Intelligent Systems and Formal Methods in Software Engineering

Bernhard Beckert

Software is vital for modern society. It manages our finances, regulates communication and power generation, controls airplanes, and processes security-critical information. Consequently, the efficient development of dependable software is of ever-growing importance. Over the last few years, technologies for the formal description, construction, analysis, and validation of software—based mostly on logics and formal reasoning—have matured. We can expect them to complement and partly replace traditional software engineering methods in the future.

Formal methods in software engineering are an increasingly important application area for intelligent systems. The field has outgrown the area of academic case studies, and industry is showing serious interest. This installment of Trends & Controversies looks at the state of the art in formal methods and discusses the developments that make successful applications possible. Tony Hoare, a pioneer in the field, convincingly argues that we've reached the point where we can solve the problem of how to formally verify industrial-scale software. He has proposed program verification as a computer science Grand Challenge. Deductive software verification is a core technology of formal methods. Reiner Hähnle describes recent dramatic changes in the way it's perceived and used. Another important base technique of formal methods, besides software verification, is synthesizing software that's correct by construction because it's formally derived from its specification. Douglas R. Smith and Cordell Green discuss recent developments and trends in this area. Surprisingly efficient decision procedures for the satisfiability modulo theories problem have recently emerged. Silvio Ranise and Cesare Tinelli explain these techniques and why they're important for all formal-methods tools. Thomas Ball and Sriram Rajamani look at formal methods from an industry perspective. They explain the success of Microsoft Research's SLAM project, which has developed a verification tool for device drivers.

–Bernhard Beckert

Verified Software: Proposal for a Grand Challenge Project

Tony Hoare, *Microsoft Research*

Verified software consists of programs that have been proved free of certain rigorously specified kinds of error. Confidence in the correctness of the proof is very high because it has been generated and checked automatically by computer. Our understanding of the concept of a proof goes back to the Greek philosophers Pythagoras and Aristotle. The German philosopher Leibniz proposed mechanical reasoning for mathematical theorems. James King explored the idea of a mechanical program verifier in his 1969 doctoral thesis.[1] Since then, the ideal of verified software has inspired a generation of research into program correctness, programming language semantics, and mechanical theorem proving. The results have moved into industrial applications to detect errors in software simulations of hardware devices.

A new Grand Challenge?

A Grand Challenge is a scientific project that an international team of scientists coordinates and carries out over 10 years or more. A recent example, the Human Genome Project (1991–2004), was inspired by an altruistic scientific ideal to uncover the six billion bits of the human genome—effectively, the blueprint from which nature manufactures each of us. It was expected that on successful completion of the project (15 years ahead), the pharmaceutical industry could exploit its results and experimental techniques and tools to benefit human health. This is now beginning to happen.

Many scientists took the turn of the millennium as an opportunity to ask themselves and their colleagues whether the time was ripe for a Grand Challenge project of similar scale and ambition in their own branch of science. I propose that computer scientists reactivate the program verifier challenge and solve it by concerted collaborative and competitive efforts within the foreseeable future. Leading computer scientists have discussed this proposal at conferences in Europe, North America, and Asia over the last two years.

Vision

The scientists planning this project envision a world in which computer programmers seldom make mistakes and never deliver them to their customers. At present, programming is the only engineering profession that spends half its time and effort on detecting and correcting mistakes made in the other half of its time.

We envision a world in which computer programs are always the most reliable component of any system or device that contains them. At present, devices must often be rebooted to clear a software error that has stopped them from working.

1541-1672/06/$20.00 © 2006 IEEE
Published by the IEEE Computer Society

Cost-benefit

Current estimates of the cost of software error to the world's economies are in the region of US$100 billion a year and increasing. Software producers and users share this cost. The total cost of our proposed research over a period of 10 years is $1 billion, shared among all participating nations.

If the research is successful, the cost of deploying its results will be many times larger than the research cost. Those who profit from the deployment will meet these costs.

Security

Verification methods already exist that detect some errors that make software vulnerable to virus attack. However, the technology is also available to those who plan such attacks. We must develop the technology not only to detect but also to eliminate such errors.

The risks

The main scientific risk to the project is that methods of computer generation and checking of proofs continue to require skilled human intervention. The tools will therefore remain as scientific tools, suitable to advance research but inadequate for engineers' needs.

Another risk is that the computing research community has little experience with long-term research, requiring collaboration among specialized research areas. The project's success will require adjustment of the current research culture in computer science and alignment of research-funding mechanisms to new scientific opportunities, methods, and ideals.

The main impediment to deploying the research results will be educating and training programmers to exploit the tools. The primary motivator for deployment will be commercial competition among software suppliers.

The method

The project will exploit normal scientific research methods. Advances in knowledge and understanding will be accumulated in comprehensive mathematical theories of correct programming. These will be tested by extensive experimentation on real working programs from a broad range of applications. The research results will be incorporated in sets of tools that will enable the practicing programmer to exploit them reliably and conveniently.

Scientific ideals

As with the Human Genome Project, the driving force of the project is scientific curiosity to answer basic questions underlying the whole discipline. For software engineering, the basic questions are: What do programs do? How do they work? Why do they work? What evidence exists for the correctness of the answers? And finally, how can we exploit the answers to improve the quality of delivered software?

The ideal of program correctness is like other scientific ideals—for example, accuracy of measurement in physics, purity of materials in chemistry, rigor of proofs in mathematics. Although these absolute ideals will never in practice be achieved, scientists will pursue them far beyond any immediate market need for their realization. Experience shows that eventually some ingenious inventor or entrepreneur will find an opportunity to exploit such scientific advances for commercial benefit.

> Programming is the only engineering profession that spends half its time and effort on detecting and correcting mistakes made in the other half of its time.

Commitment to such ideals contributes to the cohesion of integrity of scientific research. It's essential to the organization of a large-scale, lengthy project such as a Grand Challenge. A similar commitment by the engineer contributes to the trust and esteem that society accords to the profession.

Industrial participation

The programming tools developed in this project must be tested against a broad range of programs representative of those in use today. We propose that our industrial partners contribute realistic challenge programs of various sizes and degrees of difficulty. These must be programs with no competitive value so that they can be released for scientific scrutiny. Industry might also contribute valuable prizes to the teams making the greatest progress on these and other challenge problems.

The tools will be continuously improved in the light of experience of their use. Throughout the project, students will be trained to apply the tools to commercially relevant programs and will obtain employment that exploits their skills on competitive proprietary products.

The scientists engaged in the project will give any required support and advice for the development of commercial toolsets that will spread the project's results throughout the programming profession.

The state of the art

Advanced engineering industries—including transport, aerospace, electronics hardware, communications, defense, and national security—already exploit verification technology in small, safety-critical cases. Specialized tools in these areas are available from numerous start-up companies, together with consultation and collaboration in their use. Other tools are available for rapid detection of as many errors as possible.

In scientific research laboratories, guarantees based on complete verification have been given for small microprocessors, operating systems, programming language compilers, communication protocols, large mathematical proofs, and even the essential kernels of the proof tools themselves.

In the last 10 years, some of the software carrying out the basic task of constructing proofs has improved in performance by a factor of 1,000. This is in addition to the factor of 100 achieved by improvements in the hardware's speed, capacity, and affordability. Both the software and hardware continue to improve, giving a fair chance of substantial program verification over the next 10 years.

The start of the project

Many research centers throughout the world are already engaged in the early stages of this project. The next step is to persuade the national funding agencies of the countries most capable of contributing that this Grand Challenge potentially offers a good return on investment.

Reference

1. J.C. King, "A Program Verifier," PhD thesis, Carnegie Inst. of Technology, 1969.

Deductive Software Verification

Reiner Hähnle, *University of Koblenz*

Deductive software verification is a formal technique for reasoning about properties of programs that has been around for nearly 40 years (see the previous essay for more details about verification's history). However, numerous developments during the last decade have dramatically changed how we perceive and use deductive verification:

- The era of verification of individual algorithms written in academic languages is over. Contemporary verification tools support commercial programming languages such as Java or C#, and they're ready to deal with industrial applications.
- Deductive verification tools used to be stand-alone applications that could be used effectively only after years of academic training. Now we see a new generation of tools that require only minimal training to use and are integrated into modern development environments.
- Perhaps the most striking trend is that deductive verification is emerging as a base technology. It's employed not only in formal software verification but also in automatic test generation, in bug finding, and within the proof-carrying code framework—with more applications on the horizon.

What is deductive software verification?

Three ingredients characterize deductive software verification: First, it represents target programs as well as the properties to be verified as logical formulas that must be proven to be valid. Second, it proves validity by deduction in a logic calculus. Third, it uses computer assistance for proof search and bookkeeping.

In contrast to static analysis and model checking, you can model the semantics of the target programming language precisely—that is, without abstracting from unbounded data structures (integers, lists, trees, and so on) or unbounded programming constructs (such as loops or recursive method calls). The logics used for deductive verification are at least as expressive as first-order logic with induction. So, you can formalize and prove far-reaching properties of target programs.

This precision has a price, of course: even more so than in model checking, predicting the amount of computing resources necessary to achieve a verification task is difficult. In general, computability theory implies theoretical limitations that exclude a verification system that can prove program properties over infinite structures.

In light of these results, in the past, designers of theorem provers often traded off automation for precision by relying on human interaction. A recent trend is to insist on automation but to accept occasionally approximate results. In contrast to static analysis and model checking, which use abstraction, in deductive verification you give up on completeness by shortcutting proof search. In this case, you can't obtain functional correctness proofs, but automatic generation of unit tests and warnings about potential bugs are still useful.

> A new generation of deductive verification tools require only minimal training to use and are integrated into modern development environments.

Embedding versus encoding

Two approaches to logical modeling of programming languages and their semantics exist. In the first, target programs appear as a separate syntactical category embedded in logical expressions. The best-known formalism of this kind is Hoare logic. More recent ones include Dijkstra's weakest precondition calculus and dynamic logic. Logical rules that characterize the derivability of those formulas that contain programs reflect the target programming language's operational semantics. Rule application is syntax driven, and at least one rule exists for each target-language construct. You can view proofs in such calculi as *symbolic execution* of the program under verification. You can compute either a program's strongest postcondition with respect to a given start state or, vice versa, the weakest precondition with respect to a given final state. In either case, symbolic execution results in a set of program-free formulas, the *verification conditions*. A first-order theorem prover can then automatically carry out a comparison of the strongest postcondition (weakest precondition) to the final state (start state) in the specification using a first-order theorem prover.

Representative state-of-the-art systems (see table 1) employing the embedding approach include Spec# and KeY. Spec#, although its authors refer to it as a "static program verifier," is an advanced verification condition generator with a theorem prover back end for C# programs annotated with specifications written in the Spec# language. KeY supports the full Java Card 2.2 standard (http://java.sun.com/products/javacard). It's not merely a verification condition generator; it interleaves symbolic execution and automated theorem proving to increase efficiency.

The second modeling approach encodes both the target language syntax and semantics as theories, often in higher-order logic. This involves formalizing many auxiliary data structures such as sets, functions, lists, tuples, and records. To verify a target program—that is, to prove a theorem about it—you need a large library of lemmas for the auxiliary theories used in the formalization. Programs are typically encoded as mutually recursive function definitions, which can also be seen as an abstract functional programming language. Needless to say, fully formalizing an imperative programming language such as Java in this way is a substantial undertaking.

Arguably the most powerful verification system along these lines is ACL2, which its authors have been improving since they created it more than 30 years ago. Another well-known system is HOL, a general interactive theorem-proving system based on typed higher-order logic. The generic theorem prover Isabelle is increasingly popular—it even lets you define the logic it uses as a basis of formalization. HOL and Isabelle weren't designed to be software verification tools; they're logical and reasoning frameworks that can be just as well applied (and in fact have been applied) to pure mathematics.

A changing perspective

Both approaches to logic modeling of programs—embedding and encoding—have specific strengths and weaknesses. These lead to distinct usage scenarios. The embedding approach is giving rise to

Table 1. A selection of verification tools.

Name	Creator	Availability	Modeling approach	Target language	Remarks
ACL2	University of Texas	Public	Encoding	ACL2	
HOL (Higher-order logic)	University of Cambridge	Public	Encoding	Various	Programming environment
Isabelle	University of Cambridge, Technical University Munich	Public	Encoding	Various	
JACK (Java Applet Correctness Kit)	INRIA	Need to register	Embedding	Java	
JIVE (Java Interactive software Visualization Environment)	Swiss Federal Institute of Technology (ETH) Zürich, University of Kaiserslautern	No	Embedding	Java	
KeY	Chalmers University, University of Karlsruhe, University of Koblenz	Public	Embedding	Java Card	Eclipse and Borland Together plug-ins, test generation
KIV (Karlsruhe Interactive Verifier)	University of Augsburg	Public (except the Java version)	Hybrid	ASM Pascal-like Java	Multiple specification methods
Spec#	Microsoft Research	Public	Embedding	C#	Visual Studio plug-in, bug finding

a new generation of program analysis tools.

Despite ACL2's impressive performance and detailed documentation,[1] using it effectively requires a long apprenticeship: the formalization style is tightly interwoven with the control of heuristics that are essential for automation. Consequently, detailed tool knowledge is required for effective use, limiting the number of potential users.

General-purpose verification tools based on encoding programs into logical frameworks require you to master their theoretical underpinnings before you can use them. Large, imperative programming languages don't match these tools' functional flavor well, and formalizing the target language semantics is a considerable undertaking. In conjunction with their generality, this makes them predestined for theoretical investigations (for example, precise semantical modeling of programming languages) rather than routine verification tasks. On the other hand, these tools' expressivity lets you formally verify the correctness of compilers or of the calculi used in the systems based on embedding that have a simpler theory. This latter capability of cross-validating the target language's formal semantics in different verification tools is essential to ensure trust in the verification process. In the light of certification, this is increasingly important.

Research groups within the theorem-proving community rooted in logic and proof theory (where formal foundations and theoretical completeness are major virtues) often developed tools based on encoding of target programs. The recent generation of tools based on embedding, however, is heavily influenced by the programming languages and software engineering communities' needs:

- User interfaces shouldn't be more complex than those of debuggers or compilers.
- Tools must be integrated into development environments and processes.
- Bug finding and testing are as important as verification.

On the other hand, approximation via incompleteness is quite acceptable.

For verification tools to be usable, they must cover to a considerable extent modern industrial programming languages such as Java and C# as verification targets. Equally important is the possibility of writing specifications in languages close to those developers use. For example, KeY supports OCL (the Object Constraint Language, which is part of the Unified Modeling Language; see www.uml.org) and JML (the Java Modeling Language; see www.cs.iastate.edu/~leavens/JML). The latter is compatible with Java expression syntax and has become the de facto standard for verification tools targeting Java. JML is also supported by JACK (Java Applet Correctness Kit) and JIVE (Java Interactive software Visualization Environment), among other tools. Spec#, named identically to the tool set, is a JML analog for C#. In all cases, high-level specifications are compiled automatically to logic-based representations, so that users don't need to write such low-level specifications.

Trends and challenges

One major trend is that verification tools attempt to combine formal verification, automatic test generation, and automatic bug finding into a uniform framework that, in turn, is integrated into standard software development platforms such as Eclipse (www.eclipse.org). The driving force behind this trend is the insight that mere formal verification is too limited in many cases: it depends on the existence of relatively extensive formal specifications and, as I explained earlier, often isn't fully automatic. In addition, formal verification target languages are at the source-code or, at most, bytecode level. But there must also be ways to validate binaries deployed on, for example, smart cards. We should also remember that industrial users often fail to see the point of proving correctness, but they always see the need for debugging and testing.

A uniform view of verification, test generation, and bug finding is well justified: the Extended Static Checking project (http://secure.ucd.ie/products/opensource/ESCJava2) pioneered verification technology for bug finding. A theoretical basis also exists for recasting the search for bugs as verification attempts of invalid claims.[2] Likewise, automated white- and black-box test generation can be embedded into a verification framework.[3]

A renewed interest in verification itself is motivating the development of better integrated, more automatic verification tools. This is partly driven by enhanced certification requirements for safety-critical software, but also by technological and economical considerations. Mobile software hosted on smart cards and cell phones shares important deployment characteristics with hardware: recalling and updating it is difficult and costly. In addition, typical mobile software applications bear a high economic risk. Most security policies can't be enforced statically or by language-based mechanisms alone, but they can be seen as particular instances of verification problems and, therefore, can be tackled with deductive verification tools. The EU Integrated Project MOBIUS (Mobility, Ubiquity, and Security, http://mobius.inria.fr) combines type-based program analysis and deductive software verification in a proof-carrying code architecture to establish security policies offline in Java-enabled mobile devices.

Exciting new applications of deductive verification technology are emerging in dependable computing—for example, symbolic fault injection, symbolic fault coverage analysis, or deductive cause-consequence analysis.

Deductive verification is quickly moving forward, but formidable challenges lie ahead. Several important program features can't yet be handled adequately—notably concurrent threads, support for modularity, and floating-point numbers.

The lack of available documentation outside of technical research articles threatens to hamper progress. Currently, only one book attempts to document the theory and application of a newer-generation verification tool.[4] We need more.

Verification tools' reach can be extended with powerful automated theorem provers that deal with verification conditions and intermittent simplification of proof goals. Numerous theorem provers optimized for verification have appeared recently. The needs of abstract interpretation and software model checking have mostly driven their development, so these systems aren't yet fully optimized for deductive verification. Nevertheless, early experiments have been promising. Most deductive-verification tools offer a theorem prover back-end interface in the recently adopted SMT-LIB (Satisfiability Modulo Theories Library) format (www.smt-lib.org).

Although tool and method integration have progressed substantially, we don't yet fully understand how best to integrate verification into software development processes or into large-scale enterprise software frameworks.

In business and automotive applications, using software that has been automatically generated from more abstract modeling languages such as UML or Simulink is becoming popular. This poses a challenge to verification because modeling languages lack conventional programming languages' maturity and precise semantics. Besides, few verification tools can handle them. Verifying the generated source code isn't possible either because it's too unstructured.

The available specification languages for expressing formal software requirements aren't fully satisfactory. General-purpose formalisms such as Z or RSL tend to be too cumbersome to use. OCL is too abstract, while JML and Spec# are in flux and are tied to a particular target programming language. Specification is, of course, a problem not only of deductive verification but also of any verification technique.

Deductive verification is quickly moving forward, but formidable challenges lie ahead. Several important features can't yet be handled adequately.

References

1. M. Kaufmann, P. Manolios, and J Strother Moore, *Computer-Aided Reasoning: An Approach*, Kluwer Academic, 2000.
2. P. Rümmer, "Generating Counterexamples for Java Dynamic Logic," *Proc. CADE-20 Workshop Disproving*, 2005, pp. 32–44; www.cs.chalmers.se/~ahrendt/cade20-ws-disproving/proceedings.pdf.
3. A.D. Brucker and B. Wolff, "Interactive Testing with HOLTestGen," *Proc. Workshop on Formal Aspects of Testing* (FATES 05), LNCS 3997, Springer, 2005, pp. 87–102.
4. B. Beckert, R. Hähnle, and P. Schmitt, eds., *Verification of Object-Oriented Software: The KeY Approach*, LNCS 4334, Springer, 2006.

Software Development by Refinement

Douglas R. Smith and Cordell Green,
Kestrel Institute

The term *automatic programming* has been used since computing's early days to refer to the generation of programs from higher-level descriptions. The first COBOL compilers were called automatic programmers, and since then the notion of what constitutes "higher level" description has been rising steadily. Here, we focus on automating the generation of correct-by-construction software from formal-requirements-level specifications. Interest in this topic arises from several disciplines, including AI, software engineering, programming languages, formal methods, and mathematical foundations of computing, resulting in a variety of approaches. Our work at the Kestrel Institute emphasizes formal representation and reasoning about expert design knowledge, thereby taking an AI approach to automated software development.

Automated software development (also called software synthesis) starts with real-world requirements that are formalized into specifications. Specifications then undergo a series of refinements that preserve properties while introducing implementation details. These refinements result from applying representations of abstract design knowledge to an intermediate design.

The sweet spot for automated software development lies in domain-specific modeling and specification languages accompanied by automatic code generators. Domain-specific code generation helps put programming power in the hands of domain experts who aren't skilled programmers. An active research topic is how to compose heterogeneous models to get a global model of a system. Another trend is toward increased modularity through aspect- and feature-oriented technologies. These techniques provide a novel means for automatically incorporating new requirements into existing code. Here, we summarize Kestrel's approach to automated

software development, which is embodied in the Specware system (www.specware.org).

Applications of software synthesis

Projects spanning various application areas convey a sense of what's possible in automated software development.

Synthesis of scheduling software

In the late 1990s, Kestrel developed a strategic airlift scheduler for the US Air Force that was entirely evolved at the specification level. From a first-order-logic specification, the application had approximately 24,000 lines of generated code. Using the Kestrel Interactive Development System (KIDS), more than 100 evolutionary derivations were carried out over a period of several years,[1] each consisting of approximately a dozen user design decisions. KIDS used abstract representations of knowledge about global search algorithms, constraint propagation, data type refinements (for example, sets to splay trees), and high-level code optimization techniques such as finite differencing and partial evaluation. The resulting code ran more than two orders of magnitude faster than comparable code that experts had manually written.

Continued progress in generating scheduling software led to the development of the domain-specific Planware system.[2] Planware defines a domain-specific requirement language for modeling planning and scheduling problems. From such a problem model (typically 100 to 1,000 lines of text derived from mixed text and graphical input), it automatically generates a complex planner/scheduler together with editors and visual display code (more than 100,000 lines of code for some applications). Design theories similar to the ones used in KIDS are specialized and fully automated in Planware, so code generation (or regeneration) takes only a few minutes.

Synthesis of Java Card applets

The current Java Card system uses a domain-specific language for specifying Java Card applets.[3] The system not only generates Java Card code but also produces formal proofs of consistency between the source specification and the target code. Certification is a major cost in the deployment of high-assurance software. By delivering both Java Card code and proofs, an external certifier can mechanically check the proofs' correctness. Our goal is to dramatically lower the cost of deploying software that must pass government certification processes. Elsewhere, NASA Ames' AutoBayes and AutoFilter projects are also exploring the simultaneous generation of code and certification evidence from domain-specific specifications.

Model-based code generation

The Forges project explored generating production-quality code from MatLab models. To do this, the project developed an operational semantics model of the StateFlow language. It then used partial evaluation of that operational semantics with respect to a given StateFlow model to produce executable code. The result was then optimized and sent to a C compiler. The code quality compared favorably to code that commercially available tools produced.

> Domain-specific code generation helps put programming power in the hands of domain experts who aren't skilled programmers.

Deriving authentication protocols

The Protocol Derivation Assistant tool explores the transformations and refinements needed to derive correct-by-construction authentication protocols. A major result has been the derivation of various well-known families of protocols.[4] Each family arises from choices of various features that are composed in. Hand-simulating this approach, Cathy Meadows and Dusko Pavlovic found a flaw in a protocol (called Group Domain of Interpretation, GDOI) that had been extensively studied and analyzed.

Automated policy enforcement

Many safety and security policies have effects that cut across a system's component structure. Recent theoretical work has shown that you can treat many such cross-cutting requirements as system invariants. Moreover, we've developed techniques for calculating where and how to modify the system to enforce those constraints. These techniques depend on automated inference, static analysis of program flow, and synthesis of many small segments of code for insertion at appropriate locations.

More details about Specware

Most of the example applications we discussed earlier were developed using Kestrel's Specware system.

From requirements to specifications

Refinement-oriented development starts with the procuring organization's requirements, which are typically a mixture of informal and semiformal notations reflecting various stakeholders' needs. To provide the basis for a clear contract, the stakeholders must formalize the requirements into specifications that both the procuring organization (the buyer) and the developer (the seller) can agree to. You can express specifications at various levels of abstraction. At one extreme, a suitable high-level programming language can sometimes serve to express executable specifications. However, an executable specification must include implementation detail that's time-consuming to develop and get right and that might be better left to the developer's discretion. At the other extreme, a property-oriented language (such as a higher-order logic) can be used to prescribe the intended software's properties with minimal prescription of implementation detail. The solution in Specware is a mixture of logic and high-level programming constructs providing a wide-spectrum approach. This lets specification writers choose an appropriate level of abstraction from implementation detail.

Refinement: From specifications to code

A formal specification serves as the central document of the development and evolution process. It's incrementally refined to executable code. A refinement typically embodies a well-defined unit of programming knowledge. Refinements can range from situation-specific or ad hoc rules, to domain-specific transformations, to domain-independent theories or representations of abstract algorithms, data structures, optimization techniques, software architectures, design patterns, protocol abstractions, and so on. KIDS and a Specware extension called Designware are systems that automate the construction of refinements from

reusable or abstract design theories. A crucial feature of a refinement from specification A to specification B is that it preserves A's properties and behaviors in B while typically adding more detail in B. This preservation property lets you compose refinements, meaning that you can treat a chain of refinements from an initial specification to a low-level executable specification as a single property- and behavior-preserving refinement, thereby establishing that the generated code satisfies the initial specification.

The Specware framework

Specware provides a mechanized framework for composing specifications and refining them into executable code. The framework is founded on a category of specifications. The specification language, called MetaSlang, is based on a higher-order logic with predicate subtypes and extended with a variety of ML-like programming constructs. MetaSlang supports pure property-oriented specifications, executable specifications, and mixtures of these two styles. You use specification morphisms to structure, parameterize, and refine specifications. You use colimits to compose specifications, instantiate parameterized specifications, and construct refinements. You use diagrams to express the structure of large specifications, the refinement of specifications to code, and the application of design knowledge to a specification. A recent Specware extension supports specifying, composing, and refining behavior through a category of abstract state machines.

The framework features a collection of techniques for constructing refinements based on formal representations of programming knowledge. Abstract algorithmic concepts, data type refinements, program optimization rules, software architectures, abstract user interfaces, and so on are represented as diagrams of specifications and morphisms. These diagrams can be arranged into taxonomies, which allow incremental access to and construction of refinements for particular requirement specifications.

Concluding remarks

A key feature of Kestrel's approach is the automated application of reusable refinements and the automated generation of refinements by specializing design theories. Previous attempts to manually construct and verify refinements have tended to require costly rework when requirements change. In contrast, automated construction of refinements allows larger-scale applications and more rapid development and evolution.

Near-term practical applications of automated software development will result from narrowing down the specification language and design knowledge to specific application domains, as in the Planware and Java Card prototypes we mentioned earlier. By narrowing the specification language, the generator developers can effectively hardwire a fixed sequence of design choices (of algorithms, data type refinements, and optimizations) into an automatic-design tactic. Also, by restricting the domain, a generator developer can specialize the design knowledge to the point that little or no inference is necessary, resulting in completely automated code generation.

> General-purpose theorem provers are typically inadequate to work with the sort of formulas that formal-methods tools generate.

References

1. D.R. Smith, "KIDS: A Semiautomatic Program Development System," *IEEE Trans. Software Eng.*, vol. 16, no. 9, 1990, pp. 1024–1043.
2. M. Becker, L. Gilham, and D.R. Smith, *Planware II: Synthesis of Schedulers for Complex Resource Systems*, tech. report, Kestrel Technology, 2003.
3. A. Coglio, "Toward Automatic Generation of Provably Correct Java Card Applets," *Proc. 5th Ecoop Workshop Formal Techniques for Java-like Programs*, 2003; ftp://ftp.kestrel.edu/pub/papers/coglio/ftjp03.pdf.
4. A. Datta et al., "A Derivation System for Security Protocols and Its Logical Formalization," *Proc. 16th IEEE Computer Security Foundations Workshop* (CSFW 03), IEEE CS Press, 2003, pp. 109–125.

Satisfiability Modulo Theories

Silvio Ranise, *INRIA*
Cesare Tinelli, *University of Iowa*

Many applications of formal methods rely on generating formulas of first-order logic and proving or disproving their validity. Despite the great progress in the last 20 years in automated theorem proving (and disproving) in FOL, general-purpose theorem provers, such as provers based on the resolution calculus, are typically inadequate to work with the sort of formulas that formal-methods tools generate. The main reason is that these tools aren't interested in validity in general but in validity with respect to some *background theory*, a logical theory that fixes the interpretations of certain predicate and function symbols. For instance, in formal methods involving the integers, we're only interested in showing that the formula

$$\forall x\, \forall y\, (x < y \Rightarrow x < y + y)$$

is true in those interpretations in which $<$ denotes the usual ordering over the integers and $+$ denotes the addition function. When proving a formula's validity, general-purpose reasoning methods have only one way to consider only the interpretations allowed by a background theory: add as a premise to the formula a conjunction of the theory's axioms. When this is possible at all (some background theories can't be captured by a finite set of FOL formulas), generic theorem provers usually perform unacceptably for realistic formal-method applications. A more viable alternative is using specialized reasoning methods for the background theory of interest. This is particularly the case for *ground* formulas, FOL formulas with no variables (so also no quantifiers) but possibly with *free constants*—constant symbols not in the background theory.

For many theories, specialized methods actually yield *decision procedures* for the validity of ground formulas or some subset of them. For instance, this is the case, thanks to classical results in mathematics, for the theory of integer numbers (and formulas with no multiplication symbols) or the theory of real numbers. In the last two decades, however, specialized decision procedures have also been discovered for a long (and growing) list of theories of other important data types, such as

- certain theories of arrays,

- certain theories of strings,
- several variants of the theory of finite sets,
- some theories of lattices,
- the theories of finite, regular, and infinite trees, and
- theories of lists, tuples, records, queues, hash tables, and bit vectors of a fixed or arbitrary finite size.

The literature on these procedures often describes them in terms of *satisfiability* in a theory—relying on the fact that a formula is valid in a theory T exactly when no interpretation of T satisfies the formula's negation. So, we call the field *satisfiability modulo theories* and call those procedures *SMT solvers*.

Using SMT solvers in formal methods isn't new. It was championed in the early 1980s by Greg Nelson and Derek Oppen at Stanford University, by Robert Shostak at SRI, and by Robert Boyer and J Strother Moore at the University of Texas at Austin. Building on this work, however, the last 10 years have seen an explosion of interest and research on the foundations and practical aspects of SMT. Several SMT solvers have been developed in academia and industry with continually increasing scope and improved performance. Some of them have been or are being integrated into

- interactive theorem provers for high-order logic (such as HOL and Isabelle),
- extended static checkers (such as CAsCaDE, Boogie, and ESC/Java),
- verification systems (such as ACL2, Caduceus, SAL, and UCLID),
- formal CASE environments (such as KeY),
- model checkers (such as BLAST, MAGIC, and SLAM),
- certifying compilers (such as Touchstone), and
- unit test generators (such as CUTE and MUTT).

In industry, Cadence, Intel, Microsoft, and NEC (among others) are undertaking SMT-related research projects.

Main approaches

The design, proof of correctness, and implementation of SMT methods pose several challenges. First, formal methods naturally involve more than one data type, each with its own background theory, so suitable combination techniques are necessary. Second, satisfiability procedures must be proved sound and complete. Although proving soundness is usually easy, proving completeness requires specific model construction arguments showing that, whenever the procedure finds a formula satisfiable, a satisfying theory interpretation for it does indeed exist. This means that each new procedure in principle requires a new completeness proof. Third, data structures and algorithms for a new procedure, precisely for being specialized, are often implemented from scratch, with little software reuse. Three major approaches for implementing SMT solvers currently exist, each addressing these challenges differently and having its own pros and cons.

> Several SMT solvers have been developed in academia and industry with continually increasing scope and improved performance.

SAT encodings

This approach is based on ad hoc translations that convert an input formula and relevant consequences of its background theory into an equisatisfiable propositional formula (see, for instance, Ofer Strichman, Sanjit A. Seshia, and Randal E. Bryant's work[1]). The approach applies in principle to all theories whose ground satisfiability problem is decidable, but possibly at the cost of an exponential blow-up in the translation (that is, the size of the equisatisfiable formula the translation produces is exponentially larger than the size of the original formula). The approach is nevertheless appealing because SAT solvers can quickly process extremely large formulas. Proving soundness and completeness is relatively simple because it reduces to proving that the translation preserves satisfiability. Also, the implementation effort is relatively small for being limited to the translator—after that, you can use any off-the-shelf SAT solver. Eager versions of the approach, which first generate the complete translation and then pass it to a SAT solver, have recently produced competitive solvers for the theory of equality and for certain fragments of the integers theory. However, current SAT encodings don't scale up as well as SMT solvers based on the small engines approach (which we discuss next) because of the exponential blow-up of the eager translation and the difficulty of combining encodings for different theories.

Small engines

This approach is the most popular and consists of building procedures implementing an inference system specialized on a theory T. The lure of these "small engines" is that you can use whatever algorithms and data structures are best for T, typically leading to better performance. A disadvantage is that proving an ad hoc procedure's correctness might be nontrivial. A possibly bigger disadvantage is that you must write an entire solver for each new theory, possibly duplicating internal functionalities and implementation effort.

One way to address the latter problem is to reduce a theory solver to its essence by separating generic Boolean reasoning from theory reasoning proper. The common practice is to write theory solvers just for conjunctions of ground *literals*—atomic formulas and negations of atomic formulas. These pared-down solvers are then embedded as separate submodules into an efficient SAT solver, letting the joint system accept arbitrary ground formulas. Such a scheme (formulated generally and abstractly by Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli[2]) lets you plug in a new theory solver into the same SAT engine as long as the solver conforms to a simple common interface.

Another way to reduce development costs is to decompose, when possible, a background theory into two or more component theories, write a solver for each smaller theory, and then use the solvers cooperatively. Greg Nelson and Derek Oppen developed a general and highly influential method for doing this.[3] All major SMT solvers based on the small-engines approach use some enhancement or variation of this method.

Big engines

You can apply this rather recent approach

to theories T that admit a finite FOL axiomatization, capitalizing on the power and flexibility of current automated theorem provers for FOL. In particular, you can apply it to provers based on the *superposition calculus*, a modern version of resolution with a built-in treatment of the equality predicate and powerful techniques for reducing the search space. The approach consists of instrumenting a superposition prover with specialized control strategies that, together with the axioms of T, effectively turn this "big engine" into a decision procedure for ground satisfiability in T.

A big plus of the approach is a simplified proof of correctness, which reduces to a routine termination proof for an exhaustive and fair application of the superposition calculus' rules.[4] Another advantage is that, when the approach applies to two theories, obtaining a decision procedure for their union is, under reasonable assumptions, as simple as feeding the union of the axioms to the prover. A further advantage is the reuse of efficient data structures and algorithms for automated deduction implemented in state-of-the-art provers.

The main disadvantage is that to get additional functionalities (such as incrementality, proof production, or model building), you might need to modify the prover in ways that the original implementers didn't foresee, which might require a considerable implementation (or reimplementation) effort.

Standardization efforts

Because of their specialized nature, different SMT solvers often are based on different FOL variants, work with different theories, deal with different classes of formulas, and have different interfaces and input formats. Until a few years ago, this made it arduous to assess the relative merits of existing SMT solvers and techniques, theoretically or in practice. In fact, even testing and evaluating a single solver in isolation was difficult because of the dearth of benchmarks.

To mitigate these problems, in 2002 the SMT community launched the SMT-LIB (Satisfiability Modulo Theories Library) initiative, a standardization effort co-led by us and supported by the vast majority of SMT research groups. The initiative's main goals are to define standard I/O formats and interfaces for SMT solvers and to build a large online repository of benchmarks for several theories. Several SMT solvers worldwide now support the current version of the input format, and numerous formal-methods applications are adopting it. The repository, which is still growing, includes about 40,000 benchmarks from academia and industry. (See www.smt-lib.org for more details on SMT-LIB and on SMT-COMP, the affiliated solver competition.)

Future directions

Although SMT tools are proving increasingly useful in formal-methods applications, more work is needed to improve the trade-off between their efficiency and their expressiveness. For example, software verification problems often require handling formulas with quantifiers, something that SMT solvers don't yet do satisfactorily. Finding good heuristics for lifting current SMT techniques from ground formulas to quantified formulas is a key challenge.

Software verification problems often require handling formulas with quantifiers, something that SMT solvers don't yet do satisfactorily.

Other lines of further research come from the need to enhance SMT solvers' interface functionalities. For instance, to validate the results of a solver or integrate it in interactive provers, we need solvers that can produce a machine-checkable proof every time they declare a formula to be unsatisfiable. Similarly, to be useful to larger tools such as static checkers, model checkers, and test set generators, an SMT solver must be able to produce, for a formula it finds satisfiable, a concrete, finite representation of the interpretation that satisfies it. Other potentially useful functionalities are the generation of unsatisfiable cores of unsatisfiable formulas or of interpolants for pairs of jointly unsatisfiable formulas. More research on efficiently realizing all these functionalities is under way.

References

1. O. Strichman, S.A. Seshia, and R.E. Bryant, "Deciding Separation Formulas with SAT," *Proc. 14th Int'l Conf. Computer Aided Verification* (CAV 02), LNCS 2404, Springer, 2002, pp. 209–222.
2. R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Abstract DPLL and Abstract DPLL Modulo Theories," *Proc. 11th Int'l Conf. Logic for Programming, Artificial Intelligence, and Reasoning* (LPAR 04), LNCS 3452, Springer, 2005, pp. 36–50.
3. G. Nelson and D.C. Oppen, "Simplification by Cooperating Decision Procedures," *ACM Trans. Programming Languages and Systems*, vol. 1, no. 2, 1979, pp. 245–257.
4. A. Armando, S. Ranise, and M. Rusinowitch, "A Rewriting Approach to Satisfiability Procedures," *Information and Computation*, June 2003, pp. 140–164.

Static Driver Verifier: An Industrial Application of Formal Methods

Thomas Ball and Sriram K. Rajamani,
Microsoft Research

Microsoft will soon release Windows Vista, the latest in its line of PC operating systems. Each version of the Windows OS has simultaneous releases of development kits that let third-party developers create software that builds upon it. An important class of software that extends Windows is device drivers (or simply "drivers"), which connect the huge and diverse number of devices to the Windows OS. The Windows Driver Kit (www.microsoft.com/whdc/devtools/ddk/default.mspx) "provides a build environment, tools driver samples and documentation to support driver development for the Windows family of operating systems."

The Static Driver Verifier tool, part of the Windows Driver Kit, uses static analysis to check if Windows device drivers obey the rules of the Windows Driver Model (WDM), a subset of the Windows kernel interface that device drivers use to link into the operating system. The SDV tool is powered by the SLAM software-model-checking engine,[1] which uses several formal methods such as model checking, theorem proving, and static program analysis. We describe technical and nontechnical factors that led to this tool's successful adoption and release.

Successful applications of formal meth-

ods in commercial software engineering are rare for several reasons, including

- the lack of a compelling value proposition to users,
- the high cost of obtaining formal specifications, and
- a lack of scalability and automation in tools based on formal methods.

We describe how the SLAM engine and SDV address these obstacles.

SDV's value proposition

The biggest reason SDV succeeded is the importance of the driver reliability problem. The reliable operation of device drivers is fundamental to the reliability of Windows. According to Rob Short, Corporate Vice President of Windows Core Technology, failed drivers cause 85 percent of Windows XP crashes.[2] Device-driver failures appear to the customer as Windows failures. So, Microsoft invests heavily in improving the Windows Driver Kit, which includes new driver models (interfaces with supporting libraries) as well as a diverse set of tools to catch bugs early in the development of device drivers.

SDV's value proposition is that it detects errors in device drivers that are hard to find using conventional testing. SDV places a driver in a hostile environment and uses SLAM to systematically test all code paths while looking for violations of WDM usage rules. SLAM's symbolic execution makes few assumptions about the state of the OS or the driver's initial state, so it can exercise situations that are difficult to exercise by traditional testing.

The cost of creating formal specifications

SDV analyzes a driver (written in C) against interface rules for the WDM. These rules are written in a state-machine description language called SLIC (Specification Language for Interface Checking). Because many thousands of drivers use the same WDM interface, it made economic sense for Microsoft to write the specifications because the cost is amortized by running SDV across many drivers. Furthermore, the specifications themselves are programs, and, like all programs, they're error prone. As the specifications were used on drivers, they were debugged, and only those that produced valuable results are part of the SDV release. Creating the SLIC specifications as well as the environment model (a C program) representing how Windows invokes a device driver took several staff-years of effort.

Scalability and automation

Exhaustively analyzing all possible behaviors of large programs isn't possible because programs have an astronomical number of behaviors. Instead, the SLAM engine focuses on an "abstraction" of the program, which contains only the state of the program that's relevant to the property being verified. For example, to check if a driver is properly using a spinlock, SLAM tracks only the state of the lock itself and the variables in the driver that guard the state transitions involving the lock. SLAM uses a technique called counterexample-driven refinement to automatically discover the relevant state. The SLAM toolkit uses Boolean programs as models to represent these abstractions. Boolean programs are simple, and property checking for Boolean programs is decidable. Our experience with SLAM demonstrates that for control-dominated properties, this model suffices to represent useful abstractions.

Since SLAM's invention, many in academia and industry have done considerable work on improving counterexample-driven refinement for software and addressing key related problems.

SLAM's contributions

SLAM builds on much work from the formal-methods community, such as symbolic model checking, predicate abstraction, abstract interpretation, counterexample-guided abstraction refinement, and automatic theorem proving using decision procedures.

SLAM's research contribution is to apply these concepts in the context of programming languages. Its abstraction tool is the first to solve the problem of modularly building a sound predicate abstraction (Boolean program) of a C program. SLAM's model checker is a symbolic dataflow analysis engine (a cross between an interprocedural analysis and a model checker) that checks Boolean programs. SLAM symbolically executes a C program path to determine its feasibility and to discover predicates to eliminate infeasible execution paths, thus refining the Boolean program abstraction. Adapting existing techniques, such as predicate abstraction, to work in the context of a real-world programming language required new research, in addition to the engineering effort to make them scale and perform acceptably.

Directions

Since SLAM's invention, many in academia and industry have done considerable work on improving counterexample-driven refinement for software and addressing key related problems.

Performance

Of course, users want tools to run in minutes rather than hours (and seconds are even better). Increased performance also lets tool providers more quickly debug the properties that their tools take as input. Recent work on using interpolant-based model checking (rather than predicate abstraction) appears promising, as do approaches that leverage recent advances in satisfiability solvers.

Precision

SLAM and many other tools sacrifice precision for performance. For example, SLAM doesn't precisely model integer overflow or bit vector operations. Using more precise decision procedures for modular arithmetic and bit vectors would reduce the number of false error reports in tools such as SLAM and might also increase performance.

Explaining the cause of errors

Tools such as SLAM produce error paths that are often long and contain much irrelevant information. We and others have shown that you can automatically prune away this irrelevant information and present a much more concise, relevant error report.

Inferring specifications

As we noted earlier, developing useful specifications is error prone and time consuming. Recent work on automatically infer-

ring simple specifications from a code corpus shows much promise and will be an essential part of methodologies for inserting software tools such as SLAM into legacy environments. ⬒

Acknowledgments

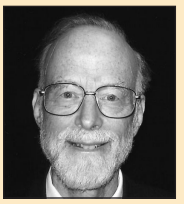
We're grateful for the contributions of many people in academia and inside Microsoft. In addition to providing the theoretic and algorithmic foundations over several decades, on which SLAM relied, the academic community was generous to license software to be used inside SDV. In particular, the CUDD (CU Decision Diagram) Binary Decision Diagram package (http://vlsi.colorado.edu/~fabio/CUDD) from the University of Colorado and the OCaml (http://caml.inria.fr) runtime environment from INRIA ship as part of SDV. This wouldn't have been possible without cooperation from the researchers who built these tools to work out the legal agreements that enabled Microsoft to include these components.

The Windows product group showed remarkable foresight in signing up to ship SLAM, which was a research prototype at that time. In addition to support from senior management, the SDV development team in Windows invested many staff-years of time and energy in turning SDV from a research prototype to a product. Without their efforts, SDV wouldn't be a shipping Microsoft product today.

References

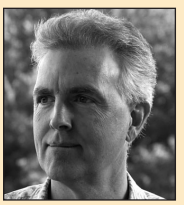
1. T. Ball et al., "Thorough Static Analysis of Device Drivers," *Proc. European Systems Conf.* (EuroSys 06), 2006, pp. 73–85.
2. M.M. Swift et al., "Recovering Device Drivers," *Proc. 6th USENIX Symp. Operating Systems Design and Implementation* (OSDI 04), USENIX, 2004, pp. 1–16.

Bernhard Beckert is an assistant professor of computer science at the University of Koblenz, Germany. Contact him at beckert@uni-koblenz.de.

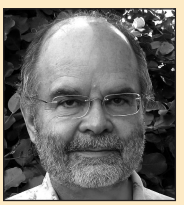


Tony Hoare is a senior researcher with Microsoft Research in Cambridge and emeritus professor at Oxford University. Contact him at thoare@microsoft.com.

Reiner Hähnle is a professor in the Department of Computer Science at Chalmers University of Technology in Gothenburg, Sweden. Contact him at reiner@chalmers.se.



Douglas R. Smith is principal scientist with the Kestrel Institute and the executive vice president and chief technology officer of Kestrel Technology. Contact him at smith@kestrel.edu.

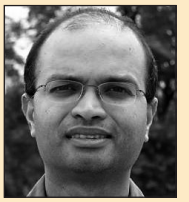


Cordell Green is the founder, director, and chief scientist of the Kestrel Institute. Contact him at green@kestrel.edu.

Silvio Ranise has a permanent research position at INRIA, in the LORIA laboratory common to CNRS (Centre National de la Recherche Scientifique), INRIA, and the Universities of Nancy. Contact him at silvio. ranise@loria.fr.

Cesare Tinelli is an associate professor of computer science at the University of Iowa. Contact him at tinelli@cs.uiowa.edu.

Thomas Ball is a principal researcher and research manager at Microsoft Research, Redmond. Contact him at tball@microsoft.com.



Sriram K. Rajamani is a senior researcher and research manager at Microsoft Research India, Bangalore. Contact him at sriram@microsoft.com.