Embedded Systems Laboratory Technical Report
ESL-TIK-00214

Formal Software Verification: Model Checking and Theorem Proving

Martin Ouimet
Embedded Systems Laboratory
Massachusetts Institute of Technology
Cambridge, MA, 02139, USA
mouimet@mit.edu

Embedded Systems Laboratory Technical Report
ESL-TIK-00214

This page is intentionally left blank.

Formal Software Verification: Model Checking and Theorem Proving

Martin Ouimet

Technical Report ESL-TIK-00214
Embedded Systems Laboratory
Massachusetts Institute of Technology
Cambridge, MA, 02139, USA
mouimet@mit.edu

1 Software verification

Formal verification of hardware and software systems has gained popularity in industry since the advent of the famous "Pentium bug" in 1994, which caused Intel to recall faulty chips and take a loss of $475 million [1]. Since this event, formal verification of hardware systems has been commonplace using mostly model checkers but also using theorem provers [2], [3]. The benefits reaped in the hardware sector has led the software sector to consider whether similar benefits could be achieved in the context of program correctness. Proofs of correctness about computer programs have been around since the early days of computer science, but academic developments were routinely ignored by industry citing advances in research as "impractical" [4]. While there are drastic differences between the properties of software and the properties of hardware, namely the strict structure of hardware, the inherently finite state of hardware, and the restricted size of hardware [5]. While there are doubts about whether the anecdotal success of formal verification of industrial hardware can be replicated in the software sector, some progress has been made though numerous challenges still remain [2].

Before embarking on the description of theorem proving and model checking approaches in the context of verifying program correctness, the correctness problem of software is formally defined. Verifying the correctness of a program involves formulating a property to be verified using a suitable logic such as first order logic or temporal logic [6]. Sample properties typically take the form of a predicate over variable values. For example, a property stipulating that a variable x be positive and that a variable y be strictly smaller than x can be formulated in the following way:

$$x > 0 \wedge y < x$$

When assessing the correctness of the program, two distinct approaches using properties are used - *pre/post condition* and *invariant assertion*. Pre/post condition approaches formulate the correctness problem as the relationship between a formula that is assumed to hold at the beginning of program execution, denoted ϕ_{PRE}, and a formula that should hold at the end of program execution, denoted ϕ_{POST}. Assessing the correctness of the program involves determining whether the semantics of the program

establishes ϕ_{POST} given ϕ_{PRE}. Given the undecidability of the halting problem, *partial correctness* is often used, by assuming termination. Approaches based an invariant assertion define correctness of a program as an invariant formula, ϕ_{INV}, which must be verified to hold throughout the program execution. Invariants can be specified by the user, denoted a *specification*, or can be automatically inferred from the program code. For example, specifications that can be inferred automatically from program code are runtime errors, that is, errors that are undesirable by definition. These types of errors include out of bound array accesses and null pointer accesses. A proof of correctness about bounded array access would infer the upper bounds and lower bounds of all variables used as array indices and would verify that the bounds do not go outside the definition of the array. Proofs of correctness are typically achieved through the derivation of a theorem. However, software verification can also be achieved without mathematical proofs. A popular approach to formal verification in the hardware industry, called *model checking*, is increasingly being used to verify software.

1.1 Software verification through model checking

The model checking problem involves the construction of an abstract model $\mathcal{M}$, in the form of variations on finite state automata, and the construction of specification formulas ϕ, in the form of variations on temporal logic [6]. The model checking verification problem involves establishing that the model semantically entails the specification:

$$\mathcal{M} \models \phi$$

This approach has been pioneered by Clarke et al. in a seminal paper published in 1983 [7]. 15 years later, Clarke published and indispensable book on the topic summarizing advances and challenges since the early 1980s [8]. The verification algorithm used in model checking involves exploring the set of reachable states of the model to ensure that the formula ϕ holds. If ϕ is an invariant assertion, the model checking approach explores the entire state space to ensure that the formula holds in all states. In order to guarantee termination, such an approach requires that the set of reachable states be finite. For software programs that involve floating point arithmetic, model checking can still be used because floating point units use fixed decimal arithmetic. However, performing software verification by model checking involves translating the program code into finite state automata [9]. The model can be used before the program code is written, to verify properties of the model before committing to implementation. The model can also be automatically extracted from program code using annotations in the code. For example, an annotated subset of the C language can be verified using the SPIN model checker [10]. Verification of software by model checking is well-suited for "control-intensive" applications where the majority of the software code is written in the form of control structures (e.g., `if` statements) operating on simple datatypes (e.g., `int` variables). Such applications include communication protocols and embedded controllers [6]. Furthermore, verification by model checking has gained popularity in industry because the verification procedure can be fully automated [7] and counterexamples are automatically generated if the property being verified does not hold.

When programs contain complex datatypes such as trees, lists, and recursive definitions, analysis by model checking is difficult to achieve [2]. Reasoning about complex structures and recursive constructs typically involves reasoning through mathematical induction, a proof approach which cannot be automated [11]. Furthermore, model checkers rely on exhaustive state space enumeration to establish whether a property holds or doesn't hold. This approach to verification puts immediate limits on the state space of problems that can be explored by model checkers. This common problem, known as the state explosion problem, is an often cited drawback of verification by model checking [12]. While there have been numerous advances in addressing the state space explosion problem, as cited in [12], the need to perform a search of the state space remains an inherent limitation of verification by model checking.

Nevertheless, the automation of the analysis and the automatic counterexample generation ensures that model checking will remain a popular approach to verification in industry. For example, Microsoft uses a model checking approach to verify that device drivers written in C conform to their API specification. The approach is implemented in the SLAM tool suite [13] and uses a model checker to verify behavior. Device drivers provide an instance of a software program that is well-suited for model checking because device driver code is relatively small, does not contain complex data structures, and has well-defined API requirements. A related approach to verify device drivers written in C is the BLAST software project [14].

The application of model checkers to software verification has been applied primarily to the C language because C lacks complex datatypes as found in object oriented programming, such as inherited types and polymorphic types. To reason about program correctness in the presence of complex types, an approach based on theorem proving is necessary [2].

1.2 Software verification through theorem proving

The calculi used in theorem proving approaches to software verification are variations on themes from two seminal papers. The first one by C.A.R. Hoare describes a calculus to reason about program correctness in terms of pre and post conditions [15]. E. G. Dijkstra extended Hoare's ideas in the concept of "predicate transformers" which, instead of starting with a pre condition and post condition, starts with a post condition and uses the program code to determine the pre condition that needs to hold to establish the post condition [16]. Hoare's approach to proving correctness introduced the concept of a "Hoare triple", which is a formula in the following form:

$$\{\phi_{PRE}\}P\{\phi_{POST}\}$$

This formula can be read as "if property ϕ_{PRE} holds before program P starts, ϕ_{POST} holds after the execution of P. The program P can refer to an entire program or to a single function call, depending on the unit that is being verified. In Hoare's calculus, axioms and rules of inference are used to derive ϕ_{POST} based on ϕ_{PRE} and P. The syntax of P described by Hoare corresponds to a simple imperative language with the usual constructs (assignment, conditional branching, looping, and sequential statements). A sample inference rule is shown below for an if statement:

$$\frac{\{\phi_1 \wedge B\}C_1\{\phi_2\} \qquad \{\phi_1 \wedge \neg B\}C_2\{\phi_2\}}{\{\phi\}\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2\{\phi_2\}}$$

Theorem provers used to prove program properties are based on variations of Hoare logic. However, the language of theorem provers is typically a dialect of LISP, based on the seminal paper by McCarthy [17]. A key difference between the theorem approach to software verification and the model checking approach to software verification is that theorem provers do not need to exhaustively visit the program's state space to verify properties [8]. Consequently, a theorem prover approach can reason about infinite state spaces and state spaces involving complex datatypes and recursion. This can be achieved because a theorem prover reasons about constraints on states, not instances of states. Theorem provers search for proofs in the syntactic domain, which is typically much smaller than the semantic domain searched by model checkers. Consequently, theorem provers are well-suited for reasoning about "data-intensive" systems with complex data structures but simple information flow. Although theorem provers support fully automated analysis in restricted cases only [18], a mature theorem proving system can provide an acceptable level of automation [19]. Reasoning about inductive structures of arbitrary size (e.g., trees, lists, or stacks) can be achieved through mathematical induction but cannot be automated [11]. Nevertheless, this tradeoff is acceptable in certain instances since this type of analysis cannot be performed by model checkers, but is still vital to the verification effort. In certain instances, lack of automation can be tolerated for increased capabilities.

While theorem provers have distinct advantages over model checkers, namely in the superior size of the systems they can be applied to and their ability to reason inductively, deductive systems also have their drawbacks. The proof system of a theorem prover for a system of practical size can be extremely large [2]. Furthermore, the generated proofs can be large and difficult to understand. Industrial reports of verification using theorem provers have described situations where a theorem prover proof required 25 megabytes of memory to print [3]. An often cited drawback of theorem provers is that they require a great deal of user expertise and effort [20]. This requirement presents perhaps the greatest barrier to widespread adoption and usage of theorem provers. This point is treated in more details in the following sections.

ESC/Java, is an example of a widely available system for program correctness based on theorem proving [21]. ESC/Java hides most of the details of theorem proving interactions from the user and issues "warnings" to the user when a property cannot be inferred to hold. However, these warnings are provided in a fashion similar to a compiler. The user can discard the warnings by annotating the Java source code with ESC specific annotations, which take the form of predicates over the values of variables. The theorem prover used behind the scenes is the Simplify theorem prover, originally developed at the Compaq research center [19]. The type of reasoning performed with ESC/Java can take the form of Hoare triples, invariant assertion, and precondition derivation. The ESC/Java system presents an interesting application of theorem proving to software verification, all the while stressing usability. The usability of ESC/Java is further analyzed in the Section 3.

1.3 Conclusion

Although theorem proving and model checking appear to be contradictory approaches to software verification, there has been considerable effort in the past 15 years to incorporate model checking and theorem proving [22], [23]. Because theorem provers and model checkers each provide complementary benefits in terms of automation and scalability, it is likely that this trend will follow and that model checkers will continue to be useful on systems or manageable size while theorem provers will be used on large systems [9].

2 Humans and interactive theorem proving

The undecidability of validity in first order predicate logic implies that automated theorem proving using a first order theory cannot be fully automated [24]. In practice, the theoretical results require that a human must be "in the loop" to derive non-trivial theorems. The term "interactive theorem proving" is used to denote a theorem proving system that requires human intervention. The human interacts with the theorem prover in a variety of ways. First and foremost, the human is responsible for representing and encoding the problem domain so that useful results can be derived by the proof system. While this issue is an important issue, we do not discuss it at length since any approach to formal verification requires a significant initial investment. This is true in of approaches based on model checking, theorem proving, or based on some other form of static analysis. While the initial investment required by theorem proving might appear larger than for other approaches, there is no empirical evidence to support this claim. The need to encode the problem domain in the theorem proving system is directly related to the usability issues of theorem provers and is discussed in the Section 3.

The human plays an important role in a theorem proving system because the human needs to guide the theorem prover in its search for a proof. The guidance takes the form of setting intermediate lemmas that should be proved on the way to the final proof [11], as well as selecting heuristics and strategies at various steps of the proof [20]. Kaufmann and Moore argue that it is best to think of theorem provers as *proof checkers*: the user describes a particular proof and the system checks that it is correct [2]. Furthermore, the authors relate their experience with developing proofs with the ACL2 theorem prover. They make an interesting point when discussing situations where they have had difficulty finding a proof; they mention the need to step back from the proof strategy and to reason intellectually or manually about possible counterexamples. They conclude by suggesting that if a counterexample cannot be generated, it is highly likely that the followed proof strategy may be flawed. Such mathematical thinking is necessary when a human guides the theorem prover in its search for a proof because the human needs to decide which intermediate subgoals to set, which lemmas to prove, and which rules of inference to select in the case of non-determinism [18]. The human operator needs to decide *what* needs to be proved and *how* to prove it. An interactive theorem proving system is a parallel process between human mathematical reasoning and computer support for proof checking.

Moreover, because theorem provers are often used to reason inductively, the theorems to be proved need to be formulated as formulas involving mathematical induction. Formulating the theorems is the responsibility of the theorem prover user and requires quite a bit of ingenuity [25]. When Kaufmann referred to theorem provers as *proof checkers*, he unveiled the need for the human to fully understand the reasoning steps of the theorem prover. Thinking of theorem provers in this way means that theorem provers do not "discover proofs" but help automate and check some of the steps that would normally be performed manually [18]. This approach to theorem proving is more realistic because it thinks of theorem provers as *proof assistants* and provides a rigorous means for verification. However, a proof assistant is not a "push button" approach as is the case in model checkers. The human interaction with theorem provers goes beyond understanding the proof procedure followed by the prover. Once a prover has successfully derived a proof, the human user needs to gain insight into the derived proof to ensure that it adequately reflects the desired property. Proofs using theorem provers can be quite long, spanning hundreds of pages [3]. In order to gain full benefits from the use of an automated theorem prover, the human needs to fully appreciate the steps of the proofs and the intermediate lemmas. As Kaufmann eloquently expresses, "all proofs of commercially interesting theorems completed with mechanical theorem proving systems have one thing in common: they require a great deal of user expertise and effort [2].

Theorem provers provide a rigorous and reliable approach to proofs of correctness, enabling establishing correctness of properties over infinite domains, properties about complex data structures, and properties of recursive structures, as described in Section 1. However, the approach relies heavily on the expertise and diligence of the human assistant for guidance in deriving a proof as described in this section. The need to involve a human brings about a set of usability issues which have plagued theorem provers since their inception [26]. These usability issues are analyzed in the following section.

3 Usability hurdles to widespread adoption of theorem proving

The need to involve a human in the use of a theorem prover places part of the burden of successful theorem proving on the human. In this section, we analyze the usability hurdles which need to be overcome for theorem provers to find widespread adoption, especially in industry. As mentioned in the Section 2, proving meaningful theorems involves significant user effort and expertise. A hurdle to usability is the size of the proof systems and the size of the proofs that are being derived using mechanical provers. As a sample case study, a verification approach conducted at Motorola utilized a proof system with relevant concepts spanning hundreds of pages [3]. Furthermore, on the way to the proof of a theorem, it is possible that hundreds and potentially thousands of lemmas must be proved along the way [11]. In the Motorola case study, the size of a proof of a meaningful theorem is stated to require 25 megabytes of memory to print [2]. Since proofs are sequences of textual characters based on derivatives of the LISP functional language, 25 megabytes of text can easily span a hundred pages. The size of the proof systems and the size of meaningful proofs are a major usability hurdle,

but it is generally understood that doing anything meaningful using a theorem prover requires a high level of complexity and lengthy proofs [11].

A hurdle to usability not related to the size of proofs and proof systems is the language used in the reasoning of theorem provers. Most theorem provers use an input and reasoning language based on the functional language LISP. The reason why LISP is used stems from the early days of theorem proving when seminal progress was made using LISP [17]. LISP is a functional language where parentheses abound, function calls are curried, and operations are often written in prefix notation. The need to understand hundreds of pages of concepts in the proof system and the need to understand proofs that could span one hundred pages create a sizable usability hurdle. This hurdle is exacerbated by the fact that the lengthy proof concepts and proofs are expressed in a functional language. The use of functional languages in theorem provers has been identified as a key usability hurdle and various approaches have been suggested to improve usability [26].

Another hurdle plaguing theorem proving systems is the rampant lack of user interfaces. Since most of the theorem provers are developed in academic laboratories, the tool support for theorem provers is typically provided in the form of text files and command-line tools [27]. Because the responsibility of academic research typically focuses on developing novel theory, it is unlikely that academia will provide commercial quality tools that meet the needs of industry [28]. Theorem provers such as HOL and PVS have found traction in industry, and commercial entities have started to develop appropriate tool support. The situation exposed in [27] has been addressed is still being addressed today. The efforts have yielded theorem provers with user interfaces to ease the representation of proof steps, to ease the selection of proof strategies, and to ease the readability of proofs [29]. The need for better user interfaces will likely be solved by market demands since commercial quality tools are likely to be developed if there is enough demand for such tools.

The hurdle of using a LISP-like language and the large number of proof steps for meaningful proofs is also being addressed in the academic community. The term "application specific theorem proving" is used to denote an approach to theorem proving which is tailored to a special application domain [30]. An application specific theorem prover attempts to hide the complexities of the input language of the theorem prover and the detailed steps of proofs. This is achieved by building a high level interface on top of a general-purpose theorem prover. The high level interface contains an input language specific to the application domain and a set of proof steps that are also application domain specific. For example, the TAME interface was built on top of the PVS prover to facilitate proofs about Timed Input/Output Automata (TIOA) models [31]. The language of the TAME interface adequately mirrors the language used to write TIOA models. Furthermore, many of the low-level proofs steps of the PVS prover are hidden in the TAME interface and appear as a single proof step to the user.

In the context of software verification, the ESC/Java system mentioned in Section 1 is another example of application specific theorem proving. When using ESC/Java, the user is unaware that the reasoning behind the scenes is performed using the Simplify theorem prover [19]. Interaction with the user is provided in the form of warnings, which are presented to the user in the form of compiler messages. The user can dismiss

the warnings by adding annotations in the Java source code. In ESC/Java, the proof guidance is provided through annotations to discard warnings without referring to axioms or rules of inference. The usability of ESC/Java in light of other theorem proving systems is surprising because ESC/Java does not require the user to select of understand the underlying proof strategy. The usability is achieved by restricting the types of proofs that can be performed in ESC/Java which are limited to pre state, post state, and invariant assertion on simple types without induction. Furthermore, ESC/Java does not strive for completeness or even for soundness, as explained in [32]. The soundness and completeness goals were abandoned as a way to increase usability by discharging numerous proof obligations that require involved human guidance.

Application-specific theorem proving is following a growing trend to "embed" formal verification techniques into compiler technology and to hide formal verification techniques from the user [9]. So far, the results have been mixed for the TAME interface to PVS [33] but have been promising for restricted systems such as ESC/Java [32]. In the end, since theorem proving systems are really computerized proof assistants for human manual proofs (and not the other way around), it is unlikely that theorem provers will ever become readily usable by non-experts. Usability by non-expert can be achieved for restricted classes of systems like ESC/Java, but probably will not be achieved for arbitrary proof systems. Furthermore, proofs of correctness about complex systems are by definition complex and require the users to think mathematically. Consequently, it is unlikely that theorem provers will become a "push button" approach in the spirit of model checkers.

As D. L. Parnas argues in an opinion paper about formal methods adoption in industry, software engineers routinely use mathematics in their day to day activities, but they rarely *invent* new mathematics [34]. What Parnas referred to is that proofs of correctness involve creating new mathematics and hence requires a high level of expertise typically not prevalent in software engineering curriculum. Jonathan Bowen, another strong supporter of formal methods, supports a similar view, reiterating the opinion that proofs about complex systems are inherently complex, prompting organization who use formal methods to keep a "formal methods guru" on hand [35]. While these attitudes do not bode well for the widespread adoption of theorem provers in software verification outside of expert groups, the situation does not mean that software will never be correct or reliable. C.A.R. Hoare, perhaps the most famous adept of formal verification through mathematical reasoning, argued that there are numerous ways to achieve reliable software; formal proofs of correctness represent only one of many possible options on the road to reliable software [36].

References

1. Coe, T., Mathisen, T., Moler, C., Pratt, V.: Computational Aspects of the Pentium Affair. IEEE Journal Computational Science and Engineering
2. Kaufmann, M., Moore, J.S.: Some Key Research Problems in Automated Theorem Proving for Hardware and Software Verification. In: Spanish Royal Academy of Science (RAMSAC). Volume 98. (2004) 181–196

3. Brock, B.C., Hunt, W.A.: Formally Specifying and Mechanically Verifying Programs for the Motorola Complex Arithmetic Processor DSP. In: Proceedings of the IEEE International Conference on Computer Design (ICCD'97). (1997) 31–36
4. Hoare, C.A.R.: An Overview of Some Formal Methods for Program Design. Computer **20**(9) (1987) 85–91
5. Shukla, S., Bultan, T., Heitmeyer, C.: Panel: Given That Hardware Verification Has Been an Uphill Battle, What Is the Future of Software Verification? In: Proceedings of the Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE '04). (2004) 157–158
6. Berard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petruccci, L., Schnoebelen, P., McKenzie, P.: Systems and Software Verification. Springer-Verlag (2001)
7. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic Verification of Finite State Concurrent System Using Temporal Logic Specifications: a Practical Approach. In: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL'83), ACM Press (1983) 117–126
8. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)
9. Holzmann, G.: Trends in Software Verification. In: Proceedings of the Formal Methods Europe Conference (FME'03). (2003)
10. Holzmann, G., Joshi, R.: Model-Driven Software Verification. In: Proceedings of the 11th Spin Workshop. Volume 2989 of LNCS., Springer-Verlag (2004) 77–92
11. Kapur, D., Subramaniam, M.: Lemma Discovery in Automated Induction. In: CADE-13: Proceedings of the 13th International Conference on Automated Deduction. (1996) 538–552
12. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Progress on the State Explosion Problem in Model Checking. In: Informatics. Volume 2000 of LNCS., Springer-Verlag (2001) 176–194
13. Ball, T., Rajamani, S.K.: Automatically Validating Temporal Safety Properties of Interfaces. In: Model Checking Software: Proceedings of the 8th International SPIN Workshop. (2001) 103–122
14. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software Verification with BLAST. In: Model Checking Software: Proceedings of the 10th International SPIN Workshop. Volume 2648 of LNCS., Springer-Verlag (2003) 235–239
15. Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. Communications of the ACM **12**(10) (October 1969) 576–585
16. Dijkstra, E.W.: Guarded Commands, Nondeterminacy and Formal Derivation of Programs. Communications of the ACM **18**(8) (1975) 453–457
17. McCarthy, J.: Checking Mathematical Proofs by Computer. Proceedings Symposium on Recursive Function Theory
18. Duffy, D.: Principles of Automated Theorem Proving. John Wiley and Sons (1991)
19. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A Theorem Prover for Program Checking. Journal of the ACM **52**(3) (May)
20. Archer, M., Vito, B.D., Munoz, C.: Developing User Strategies in PVS: A Tutorial. In: Proceedings of the First International Workshop on Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003). (2003)
21. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended Static Checking for Java. In: Proceedings of the Internatioanl ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02), ACM Press (2002)
22. Shankar, N.: Combining Theorem Proving and Model Checking through Symbolic Analysis. In: Concurrency Theory (CONCUR'00). Volume 1877 of LNCS., Springer-Verlag (2000) 1–16

23. Arkoudas, K., Khurshid, S., Marinov, D., Rinard, M.: Integrating Model Checking and Theorem Proving for Relational Reasoning. In: Proceedings of the 7th International Seminar on Relational Methods in Computer Science (RelMiCS 2003). Volume 3015 of LNCS. (2003) 21–33
24. Sipser, M.: Introduction to the Theory of Computation. Springer-Verlag (2003)
25. Brock, B., Cooper, S., Pierce, W.: Analogical Reasoning and Proof Discovery. In: Proceedings of the 9th International Conference on Automated Deduction. (1988) 454–468
26. Archer, M., Heitmeyer, C.: Human-Style Theorem Proving using PVS. In: Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs '97). (1997)
27. Thery, L., Bertot, Y., Kahn, G.: Real Theorem Provers Deserve Real User-Interfaces. In: Proceedings of the 5th International ACM SIGSOFT Symposium on Software Development Environments. Volume 17., Software Engineering Notes (1992)
28. Heitmeyer, C.L.: On the Need for Practical Formal Methods. In: Proceedings of the 5th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT '98), Springer-Verlag (1998) 18–26
29. Völker, N.: Thoughts on Requirements and Design of User Interfaces for Proof Assistants. In: Proceedings of the International on User Interfaces for Theorem Provers (UITP '03). Volume 103., Electronic Notes in Computer Science (2003)
30. Kroening, D.: Application Specific Higher Order Logic Theorem Proving. In: Proceedings of the Verification Workshop - VERIFY'02. (July 2002) 5–15
31. Archer, M.: TAME: Using PVS Strategies for Special-Purpose Theorem Proving. Annals of Mathematics and Artificial Intelligence **29** (2000) 139–181
32. Kiniry, J.R., Morkan, A.E., Denby, B.: Soundness and Completeness Warnings in ESC/Java2. In: Proceedings of the Fifth International Workshop on Specification and Verification of Component Based Systems (SAVCBS 2006). (2006)
33. Archer, M., Heitmeyer, C., Sims, S.: TAME: A PVS Interface to Simplify Proofs for Automata Models. In: Proceedings of the International on User Interfaces for Theorem Provers (UITP '98). (1998)
34. Parnas, D.L.: "Formal Methods" Technology Transfer Will Fail. Journal of Systems and Software **40**(3) (1998) 195–198
35. Bowen, J.P., Hinchey, M.G.: Ten Commandments Revisited: A Ten-Year Perspective on the Industrial Application of Formal Methods. In: Proceedings of the 10th international workshop on Formal methods for industrial critical systems (FMICS '05). (September 5–6 2005)
36. Hoare, C.A.R.: How Did Software Get So Reliable Without Proof? In: Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods (FME'96). Volume 1051 of LNCS., London, UK, Springer-Verlag (1996) 1–17