Informatik*), Business Informatics, Computer Engineering, Computer Graphics & Digital Image Processing, Computational Intelligence, Information & Knowledge Management, Media Informatics, Medical Informatics, Software Engineering & Internet Computing
[3]Ects = ECTS credit, where ECTS means European Credit Transfer System; 1 year corresponds to 60 Ects, 1 Ects equals 25 working hours.

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.

2. FORMAL VERIFICATION OF SOFTWARE

The course *Formal Verification of Software* has been taught for two decades, but until recently only theoretically in the style of Dijkstra [4] and Gries [6]. The slow but steady progress in the automation of program verification led us in 2003 to evaluate several tools in order to select one for use in the course. [5] All of them can be used free of charge for academic and teaching purposes.

Frege Program Prover (FPP) [9] FPP supports a small subset of Ada consisting of typical imperative program structures like loop-, case- and if-statements. The only data types available are integer and boolean. The language for specifying pre- and post-conditions is rather restricted. E.g., function definitions, recursive specifications and structured data types like arrays are not supported. FPP is able to verify simple programs and to compute their weakest pre-conditions. The prover, Analytica, acts as a black box signalling either the validity of a formula or returning unprovable sub-formulas; formal proofs are not supplied.

Due to its simplicity and its web interface, FPP is easy to learn and use. It seems to be a valuable tool for illustrating the ideas of formal program verification in basic courses. It is not suitable, however, for advanced courses on the subject or for real world applications. Neither is it able to deal with standard examples from Dijkstra and Gries involving arrays, nor does it support object-oriented features. Moreover, the terse output in pure ASCII makes it difficult to trace errors.

The KeY system [1] The KeY system supports a subset of Java known as JavaCard, which is increasingly used for mobile and embedded devices. Verification is based on dynamic logic, a generalization of the Hoare calculus. The system cooperates with CASE tools like *Together Control Center*, *Solo* or *Architect*. Objects and constraints can be specified using UML and OCL.

Java, UML, OCL, and CASE tools are familiar to software engineers and students alike, which helps in getting started. Nevertheless, KeY cannot be recommended for such target groups at present: the interactive prover and its interaction with the user are in their infancy and are inadequate for serious use. Moreover, OCL is not expressive enough to specify complex program behaviour. Considering that KeY is still in alpha stage, it seems to be worthwhile to reevaluate the system in a few years in order to see whether it lives up to expectations.

Perfect Developer (PD) [3] PD consists of a full-fledged object-oriented programming language called Perfect, of a powerful automated theorem prover, and of a code generator translating programs from Perfect to Java, Ada, and 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. UML class diagrams can be imported to generate the skeleton of classes automatically.

PD is a technically mature product that is ready for use in a regular development process. However, software engineers will need some time to become sufficiently aquainted with the many features of Perfect. Moreover, 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. Perfect Developer is also well suited for teaching advanced courses on formal program verification. Usually there will not be enough time to cover all features of Perfect. Therefore a tutorial is required that concentrates on just those elements of the language that are necessary to implement and verify instructive examples like those in [6].

PD is that tool of the four that comes closest to the ideal of automated program verification. But there are still some shortcomings. One is that the prover currently does not support induction. Consequently certain recursive functions and loops cannot be verified by the system. E.g., if one specifies multiplication recursively by iterating over the first argument but decides to implement it as a loop over the second argument, the loop invariant cannot be proved automatically since the proof involves induction; in such a case the user has to provide a lemma whose correctness

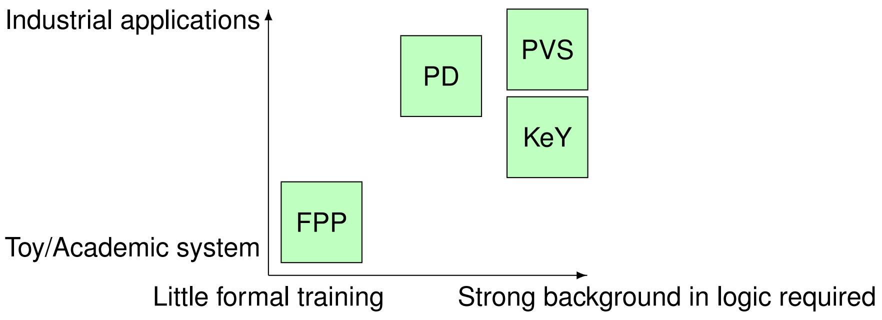

TABLE 2: Comparison of FPP, KeY, PD and PVS

has to be verified outside the system. In practice this limitation is not as severe as it seems: 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 builtins 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 (PVS) [8] PVS is a powerful interactive theorem prover, which has been used for various real world applications. In contrast to the other systems it does not generate verified program code, but it proves properties of algorithms. The prover is versatile and offers many possibilities. It is automatic to a certain degree, but usually requires frequent user interactions.

Due to its many basic inference rules and tactics it takes a long learning phase to become familiar with the system. Morover, users of PVS need a firm background in mathematics and formal logic to guide the prover. In our opinion typical software engineers and average students of computer science will have a hard time using PVS. Graduate or Ph.D. students might have a chance, provided they are given enough time. For courses with just a few hours per week in the lab PVS seems to be too complex.

Table 2 compares the four selected tools FPP, KeY, PD, and PVS regarding formal background in logic and application area. Based on this evaluation we decided to use Perfect Developer for our course in formal software verification.

3. OBSERVATIONS

The first assignment in the lab on formal program verification was to install Perfect Developer under Windows or Linux, and to get aquainted 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. This assignment posed no difficulties for the students; the errors found varied between zero and several dozens.

The second, third and fourth assignment consisted of selecting two problems from a list of six easy and one problem from a list of four harder problems (see the appendix for a short description of the problems). 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 $O(n \log n)$, respectively. 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

Informatics:	0–15	15–30	30–45
Business Informatics:	15–30	45–60	60–75

TABLE 3: Time needed for assignments (hours)

Informatics:	1	2	3	4	5
Business Informatics:	1	2	3	4	5

TABLE 4: Final grades (1=excellent, 5=failed)

they had to provide formal arguments explaining why their specification and implementation was correct nevertheless.

Not quite surprisingly, the gathered data indicate that students with a knowledge of mathematics and some experience in software engineering performed significantly better than students with less background in these areas. This observation is confirmed statistically by separating 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 one of the studies in informatics. Table 3 and Table 4 show that on average students in informatics needed less time and got better grades than their colleagues in business informatics.

4. CONCLUSION

Using automated tools for teaching formal methods helps to convince students that formal methods work. Nevertheless, formal methods still require the ability to think in an abstract and declarative manner and to analyse formal proofs. According to our experience a thorough education in mathematics and logic as well as a training in different programming paradigms forms a sound basis.

REFERENCES

[1] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4(1):32–54, 2005.

[2] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

[3] David Crocker. Perfect Developer: A tool for Object-Oriented Formal Specification and Refinement. *Tools Exhibition Notes at Formal Methods Europe*, 2003.

[4] Edsger Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[5] Ingo Feinerer. Formal program verification: A comparison of selected tools and their theoretical foundations. Master's thesis, Technische Universität Wien, Vienna, Austria, January 2005.

[6] David Gries. *The Science of Programming*. Springer, 1987.

[7] Michael R.A. Huth and Mark D. Ryan. *Logic in Computer Science – modelling and reasoning about systems*. Cambridge University Press, 2003.

[8] Sam Owre, John Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[9] Jürgen Winkler. The Frege Program Prover. *42. Internationales Wissenschaftliches Kolloquium, Technische Universität Ilmenau*, pages 116–121, 1997.

APPENDIX: PROBLEMS TO BE VERIFIED WITH PERFECT DEVELOPER

Easy Problems

Cardinality 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 $\mathrm{abs}(A[i], B[i])$ 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.

Slightly harder problems

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.