Practical Aspects of Automated Deduction for Program Verification

Wolfgang Ahrendt, Bernhard Beckert, Martin Giese, Philipp Rümmer

Software is vital for modern society. It is used in many safety- or security-critical applications, where a high degree of correctness is desirable. Over the last years, technologies for the formal specification and verification of software – using logic-based specification languages and automated deduction – have matured and can be expected to complement and partly replace traditional software engineering methods in the future. Program verification is an increasingly important application area for automated deduction. The field has outgrown the area of academic case studies, and industry is showing serious interest. This article describes the aspects of automated deduction that are important for program verification in practice, and it gives an overview of the reasoning mechanisms, the methodology, and the architecture of modern program verification systems.

1 Introduction

The field of program verification is a very broad one, and we cannot possibly give an overview of all existing approaches within this article. We focus on approaches of deductive verification, where the actual program code is verified and not an abstracted system (as opposed to, e.g., approaches where many details of the program are abstracted away, and tools like model checking are used on the result). We also focus primarily on systems for modern imperative and object-oriented languages, although some other well-known systems are also mentioned. And finally, we focus more on implementation than on theory, since modern program verification is unthinkable without extensive tool support, not only for proving itself, but also for the generation and management of specifications, proof obligations, etc.

The field of deductive program verification, i.e., formal reasoning about the behaviour of programs, is old. The idea of applying deduction to programs goes back at least to the work of Scott, Plotkin, and Milner in the late 1960s. Recent years have brought tremendous advances in both scope and practicality however. Today, program verification is applied to real-word software. For example, security-critical system software is verified in the Verisoft XT project (see the article in this issue [7]) and the L4.verified project [21].

This article focuses mostly on the aspects that make deductive program verification today different from what it was ten years ago. The KeY system [6, 2], which has been co-developed by the authors, is used as an example in this article.

Program verification is concerned with proving that a program is consistent with some specification, which may be stated in terms of pre-/post-conditions, invariant properties, or termination properties. The deductive approach to show such properties can be roughly divided into two parts.

First, the reasoning about the program logic itself: this include the program's control structures, like conditionals, loops, method calls, exception handling, etc. It also includes the language-specific aspects of data handling, like pointer dereferencing, null-pointer checks, typing issues, pointer aliasing, etc. Additional aspects that are mostly orthogonal to these include, for example, concurrency, garbage collection, data security.

Second, when all has been said about the program itself, one still needs to reason about the data. This means that facts about built-in data types like integers, floats, finite enumerations, strings, etc. need to be proven, but also about user-defined types like, e.g., lists, trees, and dictionaries.

Additionally, some "glue" is needed to bind these two parts together, either explicitly, by embedding both in a common logical language, or implicitly by tying various components of an implementation together.

Approaches vary in how the individual components are realised and bound together, and some aspects of this are discussed in the following section.

One of the most important advances of the past years in the field of deductive verification concerns the treatment of standard data types, which are particularly important for many practical purposes, and for which specialised provers have been developed. This will be discussed in Section 3.

Finally, an essential difference between deductive program verification and, e.g., the mechanical proof of mathematical theorems, is the fact that proofs about programs often fail, due to errors either in the program or in its specification. It makes sense to look at a program verification tool also as a development tool and a debugging aid. This makes it clear that, in practice, feedback about incorrect input is at least as important for a practical system as feedback about correct proofs; see Section 4.

2 System Architecture

As said above, every deductive program verification system has to perform (at least) two rather separate tasks: (a) handling the program-language- and specification-language-specific constructs and reducing or transforming them to classical logic expressions, (b) theory reasoning and reasoning in classical logics, for handling the resulting expressions and statements over data types. One can either handle these tasks in one monolithic logic/system, or one can use a combination of subsystems. The advantage of using subsystems is the possibility to use specialised techniques and tools. The advantage of a monolithic system is that user interaction and providing feedback is easier. In the following we describe exemplary representatives of these two approaches.

Toolchain with Several Subsystems. Tools following what is known as the verifying compiler paradigm, such as Spec# [4], VCC [25], and Caduceus [15], typically use several subsystems. They are all based on powerful fully-automatic provers and decision procedures, and they support real-world programming languages such as C and C#.

For example, the first stage of the VCC toolchain translates the annotated C code into first-order logic via an intermediate language called Boogie [12]. Boogie is a simple imperative language with embedded assertions. Then, in a second stage, a set of first-order logic formulas is generated from the Boogie representation. These formulas state that the program satisfies the embedded assertions. They are called verification conditions, and this second stage a verification condition generator (VCG). More information on VCC and examples can be found in the related article [7] in this issue.

In the second stage, the resulting formulas are given to an automatic theorem prover (TP) resp. SMT solver (in this case Z3 [11, 7]) together with a a background theory capturing the semantics of C's built-in operators, etc. The prover checks whether the verification conditions are entailed by the background theory. Entailment implies that the original program is correct w.r.t. its specification.

For such systems, user input and guidance of proof search solely occurs via the annotations in the source code.

An advantage of this approach is that components in the toolchain can be developed and improved independently: the toolchain will directly benefit from improvements of the underlying theorem prover. If appropriate interfaces are used, it is possible to plug different theorem provers into the toolchain. The developers of the toolchain do not usually need to develop their own theorem prover.

The disadvantage of this approach becomes apparent when proofs fail: the toolchain typically produces very many first-order proof conditions, most of which are easily proved by the theorem prover. But when one of the difficult ones fails, the fact that the proof conditions are so decoupled from the original program means that a lot of expertise is needed to find out why it failed, and what needs to be done to fix the specification or the program. This issue will be addressed further in Section 4.

Monolithic Systems. A typical example for a monolithic verification system is the KeY Program Verification System [6, 2] (co-developed by the authors).

The target language for verification in the KeY system is Java Card 2.2.1. Java 1.4 programs that respect the limitations of Java Card (no floats, no concurrency, no dynamic class loading) can be verified as well. Specifications are written using the Java Modeling Language (JML).

The program logic of KeY, called Java Card DL, is axiomatised in a *sequent calculus*. Those calculus rules that axiomatise program formulas define a symbolic execution engine for Java Card and so directly reflect the operational semantics. The calculus is written in a small domain-specific language called the *taclet* language that was designed for concise description of rules. Taclets specify not merely the logical content of a rule, but also the context and pragmatics of its application. They can be efficiently compiled not only into the rule engine, but also into the automation heuristics and into the GUI. Depending on the configuration, the axiomatisation of Java Card in the KeY prover uses 1000–1300 taclets.

The KeY system is not merely a verification condition generator (VCG), but a theorem prover for program logic that combines a variety of automated reasoning techniques. The KeY prover is monolithic in that symbolic execution of programs, first-order reasoning, arithmetic simplification, external decision procedures, and symbolic state simplification are interleaved.

At the core of the KeY system is the deductive verification component, which also can be used as a stand-alone prover. It employs a free-variable sequent calculus for first-order dynamic logic for Java. The calculus is proof-confluent, i.e., no backtracking is necessary during proof search.

While striving for a high degree of automation, the KeY prover features a user interface for presentation of proof states and rule application, aiming at a seamless integration of automated and interactive proving.

Another monolithic tool is the ACL2 system [20]. It achieves integration of program reasoning into the logic by using the same language (a subset of LISP) for programming, specification, and during proving. The same can be said of some approaches to verification that build on systems based on higher-order logics or type-theory (e.g. [17, 28, 8]): the logic is interpreted as a higher-order functional programming language. In order to make this approach sound, one needs to ensure that the "programs" actually behave like terms in the logic. In particular, special attention needs to be paid to termination. In ACL2, for instance, the system usually accepts a function definition only after it is shown that it terminates on all inputs. One advantage of this approach is that the user needs to learn only one formalism for specification, programming, and proving. One also has the advantage that proof methods developed for the logic can be directly applied to proofs about programs. A drawback of this approach is that it requires additional work for aspects of programs that are not desirable, or not naturally present, in a logic, like non-termination, side effects, or resource usage. To reason about these aspects, one needs to depart from the strict equation of terms and programs. These problems are surmountable however, and ACL2 and various of the higher-order systems have successfully been used for substantial verification tasks including all of the mentioned aspects.

The Need for Interaction. User interaction remains indispensable in deductive program verification. Efficiency must, thus, be measured in terms of user plus prover, not just prover alone. Some systems, including KeY and many of the higher-order and type theory tools, emphasise interaction during proof search, be it through a graphical user interface or through a command line. Others, like ACL2 and most toolchain systems require up-front interaction, either via program annotations, or by formulating a suitable set of lemmas to guide the proof construction. In both cases, the user has to analyse the situation when proof search fails – either during proof construction or after a failed proof attempt –, and then has to guide the proof construction or provide helpful lemmas, axiom instantiations etc. In monolithic systems that is done by invoking certain calculus rules. For toolchain systems, new annotations have to be added. This process has to be repeated until a proof is found.

Inductive invariants and *variants* (or, more generally, *well-founded orders*) are two kinds of auxiliary specifications that are

difficult to synthesise for verification systems and that traditionally have to be provided by the user. Invariants are needed for the verification of programs containing loops and the specification of data structures. Variants are used to verify the termination of loops or recursive programs by ordering the state space of a program in a well-founded manner that is consistent with the program execution.

The last years have seen significant advances in methods for the automatic inference of both invariants and variants, so that user interaction can today be avoided in many practical cases. Variants can be generated by algebraic techniques like linear programming or finite differences (e.g., [22, 9]). Besides, a variety of methods to prove termination has been developed in the context of term rewriting systems (e.g., [3, 14]) and has successfully been applied in particular to functional or logic programs. Invariants can be generated by algebraic methods, proof analysis, and many other techniques, see e.g. [19].

3 Reasoning about Theories and Data Structures

Both in monolithic systems and in systems built as toolchains, the logically last stage in the verification of a program is to check and (possibly) discharge elementary verification conditions. On this level, most constructs specific to a programming language have been eliminated, and what remains are (a) basic datatypes provided by a language, like integer and floating point arithmetic, algebraic datatypes, or strings, (b) theories that have been used in the specification of programs or libraries, which could be sets, lists, maps, or arrays, and (c) theories that have been used to encode the semantics of the program to be verified, which are primarily first-order constructs like uninterpreted functions, quantifiers, and the theory of arrays. Reasoning about formalisms like these has seen enormous advances in the last years, and there are two main approaches that dominate today's verification landscape.

Satisfiability Modulo Theories (SMT). The automatic theorem provers currently used in program verification, e.g., Z3 [11], CVC3 [5], and Yices [13], follow the "little proof engine" paradigm [26] and have their roots in the area of satisfiability checkers for propositional logic (SAT solvers). This architecture is in contrast to the design of classical automated theorem provers for first-order logic, although there has been a rich exchange of concepts and ideas between the two approaches.

SMT solvers are able to reason very efficiently about first-order formulas that have a complicated propositional structure, which is an important feature for program verification (where the number of cases to be covered often is huge, due to the size of programs and the number of execution paths through them). SMT solvers also represent a framework in which decision procedures for theories like uninterpreted functions, linear integer, rational, and bit-vector arithmetic, or arrays can naturally be integrated. These theories are sufficient for many practical program verification problems, e.g., proving that array bounds are not violated during program execution.

While SMT solvers are decision procedures for ground formulas in first-order logic with various theories (which are usually difficult for classical first-order theorem provers), they typically only offer heuristic support for quantifiers. This situation is to a certain degree inevitable, because many theories become too expressive to allow even semi-decision procedures when combined with quantifiers (for instance, linear integer arithmetic combined with uninterpreted functions and quantifiers is Π^1_1-hard). Most SMT solvers are not even complete for first-order logic, however. This restriction becomes relevant for more complicated and harder verification tasks, including proofs of the functional correctness of programs (e.g., that a sorting procedure actually returns a sorted permutation of the input array). Although the integration of SMT solvers with first-order theorem provers is an active area of research, until now such tasks are the domain of interactive systems.

Decision Procedures in Interactive Systems. Monolithic verification systems and proof assistants typically offer decision procedures for theories, too, although the setup is different than in SMT solvers. Monolithic systems are normally built on expressive logics, in which theories can be introduced by adding new symbols and axioms. In order to support the proving process, the applications of the theory axioms is automated by means of *tactics*, *proof search strategies*, or *reflected algorithms*, which are all essentially programs defined in the verification system that implement simplification or decision procedures. Such procedures can be interactively invoked by the user to generate (part of) a proof. In the KeY system, for instance, proof search strategies exist for deciding formulas in linear integer arithmetic, and to simplify expressions in nonlinear arithmetic [23].

The range of theories accessible for interactive systems is larger than that of SMT solvers, because the more expressive a theory is, the more difficult is it to achieve full automation. To reason about full Peano arithmetic or other inductively defined datatypes, for instance, it is frequently necessary to instantiate induction axioms, which is normally beyond the abilities of automatic procedures. Also for the treatment of quantifiers, user interaction is often unavoidable. Simpler reasoning steps in such theories can nevertheless be performed automatically.

Statements about program correctness for algorithms often correspond to interesting mathematical properties of the data types. Much of the deduction required in these cases is very similar to mathematical theorem proving. A possible difference is that much of mathematics concentrates on algebraic structures like rings and fields, which mostly require inductive proofs only to establish that a given type actually is an instance of such a structure. In program verification, where the specific properties of lists, trees, dictionaries, or user-defined datatypes are often central, more proofs make direct use of induction.

Since the same kind of general statement about common data types are often required in different programs, it is useful to build a library of lemmas, simplification rules, etc. to help in automating reasoning about the data types. Every reasonable verification system has some mechanism for this kind of "theory reasoning."

In practical systems, the two approaches to theory reasoning are often combined: simpler (but possibly large) parts of verification conditions can be sent to back-ends that will try to construct a proof automatically, while deeper and more intricate (but hopefully smaller) parts can be handled with a higher degree of interaction. The KeY system, for instance, supports

various SMT solvers like CVC3 and Yices as back-ends, but also offers full interactive reasoning and semi-automatic proof search strategies.

4 Automated Deduction for Bug Finding

Traditionally, approaches and systems for deductive program verification fall short of giving useful feedback in the case that the program under consideration is *not* correct. But, indeed, this is the much more frequent case, in particular when it comes to applications that exceed the size of text book examples. In the last decade or so, this challenge is increasingly acknowledged in the research community, and various approaches address this issue. To categorise those, we start with the observation that deductive program verification usually phrases the problem of program correctness as formulas, the *validity* of which implies the program being *correct*. Now, there are several ways to show programs to be *incorrect*. One is to establish the *invalidity* of formulas, by generating *counter models*. This approach is also referred to as *disproving*. An alternative way is to reduce *incorrectness* to *validity*, by coding it as a formula again, and applying theorem proving directly. (A third possibility is that the (sub-)problem at hand is decidable, in which case decision procedures can establish validity or invalidity. This case is not discussed in the following but see the previous section for a discussion of decision procedures.)

Disproving Correctness. We first take a closer look at the problem of showing invalidity of formulas. Automated deduction is traditionally tailored to showing *validity* of conjectures, while the issue of showing *invalidity* is largely neglected (at least in undecidable logics). The reasons for this are both of theoretical and practical kind. On the theoretical side, the problem of invalidity is harder than that of validity in many logics. For instance, in first-order logic, validity is semi-decidable, whereas invalidity is not. On the practical side, invalidity was not considered interesting in the early decades of automated deduction, where the application area was mostly mathematics. The conjectures to be proved automatically were largely known to be true, or at least likely to be true. This changes dramatically when deduction is used for verifying systems, be it software or hardware. In particular software is almost never correct from the start. For automated deduction to be effective for program verification, it needs to give the user helpful feedback on faulty conjectures.

The problem of showing a conjecture φ invalid can be expressed as the problem of finding a *counter model* for φ, i.e., a model in which all assumptions (axioms) are true but φ is not. In general, finding counter models is a very hard problem, as their existence is undecidable in many logics. Note that, for instance, a single boolean function over the natural numbers has already uncountably many models. The solutions suggested in the literature focus on some *finite* approximation of the general problem. One way is to consider finite approximations of possibly infinite domains, and search for models on increasing domain sizes, see for instance [10]. A different approach is to leave the domains infinite, but put a finite limit on the instantiations of universal quantifiers in the axioms [1]. Neither of these approaches can be both sound and complete. The second, however, is at least monotonic, in the sense that the number of models only gets smaller when the limit is increased.

Proving Incorrectness. In cases where incorrectness of a program can be expressed as a formula (in a logic where we have a proof system at hand), we can use proving machinery instead of constructing counter models. However, it is not as easy as that, normally. The coding of incorrectness in formulas might not be straightforward. Also, such formulas can carry characteristics which are not well supported by standard proving technology; this calls for research on theorem proving tuned for these purposes. In the following, we consider two approaches demonstrating these issues.

At first, we look at the problem of proving incorrectness of pre-post condition statements, of the kind expressed by Hoare triples [18]. As the triple $\{\phi\}\pi\{\psi\}$ demands ψ to hold after all terminating runs of π starting form a state satisfying ϕ, the incorrectness of this triple corresponds to the *existence* of a state where ϕ is true, such that execution of π *terminates*, and ψ is *not* valid afterwards. It is shown in [24] how to formulate and prove this kind of incorrectness in dynamic logic, an extension of Hoare logic able to express termination, and allowing programs in the scope of quantifiers. The main challenge for the deduction machinery is the quite massive existential quantifier over the pre-state. The target language of that work is Java, such that the state entails the object heap, and the quantifier instantiation mechanism must be able to construct such heaps. This is done using a tableau/sequent style calculus featuring *meta-variables* (also called 'free') to be instantiated by incremental *constraint solving*, extending on the work in [16].

Besides partial correctness, another aspect of program correctness is termination. In that setting, proving incorrectness corresponds to proving non-termination. For imperative languages the possible non-termination of loops $\texttt{while}(\epsilon)\{\alpha\}$ is of particular relevance. The approach taken in [27] essentially searches for a loop invariant I such that $I \rightarrow \epsilon$, which shows that the termination condition cannot be reached. The synthesis of such invariants is again based on constraint solving, and on iteratively strengthening invariants based on counter examples from failed proof attempts.

5 Conclusion

Program verification is today one of the most promising application areas of automated deduction. In line with that, the automated deduction community has in recent years put an increasing emphasis on the characteristics of deduction problems in program verification. In this article, we tried to give an overview of those aspects of deduction which are particularly significant in this context, and reviewed the typical scenarios for the interplay of deduction and verification. We categorised typical architectures (toolchain, monolithic) and the various kinds of user input (assertions, lemmas, interactive proof steps). We also discussed the dominant role of theories and data structure in the reasoning, being addressed (among others) by tailored proof strategies and decision procedures (including SMT). Again, those are combined either by integrating *tools* (toolchain) or by integrating *techniques* into the frame of a more general calculus (mono-

lithic). Finally, we highlighted the role of feedback in the case that the verification target is *not* correct

Altogether, one can say that the rapid progress of practical program verification in the last decade or so has boosted the field of automated deduction, and, we are convinced, will continue to do so.

References

[1] Wolfgang Ahrendt. Deductive search for errors in free data type specifications using model generation. In Andrei Voronkov, editor, *Proceedings, 18th International Conference on Automated Deduction (CADE-18), Copenhagen, Denmark*, LNCS 2392. Springer, 2002.

[2] Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, Philipp Rümmer, and Peter H. Schmitt. Verifying object-oriented programs with KeY: A tutorial. In F. de Boer, M. Bonsangue, S. Graf, and W. de Roever, editors, *Revised Lectures, 5th International Symposium on Formal Methods for Components and Objects (FMCO 2006), Amsterdam, The Netherlands*, LNCS 4709. Springer, 2007.

[3] Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.*, 236(1-2):133–178, 2000.

[4] Mike Barnett, Rustan Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS), International Workshop, 2004, Marseille, France, Revised Selected Papers*, LNCS 3362, pages 49–69. Springer, January 2005.

[5] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings, 19th International Conference on Computer Aided Verification (CAV '07)*, LNCS 4590, pages 298–302. Springer, 2007.

[6] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[7] Bernhard Beckert and Michał Moskal. Deductive verification of system software in the Verisoft XT project. *KI*, 2010. *In this issue*.

[8] Yves Bertot. A short presentation of Coq. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *Proceedings, 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs), Montreal, Canada*, LNCS 5170, pages 12–16. Springer, 2008.

[9] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination of polynomial programs. In Radhia Cousot, editor, *VMCAI*, volume 3385 of *LNCS*, pages 113–129. Springer, 2005.

[10] Koen Claessen and Niklas Sörensson. New techniques that improve MACE-style model finding. In *Proc. of Workshop on Model Computation (MODEL)*, 2003.

[11] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, Proceedings of the 14th International Conference, Budapest, Hungary*, LNCS 4963, pages 337–340. Springer, 2008.

[12] Rob DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[13] Bruno Dutertre. System description: Yices 1.0.10. In *SMT-COMP'07*, 2007.

[14] Stephan Falke and Deepak Kapur. A term rewriting approach to the automated termination analysis of imperative programs. In Renate A. Schmidt, editor, *CADE*, volume 5663 of *LNCS*, pages 277–293. Springer, 2009.

[15] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In *Proceedings, Formal Methods and Software Engineering*, LNCS 3308, pages 15–29. Springer, 2004.

[16] Martin Giese. Incremental Closure of Free Variable Tableaux. In *Proc. Intl. Joint Conf. on Automated Reasoning, Siena, Italy*, number 2083 in LNCS, pages 545–560. Springer, 2001.

[17] John Harrison. HOL Light: A tutorial introduction. In Mandayam K. Srivas and Albert John Camilleri, editors, *Proceedings, First International Conference on Formal Methods in Computer-Aided Design (FMCAD), Palo Alto, USA*, LNCS 1166, pages 265–269. Springer, 1996.

[18] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.

[19] Andrew Ireland and Laura Kovács, editors. *WING 2009, Workshop on Invariant Generation*, 2009.

[20] Matt Kaufmann, Panagiotis Manolios, and J. Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, Boston, MA., 2000.

[21] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles, Big Sky, MT, USA*. ACM, October 2009.

[22] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, number 2937 in LNCS, pages 239–251. Springer, 2004.

[23] Philipp Rümmer. A sequent calculus for integer arithmetic with counterexample generation. In *Proceedings, 4th International Verification Workshop (VERIFY'07)*, volume 259 of *CEUR (http://ceur-ws.org/)*, 2007.

[24] Philipp Rümmer and Muhammad Ali Shah. Proving programs incorrect using a sequent calculus for Java Dynamic Logic. In Yuri Gurevich and Bertrand Meyer, editors, *Tests and Proofs, First International Conference, TAP 2007, Zurich, Switzerland, February 12-13, 2007. Revised Papers*, LNCS 4454, pages 41–60. Springer, 2007.

[25] Wolfram Schulte, Xia Songtao, Jan Smans, and Frank Piessens. A glimpse of a verifying C compiler. In *Proceedings, C/C++ Verification Workshop*, 2007.

[26] Natarajan Shankar. Little engines of proof. In *Proceedings, International Symposium of Formal Methods Europe, Copenhagen, Denmark*, LNCS 2391, pages 1–20. Springer, 2002.

[27] Helga Velroyen and Philipp Rümmer. Non-termination checking for imperative programs. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy*, volume 4966 of *LNCS*, pages 154–170. Springer, 2008.

[28] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle framework. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *Proceedings, 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs), Montreal, Canada*, volume 5170 of *LNCS*, pages 33–38. Springer, 2008.

Contact

Dr. Wolfgang Ahrendt
Department of Computer Science and Engineering
Chalmers University of Technology
Rännvägen 6B, SE-41296 Göteborg, Sweden
Phone: +46 31 772-1011
Email: ahrendt@chalmers.se

Prof. Dr. Bernhard Beckert
Fachbereich Informatik
Universität Koblenz
Universitätsstraße 1, D-56070 Koblenz, Germany
Phone: +49 261 287-2775
Email: beckert@uni-koblenz.de

Dr. Martin Giese
Department of Informatics
University of Oslo
Postboks 1080 Blindern, N-0316 Oslo, Norway
Phone: +47 22 852737
Email: martingi@ifi.uio.no

Dr. Philipp Rümmer
Computing Laboratory
Oxford University
Wolfson Building
Parks Road, Oxford OX1 3QD, UK
Email: philr@comlab.ox.ac.uk

Bild

Wolfgang Ahrendt is a senior lecturer at Chalmers University, Gothenburg, Sweden. His research interests include formal methods in software engineering and automated deduction with a special emphasis on model generation and disproving.

Bild

Bernhard Beckert is a professor of Formal Methods and Artificial Intelligence at the University of Koblenz. His research interests include automated deduction, non-classical logics, and formal methods in software engineering.

Bild

Martin Giese is a post-doc researcher at the University of Oslo. His research interests include formal methods in software engineering, tableau-based automated deduction, equality reasoning, the integration of interactive and automated theorem proving, the implementation of proof systems, reflection, probabilistic modelling, and semantic technologies.

Bild

Philipp Rümmer is a research assistant at Oxford University. His research interests include formal methods in software engineering, verification of imperative, concurrent, and embedding software, termination checking, theorem proving, decision procedures for arithmetic, and disproving.