Invited paper, presented at the 4th IEEE International Conference on Software Engineering and Formal Methods (SEFM), Pune, India, September 2006. Appears in the SEFM proceedings pp. 21–28 ©IEEE Computer Society.

Harnessing Disruptive Innovation in Formal Verification*

John Rushby
Computer Science Laboratory,
SRI International
Menlo Park CA USA
rushby@csl.sri.com

Abstract

Technological innovations are sweeping through the field of formal verification. These changes are disruptive to tools based on interactive theorem proving, which needs new ways to integrate the capabilities of novel technologies.

I describe two approaches. One is development and use of SMT solvers: these use techniques from theorem proving but apply them in ways that enable model checking, while also supporting highly automated theorem proving. The other is a proposal for an Evidential Tool Bus: a loosely coupled architecture that allows many different verification components to collaborate to solve problems beyond the capability of any single component.

1. Historical Introduction

Ever since the first program verification systems of King [35] and Good [27], theorem provers have played an important part in the mechanically-assisted analysis of computer programs and software systems. Theorem provers have evolved over the years to better support this application, principally through improved automation for reasoning about arithmetic, data structures, and recursively or inductively defined functions and relations. During this time, the focus of formal verification has expanded from verifying small concrete programs to analyzing rather intricate (often concurrent) algorithms and the specifications (rather than the code) for fairly complex systems. These algorithms and specifications often are formalized directly in the notation of the theorem prover concerned, and these notations also have evolved, principally through use of higher-order logic and rich type systems, so that they provide attractive environments for formal specification. Until recently, the large majority of substantial formal verifications were performed using theorem provers of this kind, such as ACL2 [34], Coq [6], HOL [28] (and HOL Light [29] and Isabelle HOL [40]), NuPRL [12], and PVS [42].

Then, in the early 1990s, effective finite state model checkers such as Spin [32] and Murϕ [21] (both of these use explicit state representations) and SMV [38] (which uses a symbolic representation based on BDDs) became widely available. The limitation to finite state meant that most specifications had to be severely "downscaled" by Draconian restrictions on the size of data structures. Such downscaling usually does not preserve correctness (sometimes it does not preserve incorrectness either) and this compromises model checking as an approach to verification—hence, its early uses were mostly for refutation (i.e., bug finding), but it was highly effective for that purpose.

The invention of predicate abstraction [47] allowed model checking to expand from refutation to verification: rather than arbitrarily downscale an algorithm or system specification to finite state, predicate abstraction provides a mechanizable approach to the construction of property-preserving abstractions—so that model checking the finite state abstraction does verify the original specification. Initially, manual guidance was needed to select the predicates on which to abstract, but it was soon recognized that the predicates appearing explicitly in the specification provide a good starting point, and that the selection can iteratively be refined through examination of the counterexamples produced by model checking inadequate abstractions [45].

This approach evolved into the automated methodology of counterexample-guided abstraction refinement (CEGAR) [11], which employs a loop comprising abstraction, model checking, counterexample generation and analysis, and abstraction refinement. Decision procedures are used to construct the abstractions, and for software specifications (where the concrete transition relation may be too large or too complex to manipulate directly) these are often constructed and explored "on the fly" with an explicit state model checker; any counterexample produced by model checking an abstraction is checked in its concrete interpretation by a satisfiability solver and the proof of unsatisfi-

* This research was supported by SRI International.

ability generated by the solver (in the case that the concrete instance is not a counterexample) is mined for information (e.g., using interpolants or unsatisfiable cores) that will guide refinement of the abstraction [2, 30].

Thus, model checking grew from a technology that originally was seen as a useful adjunct to full verification by theorem proving (e.g., to debug a specification prior to verifying it), into one that increasingly seems to rival theorem proving for many verification tasks.

A similar historical trend can be seen in static analysis. The early applications of abstract interpretation [13], for example, were in optimizing compilers (for constant propagation, data flow analysis, and so on). These were then extended to strong notions of type correctness (e.g., guaranteeing no null pointer dereferencing, no division by zero, etc.) and then to arithmetic properties (e.g., no array bound violations, no floating point overflows[1]). Recent applications include calculation of worst case execution time and stack depth[2], and the integrity of data structures (e.g., shape analysis [46] and separation logic [5]). Thus, verification of certain properties that formerly required interactively guided theorem proving are now fully automated by static analyzers.

So much for the past; what of the future?

2. Disruptive Innovation

The emergence of methods based on model checking and static analysis as rivals to interactively guided theorem proving for mechanized formal verification can be seen as an instance of "low-end disruptive innovation" [9, 10]. This kind of innovation occurs when a technology, aimed at customers who do not require the full performance of an incumbent high-end technology, maintains a rapid rate of improvement and eventually overhauls and displaces the incumbent. Standard examples of low-end disruptive innovation are the microprocessor (vs. mainframes) and digital (vs. film) photography.

Low-end disruption poses a challenge to developers of incumbent high-end technologies. One response attempts to neutralize the disruption by incorporating its innovations into the incumbent technology. The danger here is that continued innovation may generate new disruptions and the combination technology may always lag the latest innovations: this is especially likely if the developers of the incumbent technology are not active participants in the continued innovations. A second response, a dual of the first, is actively to further the disruptive innovation by exploiting insights from deep knowledge of the incumbent technology. A third response is to attempt additional disruption (to disrupt the disruptors) by some innovation that exploits new possibilities discovered in the combination of the incumbent and the disruptive technologies.

The previous paragraph describes an incumbent's possible responses to disruptive innovation from the point of view of an enterprise reacting to erosion of its market share or its profitability, but I think we can usefully apply similar analysis to choices in research directions. Here, the point is not to protect the "market share" of theorem proving in verification against the encroachment of model checking and static analysis, but to ask how their innovations can stimulate new approaches to theorem proving and to its combination with model checking and static analysis and their mutual application to formal verification, for the benefit of all.

In the following sections, I will outline two ways in which theorem proving can respond to recent disruptive innovations. One, which is already beginning to demonstrate its value, is SMT solving. This can be seen as an instance of the second response outlined earlier: it is an innovation that takes some techniques from theorem proving and reengineers and reinterprets them in ways that prove disruptive in their own right. The other, an Evidential Tool Bus, is more speculative; it is a proposal for combining many different kinds of formal methods tools in a new way, and can be seen as an instance of the third response mentioned above.

3. SMT Solvers: A Disruptive Innovation

Decision procedures for linear arithmetic and other theories useful in specifying software are important components in theorem provers used for verification, such as ACL2 and PVS. Decision procedures operate over *conjunctions* of formulas in their theories; the case analysis needed to analyze disjunctions and general propositional structure is performed by the main body of the theorem prover in order to better control "case explosion," which bedevils theorem proving just as "state explosion" does model checking. However, this caution in case analysis (and expansion of definitions) is important mostly in the higher "strategic" levels of a proof: the lower level "endgame" branches of a proof often can be discharged by brute force—if sufficient brute force can be brought to bear. Strategies such as `grind` in PVS do this quite successfully; `grind` uses decision procedures in loose combination with a BDD-based simplifier for propositional structure. However, the vast improvements in propositional satisfiability (SAT) solvers over the last five years (starting with Chaff [39]) now make the combination of decision procedures with SAT solvers a more interesting option.

Integrating decision procedures with a SAT solver yields a solver for "satisfiability modulo theories," generally known as an SMT solver. There are several ways to construct such solvers, but the most successful is the "lazy"

1 See `http://www.astree.ens.fr`
2 See `http://www.absint.com`

integration [18] pioneered in ICS [14], which began as a project to make the capabilities of the decision procedures in PVS available separately. In the lazy integration, all non Boolean terms are abstracted to propositions and the SAT solver is asked to generate a satisfying assignment; if there are none, the formula is unsatisfiable and we are done, otherwise, the original interpretation of the non Boolean terms is restored and the conjunction of their truth assignments is asserted to the decision procedure (e.g., if $x \leq y$ is abstracted to P and $x < y + 1$ is abstracted to Q and the SAT solver assigns P to *true* and Q to *false*, then we assert $(x \leq y) \wedge \neg(x < y + 1)$ to the decision procedure). If the decision procedure verifies the conjunction, we are done (the formula is satisfiable); if not, we mine the conjunction for an "explanation" of its invalidity and assert its propositional abstraction (e.g., P *iff* Q in the example above) to the SAT solver as an additional clause that will prevent it generating similarly poor assignments in future, and ask it for a new assignment. The whole process repeats until it yields a result, or the SAT solver exhausts its search space (in which case the formula is unsatisfiable).

The basic lazy integration just described must be very carefully engineered to deliver high performance, rather as modern SAT solvers are carefully engineered—though SMT solvers are far more complex than SAT solvers. In particular, the SAT solver and decision procedures must be very tightly integrated, and the decision procedures must yield high-quality explanations (to achieve maximum pruning of the search space) at low cost. An experimental evaluation of SMT solvers in 2004 [17] led to the initiation of an annual competition (the 2005 results are reported in [4], the 2006 results will be available in August), which has spurred dramatic improvement in the performance of SMT solvers: modern SMT solvers routinely solve problems with tens of thousand of constraints and variables. At SRI, the culmination of this line of development is the SMT solver Yices[3], which was developed by Leonardo de Moura and Bruno Dutertre. Yices not only decides linear arithmetic on integers and reals (with optimizations for the important special case of difference logic: that is, constraints of the form $x - y \leq c$), but fixed size bitvectors, equality with uninterpreted functions, recursive datatypes (such as lists and trees), arrays, tuples, and records of all these, and some quantified formulas.

High-performance SAT solvers enabled the development of bounded model checkers [7]: these calculate the symbolic representation of the k-step unfolding of the system specification (for some explicit bound k, such as 20) and then seek an assignment to the open variables that will refute a given safety property. Bounded model checking can often handle larger problems than BDD-based model checkers, and can be extended from refutation to verification by a variety of methods, including k-induction [49] (which is ordinary inductive invariance, extended to require k steps in the basis and in the antecedent to the inductive step; ordinary induction is 1-induction in this framework, and larger values of k yield stronger proof methods).

By direct extension, SMT solvers enable development of bounded model checkers for infinite state systems: whereas a finite-state SAT-based model checker must represent (range-restricted) integers as fixed-width bitvectors, and must synthesize symbolic representations of hardware circuits to perform arithmetic operations (e.g., a ripple-carry adder for addition), an SMT-based model checker can deal with arithmetic on unbounded integers and reals using its decision procedures. These "infinite bounded model checkers," such as the one for SAL [19, 16], combine the automation and other benefits of model checking (such as counterexamples, and a language oriented to the specification of computational systems rather than arbitrary mathematics) with the broader applicability of theorem proving, and I believe they will prove disruptive to both technologies and to several application domains.

Even for restricted arithmetic, a bounded model checker using an SMT solver is often faster than one using a SAT solver with the bitvector representation, and the ability use uninterpreted functions often allows large parts of a specification to be abstracted away—for example, in verifying correctness for the bypass logic of a microprocessor pipeline we can represent the ALU as an uninterpreted function, whereas finite state model checking must use an explicit circuit, thereby incurring additional cost and the risk of unsoundness (if the chosen circuit happens to mask the manifestations of a bug in the bypass logic). The ability to reason effectively over real numbers also allows infinite bounded model checkers to encroach on the space of specialized model checkers for infinite-state systems, such as those for timed systems (e.g., UPPAAL [37]). Dutertre and Sorea [22] describe how timed systems can be represented and analyzed effectively in SAL, and Brown and Pike [43] describe some optimizations and their application to an example (the Biphase Mark protocol) that previously had posed a formidable challenge to both theorem proving and even to the combination of theorem proving with model checking for timed automata. Brown and Pike's fully parameterized treatment in SAL requires less than 300 lines of specification and is verified in under ten seconds, whereas previous treatments required thousands of proof steps and hours of computer time. This approach has some disadvantages compared with model checkers for timed automata—for example, the user must invent the "disjunctive invariant" [44] to be verified, whereas timed automata do reachability analysis—but is more widely applicable (for example, the underlying SMT solver can deal with the case ex-

3 See http://yices.csl.sri.com

plosion when fault-tolerance is added to real-time). Recent work [36] demonstrates that SMT solvers can be very effective in calculating predicate abstractions, and it is likely that this can aid the construction of suitable invariants.

In use, an infinite bounded model checker with k-induction combines many of the attractive features of both model checking and theorem proving. As with a model checker, incorrect conjectures quickly yield explicit counterexamples that manifest their falsehood (provided the SMT solver can reach sufficiently deep instances) and, as with a theorem prover, valid conjectures can be proved by induction and the use of lemmas. Unlike a theorem prover, conjectures that are insufficiently strong to support the inductive step yield an explicit counterexample that can help identify suitable strengthenings or lemmas. (If the conjecture is valid, the first state of the counterexample must be unreachable and an effective strategy is to seek strengthenings and lemmas that exclude this state.) Also unlike an interactive theorem prover, each attempted proof is fully automatic and usually takes only a few seconds.

SMT solvers enable construction of infinite bounded model checkers as rivals to interactive theorem provers, but interactive theorem provers can also use the same SMT solvers as backends. We are in the process of engineering this combination with PVS and Yices, and it will be interesting to determine whether, and in what cases, the total human time taken to develop a verification is reduced by use of an infinite bounded model checker compared with an interactive prover that uses the same SMT solver.

4. The Evidential Tool Bus: Harnessing Disruptive Innovations

Model checking, abstract interpretation and other methods for static analysis, SMT solvers, predicate abstraction, and other new technologies may all be disruptive to formal verification based on theorem proving, but each of these technologies tends to be trapped inside its own tools: their full capabilities are seldom exported to the API of the tool concerned (e.g., BLAST [30] uses predicate abstraction to discover invariants that it uses in verifying C programs, but it does not normally disclose those invariants), and their implementations are often large and complex and not readily extracted for use in other tools. Even if the full capabilities of a technology were liberated from its tool, it would not be easy to add it to some other tool—because most tools are built around a particular technology and lack the conceptual and practical "openings" into which additional technologies can be inserted. For example, verification of concurrent systems by interactive theorem proving is bedeviled by the cost and difficulty of developing suitably strong invariants, but existing interactive theorem provers lack the means to interact with the technologies (such as predicate abstraction and dynamic analysis [24]) that can propose and strengthen invariants.

There is thus a "market failure" in verification technology: component technologies are not made available as separate implementations because there are no users for them, and established tools cannot take advantage of new technologies because they are too tightly integrated around their current technologies.

Verification problems are seldom a perfect match to the capabilities of any given tool, so users must compromise and use the one they believe to be the best match, or must construct *ad hoc*, often manual, "tool chains" to allow use of, say, a model checker and a theorem prover. Some theorem provers do integrate other technologies—for example, PVS has both a symbolic model checker and a predicate abstractor—but the basic paradigm is one of theorem proving, so some of the capabilities of the other technology may be lost if they do not fit the paradigm (e.g., the PVS model checker does not provide counterexamples; this is because the theorem prover expects its backend components to deliver either truth values or a list of subgoals—it has no way to interpret the sequence of states that comprises a counterexample), and the complexities of integration generally cause the other technology to lag the state of its art. These integrations, which are only modestly successful, can be seen as instances of the first response to disruptive innovation outlined in Section 2.

We need a new framework for verification tools that allows use of all technologies that can contribute to the overall goal, and that allows them to work cooperatively, but without the burdens of tight integration. Such a framework will encourage development of components that do a single thing well, and the collection of such components will be more than the sum of its parts. I propose the *Evidential Tool Bus* (ETB) as such a framework: it provides a way to loosely integrate verification components so that they can collaborate to solve problems beyond the capability of any single component. The idea is related to coordination languages such as Linda [26] and to Web services and service-oriented architectures, but it is specialized to verification and this allows it to be simpler, and at the same time powerful.

In any decentralized service-oriented system, there must be some agreed ontology for describing the services requested and performed by components, and the data on which they operate. General purpose ontologies are challenging to develop and to specify, but formal verification is concerned with formulas in logic, so logic provides a convenient and rich foundation for the ontology of the ETB. Different verification components use different fragments of logic (e.g., propositional, quantified), and use them to describe different things (e.g., formulas, state machines, counterexamples, unsat cores), so the tool bus logic must admit numerous specializations along these two dimensions,

which we refer to as the dimensions of sublogics and of representations, respectively.

The ETB ontology will be expressed over these sublogics and representations (e.g., "this is a counterexample on a state machine expressed over propositional combinations of linear arithmetic on integers with uninterpreted functions"). A higher-order logic expressed in XML is a candidate for the top of the tool bus sublogic hierarchy; the collections of sublogics and representations will be open ended so that new components with new requirements can readily be accommodated.

The ETB will manipulate *judgments* on this ontology: a judgment is an assertion of the form $T \vdash E : A$, which denotes the claim that component tool instance T provides evidence E for assertion A. Evidence may be generated only when explicitly requested, so this field can default to empty. Typical assertions include:

1. F *is a well-typed formula in context* τ,
2. $\mathcal{C}$ *is a decision procedure context representing the input* Γ,
3. Ξ *is a satisfying assignment for* F *in theory* $\mathcal{T}$,
4. Θ *is a minimal unsatisfiable set of literals from* F *in theory* $\mathcal{T}$.

Components are viewed as oracles that can verify assertions and construct evidence for them. Each component builds judgments by forward chaining from existing judgments or backward chaining through the generation of proof obligations. The assertions in judgments may contain variables. For example, the assertion "*?m is the abstraction of state machine* M *wrt. predicates* Γ" where *?m* is a variable invites some component to construct a predicate abstraction; it will then return a judgment containing the assertion "$\hat{M}$ *is the abstraction of state machine M wrt. predicates* Γ" where $\hat{M}$ is indeed the predicate abstraction of M with respect to the predicates Γ together with some evidence for this.

Some components may be scripts that describe a recipe for performing standard kinds of analysis (e.g., "to model check an infinite state system, first get a finite state abstraction, then model check that"), and others may be interactively guided by a human user. Scripts resemble the strategies of PVS or the tactics of a HOL-style prover, but instead of classical proof steps, they invoke the capabilities of modern verification components: generation of invariants, test cases, construction of abstractions, mining of counterexamples, and so on.

Judgments are both the means for chaining inferences together—so that a suitable chain of judgments constitutes a proof for a given conjecture—and the means of resource discovery and invocation in the ETB. To its components, the ETB looks like a virtual blackboard to which judgments are posted: a component may be invoked explicitly by posting a judgment with its name in the tool field. Judgments may also use a variable as the name of the tool, in which case any component that can use the judgment may then go to work on it (possibly in competition with other components); the identity of the component concerned will appear in the judgments that it posts in return.

The main virtues of the interaction model used in the ETB are that it is open ended and loosely coupled. We do not need to know what components are "out there": new and experimental components can be added at any time and will be activated whenever they can contribute to the analysis of a posted judgment. To reduce bandwidth and other resources, large data structures (e.g., a BDD) will be transmitted only on demand—normally an identifying "handle" will be transmitted instead. The loose coupling allows components to be designed to perform a single analysis well, and in ignorance of what other components are available. Of course, finding what analyses and capabilities are really useful will depend on experimentation and consideration of the services provided by other components, but a component's specification and implementation may focus purely on the judgments that it supports.

The forward and backward chaining of the ETB resemble a distributed logic programming framework such as SRI's Open Agent Architecture (OAA) [8], but its lower-level virtual blackboard closer to the tuple-space model of Linda [26] (there are also similarities with The Information Bus [41]). The ETB will also need some simple task control functions to terminate components whose results are no longer required and to release storage of data structures whose handles have been finalized, and also an explicit resource discovery function so that human users can gain an understanding of available components, and their capabilities. An instance of the ETB could operate within a single machine, but it naturally lends itself to distribution—both within institutions and across them—where it could allow developers of different component technologies to interact seamlessly among themselves and with their users.

The ETB is intended to support interactions of a fairly coarse granularity: those that occur one to a few thousand times in the course of an overall analysis; it is not intended to support the tight, fine-grained interactions that occur inside, say, an SMT solver (where a single analysis may involve billions of interactions between decision procedures and SAT solver), nor the very simple interactions that occur between a proof assistant and its backend solvers. These limitations reflect the novel ground targeted by the ETB: the tight and the simple interactions are already served by known methods. Judgments in the ETB provide a form of semantic integration that is different than prior work such as MATHWEB [25], PROSPER [20], or the Logic Broker Ar-

chitecture [3], which has focused on operational aspects of integration.

We anticipate that the ETB will be used in contexts, such as certification for safety or security, where credible evidence may be demanded for any claim. Because evidence is crucial to some applications, we treat it as a first class entity and that is why we call it the *Evidential* Tool Bus. However, although the ETB manages evidence, it does not specify or restrict its form; in particular, it does not require that evidence is reduced to a preselected set of elementary proof steps.

In some circumstances, the reputation of a component may be considered sufficient evidence; in others, diverse verifications using different components (e.g., two different SMT solvers) may be an attractive choice; and in others, satisfactory execution on (mechanically generated) test cases may be preferred; yet others may require an independently checkable proof. Many verification components can be supported by an independent "reference" component that lacks the performance of the primary component, but that is much simpler, and may be deemed trustworthy—either because it has been formally verified, or by virtue of other V&V processes. It is conceivable that a high performance component can generate "hints" to boost the performance of a reference component without compromising the soundness of the latter's guarantee. For example, a reference SAT solver might lack the heuristics of a high-performance solver and could take days on problems that the high-performance component solves in minutes. But the high-performance solver could generate evidence that includes (or can be mined for) a list of which variables to split on first. Such a hint could vastly boost the performance of the reference solver. There is likely to be a range of possibilities for reference components that differ in performance and assurance, and in the hints that they can use productively.

Although most components and scripts will chain on the assertions contained in judgments, others could chain on the evidence to automate strategies such as diversity or the guidance of trusted reference components. In "exploration" mode, we may prefer the performance of state of the art components, and switch to the reference solver only for the final "certification" runs. In this mode, chaining on evidence could be used to mine hints from high-performance components for consumption by reference components. We can imagine scripts that automate this process to provide high quality evidence with acceptable performance. The final product will be a chain of argument that tracks the provenance of every assumption and claim (rather like the "proofchain analyzer" of PVS), together with the supporting evidence that can be used in a safety case or other certification process.

5. Conclusions

Technological innovations are sweeping through the field of formal verification. These changes are disruptive to approaches based on interactive theorem proving, which needs to find ways to exploit the capabilities of new technologies. A plausible, but inadequate response, uses the new technologies as backend solvers. It is inadequate because the new technologies often do not fit the paradigm of theorem proving and return results (e.g., counterexamples, unsatisfiable cores, lists of predicates) that are of little use to traditional proof procedures.

I have described two different and, I believe, more successful responses to disruptive innovation. The first, exemplified by SMT solvers, contributes to the disruption by extracting a core capability—namely, decision procedures—from theorem proving and combining it with a recent innovation—namely, fast SAT solving—to yield an innovation that is disruptive to both model checking (where it enables construction of bounded model checkers for infinite state systems), and theorem proving (where k-induction automates many proofs that previously required interactive guidance).

Fast SAT solvers proved disruptive in areas beyond those where SAT solving was traditionally employed: for example, the best methods for certain kinds of planning and scheduling problems now use SAT solvers. I speculate that SMT solvers will similarly prove disruptive beyond their current application in verification (again, planning is a candidate, where SMT solvers could extend SAT-based methods with the ability to handle metric and temporal constraints).

The second response that I described, the Evidential Tool Bus (ETB), is more speculative and currently only a proposal. However, we expect to start constructing a prototype very soon and to make it available as an open source project sometime in 2007. ETB-ready components available with the prototype will be extracted from our own existing tools, and we hope to solicit early collaborators who will provide novel components for the same release.

We plan use the ETB to construct an analyzer for hybrid systems based on hybrid abstraction [50], and to support verification with components that generate invariants by abstract interpretation and predicate abstraction. We will validate the ETB by applying it to interesting problems, including some proposed for the repository of the Verified Software Grand Challenge [48, 33, 31].

I hope others find the ETB an attractive proposal and will consider using it when available and will adapt their tools to provide components that can be attached to it. Although most instances of the ETB will operate within a single institution, it is conceivable that a worldwide verification re-

source could be constructed in this way and that it could provide a forum for extensive collaboration.

If fully successful, the approaches described here could have a disruptive impact beyond the technologies used for formal verification: the real targets for disruption are traditional methods for software development, validation, and certification.

Acknowledgments. The proposal for an evidential tool bus was developed with N. Shankar and discussions with our colleagues Leonardo de Moura, Bruno Dutertre, Josh Levy, Sam Owre, Rachid Rebiha, Hassen Saïdi, and Ashish Tiwari. An earlier version appears as [15].

References

[1] R. Alur and D. Peled, editors. *Computer-Aided Verification, CAV '2004*, volume 3114 of *Lecture Notes in Computer Science*, Boston, MA, July 2004.

[2] N. Amla, X. Du, A. Kuehlmann, R. P. Kurshan, and K. L. McMillan. An analysis of SAT-based model checking techniques in an industrial environment. In D. Borrione and W. Paul, editors, *Correct Hardware Design and Verification Methods: 13th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2005*, volume 3725 of *Lecture Notes in Computer Science*, pages 254–268, Saarbrücken, Germany, Oct. 2005.

[3] A. Armando and D. Zini. Towards interoperable mechanized reasoning systems: The logic broker architecture. In A. Corradi, A. Omicini, and A. Poggi, editors, *First AI*IA-TABOO Joint Workshop from Objects to Agents: Evolutive Trends of Software Systems*, pages 70–75, Parma, Italy, May 2000. Pitagora Editrice Bologna.

[4] C. Barrett, L. de Moura, and A. Stump. SMT-COMP: Satisfiability modulo theories competition. In K. Etessami and S. K. Rajamani, editors, *Computer-Aided Verification, CAV '2005*, volume 3576 of *Lecture Notes in Computer Science*, pages 20–23, Edinburgh, Scotland, July 2005.

[5] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In K. Yi, editor, *Programming Languages and Systems: Third Asian Symposium, APLAS*, number 3780 in Lecture Notes in Computer Science, pages 52–68, Tsukuba, Japan, 2005.

[6] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004. Coq home page: http://coq.inria.fr/.

[7] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In W. R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207, Amsterdam, The Netherlands, Mar. 1999.

[8] A. Cheyer and D. Martin. The open agent architecture. *Autonomous Agents and Multi-Agent Systems*, 4(1–2):143–148, Mar. 2001.

[9] C. M. Christensen. *The Innovator's Dilemma*. Harvard Business School Press, 1997.

[10] C. M. Christensen. *The Innovator's Solution*. Harvard Business School Press, 2003.

[11] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In Emerson and Sistla [23], pages 154–169.

[12] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, Englewood Cliffs, NJ, 1986. Nuprl home page: http://www.cs.cornell.edu/Info/Projects/NuPRL/.

[13] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, Jan. 1977. Association for Computing Machinery.

[14] L. de Moura, S. Owre, H. Rueß, J. Rushby, and N. Shankar. The ICS decision procedures for embedded deduction. In D. Basin and M. Rusinowitch, editors, *2nd International Joint Conference on Automated Reasoning (IJCAR)*, volume 3097 of *Lecture Notes in Computer Science*, pages 218–222, Cork, Ireland, July 2004.

[15] L. de Moura, S. Owre, H. Rueß, J. Rushby, and N. Shankar. Integrating verification components. In Shankar [48].

[16] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In Alur and Peled [1], pages 496–500. SAL home page: http://sal.csl.sri.com/.

[17] L. de Moura and H. Rueß. An experimental evaluation of ground decision procedures. In Alur and Peled [1], pages 162–174.

[18] L. de Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. In A. Voronkov, editor, *18th International Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Computer Science*, pages 438–455, Copenhagen, Denmark, July 2002.

[19] L. de Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification. In W. A. Hunt, Jr. and F. Somenzi, editors, *Computer-Aided Verification, CAV '2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 14–26, Boulder, CO, July 2003.

[20] L. A. Dennis, G. Collins, M. Norrish, R. Boulton, K. Slind, G. Robinson, M. Gordon, and T. Melham. The PROSPER toolkit. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, volume 1785 of *Lecture Notes in Computer Science*, pages 78–92, Berlin, Germany, Mar. 2000.

[21] D. L. Dill. The Murϕ verification system. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393, New Brunswick, NJ, July/Aug. 1996.

[22] B. Dutertre and M. Sorea. Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In *Formal Techniques in Real-Time and Fault-*

Tolerant Systems, volume 3253 of *Lecture Notes in Computer Science*, Grenoble, France, Sept. 2004.

[23] E. A. Emerson and A. P. Sistla, editors. *Computer-Aided Verification, CAV '2000*, volume 1855 of *Lecture Notes in Computer Science*, Chicago, IL, July 2000.

[24] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, Feb. 2001. Daikon home page: http://pag.csail.mit.edu/daikon.

[25] A. Franke and M. Kohlhase. MATHWEB, an agent-based communication layer for distributed automated theorem proving. In *16th International Conference on Automated Deduction (CADE)*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 217–221, Trento, Italy, July 1999.

[26] D. Gelernter and N. Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–102, Feb. 1992.

[27] D. I. Good. *Toward a Man-Machine System for Proving Program Correctness*. PhD thesis, University of Wisconsin, 1970.

[28] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, UK, 1993. HOL home page: http://www.cl.cam.ac.uk/Research/HVG/HOL/.

[29] J. Harrison. HOL Light: A tutorial introduction. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269, Palo Alto, CA, Nov. 1996. HOL Light home page: http://www.cl.cam.ac.uk/~jrh13/hol-light/index.html.

[30] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. May 2003. BLAST home page: http://embedded.eecs.berkeley.edu/blast/.

[31] T. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.

[32] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003. SPIN home page: http://spinroot.com/.

[33] C. Jones, P. O'Hearn, and J. Woodcock. Verified software: A grand challenge. *IEEE Computer*, 39(4):93–95, Apr. 2006.

[34] M. Kaufmann and J. S. Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, Apr. 1997. ACL2 home page: http://www.cs.utexas.edu/users/moore/acl2/.

[35] J. C. King. *A Program Verifier*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1969.

[36] S. Lahiri, R. Nieuwenhuis, and A. Oliveras. SMT techniques for predicate abstraction. In *Computer-Aided Verification, CAV '2006*, Lecture Notes in Computer Science, Seattle, WA, 2006. To appear.

[37] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct. 1997. Uppaal home page: http://www.uppaal.com.

[38] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1993.

[39] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*, pages 530–535, Las Vegas, NV, June 2001.

[40] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002. Isabelle home page: http://isabelle.in.tum.de/.

[41] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus—an architecture for extensible distributed systems. In *Fourteenth ACM Symposium on Operating System Principles*, pages 58–68, Asheville, NC, Dec. 1993. (ACM Operating Systems Review, Vol. 27, No. 5).

[42] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, Feb. 1995. PVS home page: http://pvs.csl.sri.com.

[43] L. Pike and G. M. Brown. Easy parameterized verification of biphase mark and 8N1 decoders. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '06)*, number 3920 in Lecture Notes in Computer Science, pages 58–72, Vienna, Austria, Apr. 2006.

[44] J. Rushby. Verification diagrams revisited: Disjunctive invariants for easy verification. In Emerson and Sistla [23], pages 508–520.

[45] V. Rusu and E. Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. In W. R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 178–192, Amsterdam, The Netherlands, Mar. 1999.

[46] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Prog. Lang. Syst.*, 24(3):217–298, May 2002.

[47] H. Saïdi and S. Graf. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Computer-Aided Verification, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 1997.

[48] N. Shankar, editor. *IFIP Working Conference on Verified Software: Theories, Tools, and Experiments*, Zurich, Switzerland, Oct. 2005. Available at http://vstte.inf.ethz.ch/papers.html.

[49] M. Sheeran, S. Singh, , and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer-Aided Design (FMCAD 2000)*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125, Austin, TX, Nov. 2000.

[50] A. Tiwari. *Abstractions for Hybrid Systems*. Computer Science Laboratory, SRI International, Menlo Park, CA, 2004. Combines several conference papers: available at http://www.csl.sri.com/~tiwari/new.pdf.