SRI International

Technical Report SRI-CSL-03-05 ● November 2003
Updated January and October 2004

Formal Methods Roadmap: PVS, ICS, and SAL

Formal Methods and Dependable Systems Program
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

Abstract

Our laboratory began building tools to support formal methods in the 1970's and we expect to continue doing so for many years to come. Currently, our most widely used tool is the PVS verification system, but we also provide the ICS decision procedures and the SAL model checking toolkit. The latest versions of all these tools offer state-of-the-art capabilities and performance, and are freely available for noncommercial research purposes from `http://formalware.csl.sri.com`.

Some of our users have told us they are perplexed by the range of tools we offer and do not know which is best for their purposes; others have said they are worried that the arrival of our newer tools may signal a weakened commitment to the older ones; still others say they hesitate to adopt the newer tools because their initial incarnations offered only modest capabilities and they wondered where we plan to take them, while others have questions about our licensing terms.

This document attempts to answer these questions: we explain the role that each tool plays within our overall strategy, the synergies between them, and their current capabilities. We then describe the new capabilities that we are developing and using in-house and that will soon be made available in new releases of our tools; finally, we describe directions we expect to explore in future and our plans for existing and new tools.

January 2004 update: includes SAL 2.1 and the witness model checker.

October 2004 update: includes SAL 2.2 and 2.3, and previews SAL 2.4 with the new explicit state model checker.

Contents

1 Introduction: PVS, ICS and SAL

In this section, we outline the three tools that we provide for mechanized support of formal methods, and describe the role that each plays within our overall vision. Each of the tools has its own website where you will find full documentation and download instructions, and all of our tools and papers can be accessed through our top-level website at `http://formalware.csl.sri.com`.

1.1 PVS

PVS (see `http://pvs.csl.sri.com`) is a classical "verification system": that is, it provides a specification language in which mathematical theories and conjectures about them can be specified, together with an interactive theorem prover that is used to discharge the conjectures. Although the PVS specification language provides many constructs that are familiar and useful to computer scientists, such as records, datatypes, and tabular specification of cases, and although it generally is used to formulate and examine topics in computer science (e.g., algorithms, circuits, architectures, or languages), its specification language is just a particularly rich and expressive *logic* and it provides no built-in status or support for computational systems. In particular, there are no built-in notions of "state," "state variable," or "assignment." Such computational notions must be modeled explicitly (e.g., by encoding the semantics of state machines and their transition relations) or indirectly (e.g., as recursive functions). In this way PVS differs from SAL, which is specialized to the specification of computational systems and does have notions of state and state transition built in.

Because it is not committed to specific notions of computation, PVS can be used to model a wide variety of computational phenomena as well as purely mathematical and logical constructions—indeed, some people use it for teaching logic.

1.2 ICS

One of the distinctive features of PVS is its use of decision procedures for a combination of theories including linear arithmetic over integers and reals and equality with uninterpreted functions. For some kinds of formal analysis, such as consistency and completeness checking in tabular specifications, these decision procedures provide all the automation that is required and several groups use PVS just to gain access to this capability. However, PVS is a rather monolithic system and it is not particularly easy to access its components, nor to interface other tools to it. Accordingly, we developed ICS (the name stands for "Integrated Canonizer/Solver") to satisfy the need for "embedded deduction."

ICS (see `http://ics.csl.sri.com`) decides a superset of the theories decided by PVS and does so with much higher performance and based on more recent theoretical insights. Furthermore, ICS integrates a SAT solver with the decision procedures in a revolutionary new way and can decide propositionally complex formulas over decided terms. ICS routinely decides formulas having hundreds of thousands of variables and constraints in a few seconds and also is fully competitive as a pure SAT solver and as a pure decision procedure (i.e., for conjunctions of terms). ICS is distributed as a C library and can be embedded using a small amount of "glue logic" into applications written in almost any language. There is also a simple interactive text interface to ICS for those who wish to experiment with its capabilities in this way. ICS provides the deductive core for bounded model checking and automated k-induction in the SAL toolkit and PVS will soon offer the option of substituting ICS for its "legacy" decision procedures.

1.3 SAL

It may be possible to apply much more specialized and automated methods of analysis and deduction if something is known about the class of problems being considered. So whereas a PVS theory could be about anything, SAL (see http://sal.csl.sri.com) is dedicated to analysis of computational systems specified as transition relations—and it provides a toolkit of powerful methods specialized to that application. Also, whereas PVS is a monolithic and somewhat closed system, SAL (the name stands for Symbolic Analysis Laboratory) is an open environment to which it is easy to attach new components or to interface other systems.

The core of SAL is its language, which is designed for the modular specification of (nondeterministic) state machines by means of transition relations. Unlike the languages that provide front ends to model checkers, SAL does not restrict specifications to finite state constructions, and instead provides a rich type system similar to that of PVS. The SAL language is defined as an XML DTD and is intended to serve as a common intermediate representation for many concrete languages and analysis tools. Some tools examine a SAL specification and deliver an analysis (e.g., a model checker may produce a counterexample), others operate as SAL-to-SAL transformers (e.g., a slicer), while others may perform more complex transformations (e.g., an abstractor may take a SAL specification for an infinite state system and a SAL description of an invariant for that system discovered by some other tool, and return the SAL specification for a finite-state abstraction that can then be analyzed by a model checker).

The current collection of SAL tools includes a translator from a textual syntax into the XML representation, a lightweight well-formedness checker, a deadlock checker, and five model checkers: one performs explicit state exploration, a second does symbolic model checking using BDDs, the third does bounded model checking using a SAT solver, the fourth does "infinite bounded" model checking (that is bounded model checking for systems defined over possibly infinite data such as integers) using decision procedures, while the fifth does "witness" model checking. There is also an interactive simulator that uses the capabilities of the symbolic model checker so that the user can say "take me forward to a state satisfying this property." The first four model checkers use linear temporal logic (LTL) as their assertion language, while the Witness model checker uses computation tree logic (CTL). The bounded model checker can use several different SAT solvers in addition to that of ICS, and the infinite bounded model checker can use several different decision procedures in addition to ICS. (We use this capability to benchmark the performance of ICS against its competitors.) Both the bounded model checkers are also able to perform proof by k-induction. All five model checkers deliver state-of-the-art performance and the capabilities of the infinite bounded and witness model checkers are unique. Soon to be released is a full typechecker for SAL. We expect that we and others will soon release translators into SAL from other popular modeling languages.

2 Evolution and Current Capabilities

The previous section has outlined the essential characteristics of our three tools: PVS for wide-spectrum specification and interactive theorem proving, SAL for automated analysis of state machines, and ICS for embedded deduction. ICS provides a deductive core for both PVS and SAL. In this section we describe more about the tools, their evolution, and their current state.

2.1 PVS

We began development of PVS in 1990 and publicly released it in April 1993 [ORSvH95]. The specification language of PVS is higher-order logic extended with predicate and dependent types, and a theory (cf. module) system. Its type constructors include functions, tuples, records, recursive datatypes (e.g., lists and trees) and enumerations (which are degenerate datatypes); sets are represented by their characteristic predicates. Functions (which include predicates and relations) may be defined constructively or axiomatically; the constructive mechanisms include simple, recursive, and (for relations) inductive definitions. Theories may be packaged in libraries. The built in "prelude" library includes 150 theories and 9 datatypes and provides 350 definitions, 76 axioms, and 760 proved theorems for concepts ranging from real and integer arithmetic, the ordinals up to ε_0, through various kinds of induction, to μ-calculus and computation tree logic (CTL). Libraries distributed with PVS provide finite sets and bitvectors (originally developed by NASA), while user-contributed libraries provide many additional theories. Particularly notable are 17 substantial libraries developed and maintained by NASA Langley Research

Center that formalize much foundational mathematics including analysis and graph theory.

The PVS theorem prover is interactive, based on a sequent calculus presentation. A graphical representation is available to help navigate large proof trees. Proofs can be saved as scripts and rerun either automatically, or in a single-step mode. While the basic proof commands are built-in, most are programmed as *strategies* in a Lisp-like language and can be augmented by users without risk of compromising soundness, which rests only on the basic commands. As well as the basic rules of logic, the built-in commands provide very powerful capabilities including decision procedures for ground (unquantified) integer and real linear arithmetic, automatic rewriting, and BDD-based propositional simplification and symbolic model checking. The power of its automation and, in particular, its use of decision procedures, is one of the characteristics that distinguished PVS from contemporaneous verification systems based on higher-order logics such as HOL and NuPRL, while the richness of its specification language and its support for interactive proof development distinguished it from other highly-automated systems such as ACL2. Indeed, one of the intellectual motivations behind PVS has been to explore synergies between an expressive logic and powerful proof automation.

PVS 2, which was released in 1995, added several capabilities to PVS but, more significantly, much of the implementation of the system was revised for this release. The reimplementation streamlined many internal data structures and algorithms for improved performance and ease of maintenance and enhancement. When considering the past and future evolution of our tools, it is important to appreciate that these are very large and complex software systems founded on sophisticated mathematics and that experience in their construction and use often stimulates new ideas and suggests better ways to do things—that is why this is a research activity. Unlike groups that have dissolved after just a few years, we have had the opportunity to learn from experience and to revise our ideas and to reimplement our software in the light of lessons we have learned and feedback from users.

PVS 2 also added a "batch" mode of operation; in conjunction with a large (and continuously growing) suite of benchmark specifications and proofs, this allows us to perform extensive validation and regression tests on new and modified versions of PVS. The validation suite includes all the major PVS libraries known to us and amounts to over 500 Mb of material.

Predicate subtypes are one of the most attractive features of the PVS specification language [ROS98]: they allow a predicate to be associated with a type (e.g., the type of even numbers is the subtype of the integers associated with the predicate "is divisible by two"). Predicate subtypes allow much of a specification to be embedded in its types, which contributes to clarity and economy in specification, and it also makes information available to the theorem prover in a manner that enhances automation. Some questions of type-correctness become undecidable in the presence of predicate subtypes, and PVS generates proof obligations called Type Correctness Conditions (TCCs) that must be discharged to ensure type-correctness. PVS 2 enriched support for predicate subtypes by adding *judgments*, which can inform the typechecker of certain typing relationships (e.g., that the sum of two even numbers is even) and generate a single TCC that subsumes many others, and *conversions*, which are the logical analog of casts in programming languages (e.g., indicating how a set of even numbers may be interpreted as a set of integers).

A major new capability added in PVS 2 (actually, in 2.3) was *evaluation* [Sha02b]. Many PVS constructions (e.g., simple and recursive definitions, and quantification over bounded types) can be evaluated by direct execution when applied to concrete values. The PVS evaluator provides this capability by compiling suitable PVS constructions into Lisp (the implementation environment used by PVS). The compilation performs extensive static analysis so that variables are updated in place (i.e., destructively) whenever it is safe to do so, thereby overcoming the performance penalty associated with most implementations of purely applicative languages: compiled PVS executes at about half the speed of unoptimized C. The evaluator can be used to explore specifications prior to undertaking proof, or in parallel with proof (so that, for example, execution of the PVS specification for a processor can replace the traditional hand-crafted simulator), and it allows PVS to be used as an implementation language. This avoids the difficulty traditionally associated with "code proof" due to the discontinuity at the interface between the purely logical specification and the imperative program, and it also opens the door to "computational reflection," whereby proof procedures formally verified in PVS can be used to augment the prover. The mechanisms that the evaluator uses to associate Lisp functions with PVS functions can also be used to provide *semantic attachments* that allow PVS to be used in rapid prototyping or to drive external systems (e.g., graphics) [COR$^+$01]. These capabilities have been significantly extended and exploited by researchers at NASA and NIA [Muñ03].

PVS was the first system to provide an integration of theorem proving and model checking [RSS95]: in 1995, a BDD-based decision procedure for μ-calculus over finite types was added to the basic proof commands of PVS, and the typechecker was augmented to keep track of finite types. The standard resources of PVS then extend this to CTL model checking. PVS 2 added automated *abstraction*, which uses theorem proving to calculate the image of a specification under a given abstraction function [SS99]. Typically, the target of the abstraction is finite state so that model checking may be used to discharge proof goals.

Similar to the way in which a decision procedure for finite μ-calculus is provided by an external BDD package, PVS 2.3 also adds a decision procedure for the very powerful logic called WS1S using the Mona library [OR00]. The external WS1S and μ-calculus decision procedures differ from those in the PVS core in that they are used only to discharge the leaf nodes of a proof: if they fail to discharge a proof goal, they do not return further goals from which the proof may proceed (though the WS1S procedure can return counterexamples).

PVS 3, which was released in December 2002, added theory interpretations to the PVS language [OS01]. These may be used to establish an "implementation" relationship between theories, and also to demonstrate consistency of axiomatically-specified theories. A related capability allows theories to be supplied as parameters to other theories. Co-inductive definitions and co-tuples were also added to this version of PVS, in anticipation of full support for co-algebraic constructions. PVS 3.2, to be released in early 2004, also allows the theorem prover to switch between its standard "legacy" decision procedures and those of ICS.

PVS has a large and complex implementation; the core system is implemented in Common Lisp while its interface uses Emacs and is written in Emacs Lisp. Graphical displays use Tcl/TK, while the theorem prover uses an external BDD and model checking package implemented in C and external decision procedures (including ICS) implemented in a variety of languages. PVS binary distributions are available for Sun Solaris and for RedHat Linux systems (it also seems to work for other Linux distributions and for FreeBSD and Debian, although we do not guarantee this).[1]

PVS is a mature and reliable system with many users; several of the modifications and enhancements made to PVS were in response to suggestions and feedback received from users. PVS is available from `http://pvs.csl.sri.com`; the current version is 3.1. Release or announcement notes describing the changes and enhancements made in each version and release (back to 2.3 in September 1999) can be found by following the `http://pvs.csl.sri.com/new.html` link. We are currently documenting its APIs to assist users who wish to interface PVS with other systems.

2.2 ICS

Decision procedures are valuable in interactive theorem provers because they raise the level of human interaction to one of giving strategic direction rather than managing tactical details. True decision procedures (as opposed to heuristics) are necessary to support a good mental model of the capabilities of the system, and to encourage perspicuous specifications (because specifications do not need to be distorted to fit the heuristics of the prover).

There are many decidable theories of importance in computer science applications (e.g., real and integer arithmetic, bitvectors, datatypes, equality with function symbols) and the goal is to decide the *combination* of these theories. This should be accomplished by combining decision procedures for the individual theories in a modular manner. Two general methods are known for combining decision procedures. The method of Nelson and Oppen imposes few restrictions on its component

[1]We have only RedHat systems locally; our distribution licenses for Allegro Common Lisp are for versions that support Solaris 2.7, 2.8 and 2.9 and Linux x86, Kernel 2.x, and glibc 2.1.

theories and their decision procedures, but yields relatively low performance because the separate decision procedures do not share much state and communicate only by propagating newly discovered equalities back and forth. The combination method of Shostak, on the other hand, requires that its component theories are *canonizable* and *solvable*, and achieves high performance by tightly integrating these components through an efficient data structure for congruence closure. Most theories of practical interest are canonizable and solvable.

The decision procedures in PVS employ Shostak's method and, indeed, are based on his original Lisp code from the early 1980s that was previously used in the STP and EHDM verification systems that preceded PVS. Over the years, we have discovered and corrected flaws in the theoretical basis of Shostak's method and in its implementation. These developments culminate in papers that give the definitive presentation of Shostak's corrected method and the proofs that it is sound, complete, and terminating [RS01,SR02,Sha02a]. The core argument has been formally verified in PVS [FS02].

ICS employs these theoretical insights and new implementations of the individual decision procedures to provide similar capabilities to the PVS decision procedures, but with far greater performance, in a standalone package suitable for embedded deduction. ICS Version 1 was released in June 2002 and built on experience gained with an unreleased prototype of the previous year [FORS01].

An experimental combination of ICS with the Chaff SAT solver produced very attractive performance for large propositional combinations of terms over the theories decided by ICS but it became clear that for really high performance the decision procedure and SAT solver would need to be more tightly integrated and each would need to be customized to operate in partnership with the other [dMR02]. Therefore, we developed our own SAT solver for this purpose and obtained a substantial increase in performance. The SAT solver was included in ICS Version 1.1 (released in March 2003), whose API was enlarged to give access to the new services.

Benchmarking indicates that a significant factor in the overall performance of a decision procedure/SAT solver combination is the quality of the "explain" function that the decision procedures use to generate constraints (we call them "lemmas on demand") that prune the search space explored by the SAT solver. Theoretical considerations suggest that the best explain function would use dependency information from the "proof objects" that decision procedures can generate to justify their decisions.

Availability of the early versions of ICS allowed us to explore "infinite bounded model checking" in SAL and to contemplate new applications such as test case generation. Both of these require the decision procedure to examine satisfiability in addition to validity/unsatisfiability and to return a satisfying instance for those formulas that are satisfiable. Accordingly, ICS was substantially redeveloped to yield ICS Version 2, which uses dependency information in explanations and provides instances for satisfiable formulas. The constituent decision procedures were also reworked, particularly the real arithmetic procedure, which uses a new variation on the Simplex algorithm and is of extremely high performance. New theoretical insights relax the requirement for solvable theories in a Shostak integration and this allows some elements of nonlinear real arithmetic to be added in a principled way; ICS 2.0 provides only modest capability of this kind, but we plan to extend it in future releases.

The theories decided by ICS 2.0 comprise ground (i.e., unquantified) real and integer linear arithmetic with an incomplete extension to the nonlinear cases, equality with uninterpreted function symbols, products (i.e., tuples) and co-products (i.e., disjoint sums), propositional calculus and propositional sets, and restricted forms of lambda calculus, bitvectors, and arrays. The interface to ICS supports "online" construction and querying of a logical context: that is, formulas can be asserted and retracted in an incremental manner and checked for validity, satisfiability, or unsatisfiability. A model or satisfying assignment can be generated for valid and satisfiable contexts.

ICS 2.0 was released in November 2003. Its core decision procedures are implemented in Objective CAML while its SAT solver is implemented in C++; binary distributions are available for Linux (we develop it on RedHat 7.3), Sun Solaris (we develop it on Version 2.7) and Mac OS X. It also runs on Windows under Cygwin but we have not yet sorted out the licensing issues for that version. ICS appears as a C library so that it can be linked to applications written in a variety of languages; we also provide a simple text interface.

ICS is fast and scalable: on modest contexts of the kind that arise during interactive theorem proving (e.g., PVS subgoals), it decides several thousand theorems a second, and for large examples of the kind that arise during automated verification and computational biology, it routinely decides formulas whose representation is several megabytes in size in some tens of seconds. ICS has several competitors; these include SVC (`http://chicory.stanford.edu/SVC/`), CVC (`http://verify.stanford.edu/CVC/`), and CVC Lite (`http://verify.stanford.edu/CVCL/`), all from Stanford, and UCLID (`http://www-2.cs.cmu.edu/~uclid/`) from CMU. The Stanford systems have similar architectures to ICS (i.e., they integrate separate decision procedures for their constituent theories), but UCLID is completely different in that it encodes all its theories into propositional calculus and uses off-the-shelf SAT solvers for the final search. ICS decides more theories and has a richer interface than these other systems (in particular, it supports incremental assertion and retraction) and on our benchmarks it is usually faster—often much faster. A comprehensive benchmarking of these decision procedures was published in 2004 [dMR04] (the benchmarks, translators, and detailed results are available at `http://www.csl.sri.com/users/demoura/gdp-benchmarks.html`), and a competition has been introduced by the Conference on Automated Verification (see `http://www.csl.sri.com/users/demoura/smt-comp/`). As a standalone SAT solver, ICS is comparable in performance to zChaff (when the time taken to clausify is taken into account).

2.3 SAL

Powerful new methods for automated analysis and verification have emerged since PVS was first developed: symbolic and bounded model checking, predicate abstraction (whose inventors used PVS for the first implementation [SG97]), invariant generation, and so on. These methods are specialized to specifications of certain kinds of computational systems—typically state machines expressed as transition relations. It is perfectly feasible to incorporate some of these methods as general proof procedures in PVS (as is done for model checking [RSS95] and predicate abstraction [SS99]), but more automation is possible when the specification is known to describe a state machine in a particular way. Again, it is perfectly feasible to define a stereotypical style for specifying state machines within PVS and to develop tools that recognize such specifications and provide special-purpose automation for them: InVeSt [BLO98] does this.

However, we considered that there are advantages to defining a language specifically for state machines; we also wished to explore a style of analysis that combines the capabilities of many different tools within an architecture that is more open and modular than PVS. The idea behind the SAL architecture is that many tools can cooperate in an analysis by iteratively transforming or enriching a common representation of the specification under examination and its related artifacts: an abstractor, for example, may transform a state machine into a simpler one that preserves certain properties, while a reachability analyzer may enrich the state machine with a description of its reachable states, and a model checker may construct a representation of a counterexample that a test case generator may later transform into a concrete test case. The core of SAL, therefore, is its language for representing state machines and their properties and related artifacts. The requirements we set for this language were that it should be sufficiently rich that it can express different styles of description and thereby serve as an intermediate representation for specifications originally written in other languages, sufficiently attractive that people will want to write in it directly, and sufficiently parsimonious that analysis tools are not burdened by unnecessary linguistic complexities.

The core SAL language supports the specification of transition relations (i.e., nondeterministic state machines), either as guarded commands or as (SMV-style) invariant definitions, or a combination of the two. SAL transition relations are packaged in parameterized modules that can be composed both synchronously and asynchronously. Unlike most other state machine notations, SAL is not restricted to finite data types: it supports reals, integers, and subtypes of these—in fact, it supports a full type system similar to that of PVS. As in PVS, certain subtype applications may generate proof obligations (TCCs); however, those TCCs that concern state variables need only be shown to be invariants, rather than statically valid. Also, unlike other similar notations, SAL does not rely on simple static rules (e.g., orderings on the state variables) to exclude combinational loops but generates TCCs for this purpose. These and other attributes of the SAL language provide flexibility and richness of expression beyond that available in other state machine languages and make it fairly easy to translate *into* SAL, but correspondingly rather difficult to translate *out* of SAL into more impoverished languages.

We originally intended to analyze SAL specifications using model checking tools developed by others. However, our initial experiments [BGL+00] revealed that providing a symbolic model checker for full SAL (restricted to finite state) via translation to SMV (for example) is difficult: the amount of translation required from guarded commands into SMV is such that one may as well extend the translation a little and invoke the basic machinery of symbolic model checking directly. The VIS symbolic model checker and its intermediate language BLIF-MV pose similar problems as targets for translation from SAL. No other well-supported symbolic model checkers were available at the time, and no bounded model checkers were publicly available at all. Furthermore, Murϕ, our preferred explicit-state reachability analyzer, which does provide guarded commands, had ceased to be supported. Moreover, we realized that the decision-procedure/SAT solver integration in ICS could provide a new kind of model checking (namely, bounded model checking for infinite-state systems, which we call infinite bounded model checking), and that certain applications of model checking (e.g., test case generation) would be realized most powerfully if one could extend the model checker interface. For all these reasons, we decided to develop our own model checkers for SAL.

The current SAL toolset provides explicit state, symbolic, bounded, infinite bounded, and witness model checkers for SAL. Explicit state model checking (sometimes called "on the fly" model checking) is provided by SAL 1.0, which was released in July 2002. SAL 1.0 is not simply a model checker, but rather a generic environment for state space exploration: it provides a rich API that allows many different kinds of analysis to be programmed as simple scripts (in the Scheme programming language) over this API. For example, a script to model check safety properties by breadth first search is just 43 lines of Scheme. The main script supplied with SAL 1.0 performs Linear Temporal Logic (LTL) model checking by breadth-, depth, or iterative-deepening depth-first search, with or without hash compaction (cf. "supertrace" in Spin); others check for deadlocks, perform simulation, and draw a picture of the statespace. Users can modify the scripts or write their own to perform targeted search, or search-space reductions based on symmetry or partial orders. The underlying state space exploration engine is quite efficient, and the Büchi automata construction uses state of the art methods.

Explicit state model checkers receive less attention than symbolic and bounded model checkers but they do have certain advantages. First, their performance depends on the size of the statespace, not the transition relation: there are examples whose transition relation is so large that a symbolic model checker cannot even build the BDD that represents it, but whose state space (particularly under symmetry or other reductions) is quite small and can be completely explored by an explicit state model checker. Second, an explicit state model checker can always do *something*: even when a state space is huge or is highly complex (e.g., with linked list data structures), an explicit state model checker can perform complete exploration to some depth and will often find bugs that are not found by random simulation. The SAL explicit state model checker can explore even infinite state systems to a specified depth. Finally, it is easy to extract highly informative counterexamples to invalid properties from an explicit state model checker.

However, explicit state model checkers are overwhelmed by large state spaces: the practical limit is typically a few tens of millions of states. Symbolic and bounded model checking will sometimes succeed on problems that are beyond the reach of an explicit state model checker, and are often much faster even on problems that are within its scope. Symbolic and bounded model checking for SAL are provided by SAL 2.0, which was released in November 2003 (SAL 1.0 continues to be available, but all its capabilities will eventually be integrated in SAL 2.x). All these model checkers use LTL as their assertion language (via Büchi automata constructions), with optimizations for the common special cases. The SAL symbolic and bounded model checkers use common front end procedures that "compile" SAL down to flattened representations as BDDs or satisfiability problems. These procedures perform extensive optimizations including partial evaluation, common subexpression elimination, and cone of influence reduction (i.e., slicing).

The SAL symbolic model checker uses CUDD (see `http://vlsi.colorado.edu/~fabio/CUDD`) as its BDD package. This is the same package used by NuSMV (see `http://nusmv.irst.itc.it/`) and the performance of the SAL symbolic model checker is comparable to that of NuSMV.

Unless it runs out of resources, the SAL symbolic model checker will either verify a property or (for LTL formulas) produce a counterexample. The SAL bounded model checkers are specialized to finding counterexamples, but also are able to attempt verification of safety properties using k-induction [dMRS03]. The basic bounded model checker for SAL handles only finite state systems and uses propositional SAT solving. It is possible to select either GRASP (see `http://sat.inesc-id.pt/~jpms/grasp/`), zChaff (see `http://www.ee.princeton.edu/~chaff/zchaff.php`), or ICS as the SAT solver; ICS usually provides the best performance in this application. The infinite bounded model checker for SAL can handle infinite state systems defined over the reals, integers, or other theories interpreted by its underlying decision procedure [dMRS02]. It is possible to select either

SVC, CVC, CVC Lite, UCLID (see the links provided earlier), or ICS as the decision procedure; again, ICS usually provides the best performance in this application (and note that some of the other decision procedures do not support all the theories of ICS).

The modular construction of SAL 2.0 allows additional tools to be constructed quite readily. Other SAL tools built in this way and available in SAL 2.0 include a lightweight well-formedness checker (the full typechecker will be available soon), a slicer, a "pathfinder" that generates random paths by bounded model checking, and an interactive simulator based on the symbolic model checker. Unlike a traditional simulator, the SAL simulator has the power of the symbolic and bounded model checkers at its disposal and can be instructed, for example, to search all paths from its current state to one leading to a state in which some given property is true.

Experience with SAL 1.0 led to some improvements in the design of the SAL language that are incorporated in SAL 2.0.

SAL 2.1, which was released in January 2004, added the Witness Model Checker to the repertoire of SAL model-checking tools. Witness model checking [SS03]is a new approach that operates by explicitly constructing evidence in the form of witnesses (for true properties) and counterexamples (for false ones). The witness model checker works on full CTL, so that its evidence has the form of tree-like structures (possibly with loops). The version of the model checker released in SAL 2.1 prints representative paths for some properties; the next release will include a capability to browse paths interactively. The witness model checker operates symbolically using the infrastructure of the symbolic model checker. Although the witness model checker uses CTL and the other model checkers use LTL, the SAL tools automatically convert formulas in the intersection of these two languages into the form needed by the model checker concerned (thus, an LTL G property will be interpreted as CTL AG for the witness model checker, and vice-versa for the other model checkers). On formulas in this intersection, the witness model checker typically operates at about half the speed of the symbolic model checker.

SAL 2.2, which was released in April 2004, provided numerous small enhancements and bug fixes, and added the tool `ltl2buchi`, which draws a visual representation of the Büchi automaton corresponding to a named property in a SAL file.

SAL 2.3, which was released in July 2003, provided several performance enhancements for the symbolic and bounded model checkers. The symbolic model checker was updated to use the latest version of the CUDD BDDD package (CUDD 2.4.0), and several options were added to allow more control over dynamic reordering of the BDDs, and over the partitioning of the transition relation. Prioritized traversal [FKZ$^+$00] was also added as an option: this divides the frontier in the symbolic forward image computation into pieces and uses a variety of strategies to prioritize the order in which the image computation is pushed forward on the separate pieces.

Prioritized traversal is very effective for finding deep bugs in very large models. The symbolic model checker is also now able to examine all the properties in a file (or, optionally, just those that are invariants) in a single invocation. In the case of invariants, it first computes the reachable states (which may be expensive), and is then able to analyze each property in neglible time.

SAL 2.4, which will be released in late 2004 (but is already available for alpha testing—contact us if you would like access to a pre-release copy) provides a high-performace explicit state model checker. This is not only much faster than the explicit state model checker of SAL 1.0, but it also operates on the full SAL 2 language and provides optimizations for symmetry reduction. The symbolic model checker has additional strategies for prioritized traversal, and the user can provide Scheme predicates to control BDD variable reordering. The perfomance of the bounded model checker has been significantly improved, and it is now able to use the `siege` (see `http://www.cs.sfu.ca/~loryan/personal`) and `BerkMin` (see `http://eigold.tripod.com/BerkMin.html`) SAT solvers.

2.4 Availability and Licensing

Binary distributions of our tools are freely available for noncommercial research purposes, but under a license that preserves our commercial opportunities. Source code is also available for noncommercial research purposes, but the licensing conditions are a little more strict. Please visit `http://formalware.csl.sri.com` for download instructions. To discuss commercial evaluation or licensing opportunities, please contact `Rushby@csl.sri.com`.

We are committed to the continued development and support of our tools, and to their free availability for noncommercial research purposes. However, SRI is a not-for-profit research institute with neither an endowment nor philanthropic benefactors; all of our research is funded through grants, contracts, and licenses. The climate for research funding has changed considerably in the past few years: many major research laboratories have closed, downsized, or undergone drastic changes of focus; several major tools for formal methods are no longer actively developed, others never got beyond their first release. We do not intend to share these fates. Fortunately, the technology of formal methods, as embodied in our tools, has now achieved sufficient power, automation, and speed that it is feasible to imagine it delivering value to real engineers—perhaps by generating test cases for Simulink/Stateflow or model checking a UML design. Our licensing terms are intended to ensure that we are able to develop commercial opportunities of these kinds that will help us stay in business. We hope that all our users see the benefits to this and therefore tolerate the minor inconvenience of clicking to acknowledge our licensing terms (for binary distributions) or signing an agreement (for source code). Note that users who have existing licenses do not need new ones and that the we have not changed the arrangements for PVS.

Where we consider it appropriate, we do release material in open source form and we host the QPQ repository for deductive software (see `http://www.qpq.org`). The PVS Prelude was one of the first items added to QPQ and we may later contribute more of the system.

3 Current Developments and Future Plans

The current releases of PVS (Version 3.1), ICS (version 2.0), and SAL (Version 2.1) provide very powerful capabilities. However, we continue to research new ways to provide effective support for formal methods and are developing several new capabilities that will be released in the next few months and years. In this section, we outline some of the research directions that we pursuing and the new tools and capabilities that we expect to release in the near future.

3.1 PVS

Our plans for PVS are to fully realize its existing capabilities by improving some features and enhancing its interfaces. We also plan some extensions to the PVS language and to the prover.

The PVS evaluator compiles a large fragment of PVS into executable Lisp code. The evaluator in current versions of PVS provides an effective proof of concept but has some limitations that are being removed by a reimplementation that will be included in a forthcoming release. The reimplemented evaluator can handle parameterized theories and performs more exact static analysis. The resulting code executes faster, is cognizant of its TCCs to ensure soundness, and can be used in proofs.

Several groups have integrated other tools with PVS (e.g., Maple [ADG+01]), but the process has been difficult and has often required our help. We are in the process of better documenting the interfaces to PVS that can be used by external tools, and the methods for using PVS as a backend to other tools. We will also describe how to add new primitive rules to the prover (i.e., how the WS1S and model-check rules are provided) and will document the API used by the PVS strategy language to give access to terms. We expect the revised evaluator and extended documentation to be released with a version of PVS in 2005.

In the longer term, we plan to provide an XML representation for specifications and proofs; PVS will be able to import and export these and the structured representation should make them easier for other tools to process. We also plan to specify the API used by the strategy language in PVS itself, then use the evaluation capability of PVS to allow strategies to be written as PVS functions over this API specification. We call this capability "PVS in PVS" and believe that it will greatly simplify the development of strategies and improve their documentation. It will also allow strategies to be typechecked and opens the door to computational reflection, whereby evaluation of verified PVS functions can be used to extend the prover in an assuredly sound manner.

Also in the longer term, we plan to extend the PVS language to allow polymorphism. This will be provided first at the declaration level, and later at the expression level. We also plan to add functors: in conjunction with co-tuples, which are already present, these will provide full support for co-algebras and co-datatypes. (We previously tried a different approach to co-datatypes and co-algebraic constructions but withdrew it after some experimentation in favor of this method).

These longer term plans are currently unfunded, but we hope it may be possible to undertake them within the next couple of years. We have also developed a theoretical basis for integrating structural subtyping (the kind used in object-oriented programming languages) with the predicate subtyping of PVS, but do not currently plan to implement this.

The current version of PVS can use ICS as its decision procedure but makes only modest use of its capabilities. During 2005, we plan to extend the proof commands in PVS to use the full capabilities of ICS (e.g., its integration with propositional SAT, and its ability to produce proof objects and counterexamples). As ICS is itself extended (e.g., with quantifier elimination), its new capabilities will also made available in PVS. We have also developed a very effective decision procedure for nonlinear real arithmetic [Tiw03a] that is used in our work on hybrid systems. We plan to make this available in PVS, either directly, or by incorporating its capabilities in ICS.

3.2 ICS

Our benchmarks indicate that ICS has the highest performance of decision procedures in its class [dMR04]; it also has the richest interface and decides the largest collection of theories. We plan to further increase the speed of ICS, the number and character of the theories that it decides, and the range of its capabilities.

ICS integrates classical decision procedures with a SAT solver; its current performance derives in large part from optimizing each component to cooperate with the other. Our measurements indicate that further performance improvements can be achieved with even tighter integration. For example, one of the ways in which SAT solvers achieve high performance is by using the unit clause rule to propagate constraints, but this is hampered if constraints known to the decision procedure are made available to the SAT solver only infrequently. We have had some success with an adaptive strategy that changes the frequency of interaction as search times increase, but for the longer term we intend to explore ways for making propagation more efficient by increasing the information shared between the decision procedure and SAT solver.

The overall performance of ICS is very sensitive to the quality of the "explanations" (of unsatisfiability) that the decision procedures return to the SAT solver. The explanations currently generated contain redundancies that reduce their effectiveness. It is possible to prune the explanations after they are generated but this incurs its own cost and pays off only on large examples. Again, we have had some success with an adaptive strategy that prunes when certain thresholds are exceeded, but for the longer term we are exploring new ideas for the direct generation of parsimonious explanations. We will also consider adding the capability to generate proof objects.

For bounded integer arithmetic, we have found that the SAT solver operating on a bitvector encoding is more efficient than the arithmetic decision procedures. This is in part due to improved constraint propagation as mentioned above, and in part due to the fact that case analysis is often required—and this is performed by the SAT solver. We intend to generalize the SAT solver to a finite domain solver; when the decision procedure discovers that a variable is constrained to a finite range, so responsibility for managing its constraints will shift to the finite domain solver.

In model checking, we desire a counterexample when a property violation is detected. ICS has the ability to produce satisfying instances, and the SAL infinite bounded model checker uses this to provide counterexamples. To provide the most information, these counterexamples are expressed in terms of constraints. However, for applications such as metric planning and test case generation it is necessary to construct concrete instances from the constraints. The data structures of ICS make it possible to extract the least or the greatest (and hence a middle) value satisfying given sets of constraints and we intend to extend ICS and SAL so that it is easy to produce such concrete satisfying instances. We will extend the XML representation used in SAL to incorporate additional artifacts such as counterexamples and test cases.

The implementation of ICS is designed for research and development convenience rather than raw performance: the SAT solver is written in C++ while the decision procedure is in OCaml; as the optimum algorithmic strategies become clear, we will move to a more efficient implementation. As we develop its algorithms and implementation and continue to benchmark its performance, we are confident that ICS performance will steadily improve and remain well ahead of its competitors.

The SAL infinite bounded model checker can use a number of decision procedures and provides a convenient way to benchmark their performance on this type of application. We intend to develop representative examples from SAL and a variety of other application areas and to cooperate with other groups in establishing reliable performance benchmarks for decision procedures, similar to what the SAT community has done.

In addition to increasing its speed, we intend to expand the number of theories that ICS decides. Planned and possible extensions include several arithmetic theories, such as interval arithmetic, complex numbers, nonlinear arithmetic, modular arithmetic, solvable Diophantine equations, and continued fractions. There are a couple of research challenges here. First, several of these theories are not disjoint (i.e., they share constants and function symbols) so the standard methods for combining decision procedures do not apply. Second, some of these theories (e.g., nonlinear arithmetic) have very high complexity and are usually implemented in an incomplete manner. This is not a problem in interactive applications (e.g., in PVS it just means that some true subgoals are not discharged automatically and require interactive guidance) nor in applications such as automated abstraction (where incompleteness results only in the abstraction being coarser than necessary), but for infinite bounded model checking in SAL, for example, it can result in invalid counterexamples, which may be unacceptable to users. Experimentation is needed to discover how best to manage incomplete decision procedures in different application contexts.

Possible extensions to nonarithmetic theories include general datatypes (currently only functors built up from product and co-product are supported), finite fields, and decidable or partially decidable fragments of graph theory, pointers, quaternions, and elliptic curves. We may also contemplate incorporation of user-provided decision procedures, or synthesis of decision procedures for user-provided theories. Some of these might not be Shostak theories, so the method of combining decision procedures may be extended to incorporate the method of Nelson and Oppen.

ICS decides ground, that is to say unquantified, theories. An extension to quantified theories will be highly attractive for many applications. Unfortunately, while the combination of decidable ground theories is decidable, this may not be so for quantified theories: for example, the combination of quantified integer linear arithmetic (Presburger arithmetic) with equality over uninterpreted function symbols is undecidable. However, although the full combination is undecidable, the circumstances that trigger incompleteness are sharply defined and quite rare in practice. We experimented with a prototype quantifier elimination procedure in previous versions of PVS and plan to build on this experience to add quantifier elimination to ICS. As mentioned earlier, incomplete procedures are adequate for some applications but problematic to others and experimentation is needed to find the best approach.

ICS provides deductive capability that can be embedded in other applications (such as compilers or planners). Many of these applications will employ full first or higher order logic, and will have some method for defining functions and supplying axioms or lemmas. Although we do not intend to extend ICS into a complete prover, we do plan to provide capabilities that reduce the amount of "glue logic" that application developers will need to write to adapt ICS to their domain. These include Skolemization, rewriting (which can automate expansion of definitions and application of lemmas), and proof search. The matching used in rewriting and unification used in proof search will both be extended to operate modulo the decided theories of ICS.

3.3 SAL

The most urgently required addition to the SAL toolkit is a full typechecker. The lightweight typechecker that is currently available operates like the typechecker of a programming language: it checks that functions and operators are applied to arguments of the correct types, but it does not perform the deeper checks needed to ensure soundness (some of which require generation and proof of TCCs). We expect to release the full SAL typechecker in early 2005; it is implemented in a modified version of the PVS Lisp environment and uses many well-tried and tested mechanisms from PVS.

The full typechecker will generate auxiliary SAL files containing TCCs that must be discharged to ensure full type correctness of its parent context. There must be some mechanism that keeps track of the relationship between parent contexts and their TCCs, checks that they are current, and monitors the status of proofs (rather like the Proofchain Analyzer in PVS). We anticipate adding additional tools to SAL that will generate other auxiliary files containing abstractions, invariants, analysis results, and so on. We refer to the mechanism that manages the association between these different tools and the various files and types of information that pass between them as the SAL *Tool Bus*. Tools attach to the tool bus by publishing the types of the information they consume and produce and the logical relations between these. The latter have the character of proof rules. For example, if a SAL specification is associated with an "ok" output from the typechecker and all its TCCs are associated with "proved" outputs from a proof tool, then the specification is considered "fully typechecked." The tool bus performs *evidence management* using these proof rules (e.g., backward chaining can be used to bring an analysis up to date when an underlying file is changed).

We expect to release a preliminary version of the tool bus at the same time as the full typechecker in early 2005, but we also expect its capabilities to evolve considerably in later releases and that the concepts of tool bus and of evidence management will provide powerful organizing principles.

Once the tool bus is available, we plan to provide a tool for predicate and data abstraction based on that in PVS [SS99] and one for invariant generation and strengthening based on fixed point calculations [TRSS01] and another based on model checking methods [BGL+00]. We will also provide a translator from SAL to PVS. We expect these tools to become available during 2005.

We have developed a very powerful method for analysis of hybrid systems by automated construction of conservative discrete abstractions [TK02, Tiw03b]. This method has successfully analyzed systems that are significantly larger (have more continuous variables) than other methods [GTT03]. The systems are specified in an extension to SAL that we call "HybridSAL." We plan to make this extension to SAL and its associated method of *hybrid abstraction* available in 2005. We will also make available a similar extension for timed systems [Sor02]. As well as additions to the SAL language, we will also identify sublanguages and provide tools that are specialized to these. For example, we believe it is feasible to construct a translator to NuSMV for the SAL sublanguage that excludes guarded commands.

We encourage others to develop translators to SAL from their preferred input notation. The Veritech project at The Technion (see http://www.cs.technion.ac.il/Labs/ssdl/research/veritech/) has developed translators from several notations into their own core language and we anticipate a translator from that into SAL. We ourselves are developing a translator from the Simulink/Stateflow notation of Matlab into SAL; we have a formal semantics for a well-behaved subset of Stateflow and a translator from that subset to SAL. We anticipate enlarging the translation to include a subset of Simulink once HybridSAL is available.

In addition to analysis of hybrid systems specified in Simulink/Stateflow, we are interested in automated generation of test cases for them. It is well known that unit test cases can be constructed from the counterexamples produced by model checkers. Infinite bounded model checking is ideal for this (because it handles arithmetic constraints properly and is specialized to the generation of counterexamples). The infinite bounded model checker generates symbolic counterexamples and the translation of these into concrete test cases can be accomplished elegantly through the interaction of several tools attached to the SAL Tool Bus. We believe it is possible to generate more efficient test *sets* by augmenting the interface to the SAL infinite bounded model checker so that it generates "tours" rather than single traces.

Test generation above the unit level (e.g., integration tests) requires synthesis of a reactive test *program* rather than construction of test inputs, while hardware-in-the-loop tests require consideration of the plant as a hybrid system. We plan to approach these problems though construction of a general tool for controller synthesis in SAL.

Although our tools are among the most powerful available, the raw problems for which analysis is desired are often too large to be processed directly: hardware designs may have thousands of latches where a few hundred is the limit for model checking, and hybrid systems may have hundreds of continuous variables where a few tens is the limit for hybrid abstraction. We regard the problem of model *extraction* as one of the most interesting and challenging facing practical application of formal methods and we plan to develop effective methods for doing this based on various kinds of slicing and automated assume-guarantee reasoning. In hybrid systems, for example, a particular property may depend strongly on only a few variables and all others may be assumed (and later guaranteed) to be bounded in value or change of value.

4 Conclusion

We hope this overview of our three tools, PVS, ICS, and SAL, clarifies the relationships and synergies between them. Each tool serves a different purpose: PVS is for interactive theorem proving over a wide range of application domains; SAL is for automated analysis of computational systems specified as transition relations; and ICS is for embedded deduction. However, there are strong links between them: ICS provides a deductive core for both PVS and SAL; PVS and SAL share similar type systems and a translator from SAL to PVS will soon be forthcoming so that a single analysis can draw on the capabilities of both tools. The forthcoming SAL Tool Bus will provide an open environment in which to manage large analyses that call on the capabilities of many tools.

All three tools represent the state of the art in their respective fields. PVS provides a massive range of capabilities and an environment for interactive theorem proving that most users find highly productive. SAL provides an attractive modeling language and is alone among model checking environments in supporting all four kinds of model checker (explicit, symbolic, bounded, and infinite-bounded). Its symbolic and bounded model checkers provide state-of-the-art performance, while its infinite bounded model checker is unique. Among decision procedures for embedded deduction, ICS has the richest interface, decides the largest class of theories, and delivers the highest performance.

We are committed to the maintenance and continued development of all three tools. The model checkers of SAL will soon be augmented by tools for predicate, data, and hybrid abstraction, and the performance and capabilities of ICS will continue to grow.

The longer term evolution of our tools will be guided by our experience in using them and by feedback from you, our users. We hope this document encourages, perhaps even excites, current and new users, and inspires you to experiment with some of our newer tools and with the newer capabilities of older ones. We are always interested to hear of your experiences, good and bad, with our tools and to receive feedback and suggestions for improving them.

References

[ADG$^+$01] Andrew Adams, Martin Dunstan, Hanne Gottliebsen, Tom Kelsey, Ursula Martin, and Sam Owre. Computer algebra meets automated theorem proving: Integrating Maple and PVS. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 27–42, Edinburgh, Scotland, September 2001. Springer-Verlag. 14

[BGL$^+$00] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, June 2000. NASA Langley Research Center. Proceedings available at `http://shemesh.larc.nasa.gov/fm/Lfm2000/Proc/`. 10, 18

[BLO98] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. InVeSt: A tool for the verification of invariants. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-Aided Verification, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, pages 505–510, Vancouver, Canada, June 1998. Springer-Verlag. 9

[COR$^+$01] Judy Crow, Sam Owre, John Rushby, N. Shankar, and Dave Stringer-Calvert. Evaluating, testing, and animating PVS specifications. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, March 2001. Available from `http://www.csl.sri.com/users/rushby/abstracts/attachments`. 5

[dMR02] Leonardo de Moura and Harald Rueß. Lemmas on demand for satisfiability solvers. Presented at SAT 2002, accepted for journal publication, May 2002. Available at `http://www.csl.sri.com/users/demoura/sat02_journal.pdf`. 7

[dMR04] Leonardo de Moura and Harald Rueß. An experimental evaluation of ground decision procedures. In Rajeev Alur and Doron Peled, editors, *Computer-Aided Verification, CAV '2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 162–174, Boston, MA, July 2004. Springer-Verlag. 8, 15

[dMRS02] Leonardo de Moura, Harald Rueß, and Maria 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. Springer-Verlag. 11

[dMRS03] Leonardo de Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: From refutation to verification. In Warren A. Hunt, Jr. and Fabio Somenzi, editors, *Computer-Aided Verification, CAV '2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 14–26, Boulder, CO, July 2003. Springer-Verlag. 11

[ES00] 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. Springer-Verlag. 22, 23

[FKZ$^+$00] Ranan Fraer, Gila Kamhi, Barukh Ziv, Moshe Y. Vardi, and Limor Fix. Prioritized traversal: Efficient reachability analysis for verification and falsification. In Emerson and Sistla [ES00], pages 389–402. 12

[FORS01] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated Canonization and Solving. In G. Berry, H. Comon, and A. Finkel, editors, *Computer-Aided Verification, CAV '2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 246–249, Paris, France, July 2001. Springer-Verlag. 7

[FS02] Jonathan Ford and Natarajan Shankar. Verifying Shostak. In A. Voronkov, editor, *18th International Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Computer Science*, pages 347–362, Copenhagen, Denmark, July 2002. Springer-Verlag. 7

[GTT03] Ronojoy Ghosh, Ashish Tiwari, and Claire Tomlin. Automated symbolic reachability analysis, with application to delta-notch signaling automata. In Maler and Pnueli [MP03], pages 233–248. 19

[HP99] Nicolas Halbwachs and Doron Peled, editors. *Computer-Aided Verification, CAV '99*, volume 1633 of *Lecture Notes in Computer Science*, Trento, Italy, July 1999. Springer-Verlag. 24

[MP03] O. Maler and A. Pnueli, editors. *Hybrid Systems: Computation and Control, 6th International Workshop, HSCC 2003*, volume 2623 of *Lecture Notes in Computer Science*, Prague, Czech Republic, April 2003. Springer-Verlag. 22, 24

[Muñ03] César Muñoz. *Rapid Prototyping in PVS*. National Institute of Aerospace, Hampton, VA, 2003. Available from http://research.nianet.org/~munoz/PVSio/. 5

[OR00] Sam Owre and Harald Rueß. Integrating WS1S with PVS. In Emerson and Sistla [ES00], pages 548–551. 5

[ORSvH95] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995. 3

[OS01] Sam Owre and N. Shankar. Theory interpretations in PVS. Technical Report SRI-CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park, CA, April 2001. 6

[ROS98] John Rushby, Sam Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, September 1998. 4

[RS01] Harald Rueß and Natarajan Shankar. Deconstructing Shostak. In *16th Annual IEEE Symposium on Logic in Computer Science*, pages 19–28, Boston, MA, July 2001. IEEE Computer Society. 7

[RSS95] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, June 1995. Springer-Verlag. 5, 9

[SG97] Hassen Saïdi and Susanne Graf. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer-Aided Verification, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 1997. Springer-Verlag. 9

[Sha02a] Natarajan Shankar. Little engines of proof. In Lars-Henrik Eriksson and Peter Lindsay, editors, *Formal Methods Europe (FME'02)*, volume 2391 of *Lecture Notes in Computer Science*, pages 1–20, Copenhagen, Denmark, July 2002. Springer-Verlag. 7

[Sha02b] Natarajan Shankar. Static analysis for safe destructive updates in a functional language. In A. Pettorossi, editor, *11th International Workshop on Logic-based Program Synthesis and Transformation (LOPSTR 01)*, volume 2372 of *Lecture Notes in Computer Science*, pages 1–24, Paphos, Cyprus, November 2002. Springer-Verlag. 5

[Sor02] Maria Sorea. Bounded model checking for timed automata. *Electronic Notes in Theoretical Computer Science*, 68(5), 2002. Available at `http://www.elsevier.com/locate/entcs/volume68.html`. 19

[SR02] Natarajan Shankar and Harald Rueß. Combining Shostak theories. In Sophie Tison, editor, *International Conference on Rewriting Techniques and Applications (RTA '02)*, volume 2378 of *Lecture Notes in Computer Science*, pages 1–18, Copenhagen, Denmark, July 2002. Springer-Verlag. 7

[SS99] Hassen Saïdi and N. Shankar. Abstract and model check while you prove. In Halbwachs and Peled [HP99], pages 443–454. 5, 9, 18

[SS03] N. Shankar and Maria Sorea. Counterexample-driven model checking. Technical Report SRI-CSL-03-04, Computer Science Laboratory, SRI International, Menlo Park, CA, October 2003. Available at http://www.csl.sri.com/users/sorea/reports/wmc.ps.gz. 12

[Tiw03a] Ashish Tiwari. Abstraction based theorem proving: An example from the theory of Reals. In Silvio Ranise and Cesare Tinelli, editors, *Proceedings of the CADE-19 Workshop on Pragmatics of Decision Procedures in Automated Deduction, PDPAR 2003*, pages 40–52, Miami, FL, July 2003. The full proceedings are available at http://www.cs.miami.edu/~geoff/CADE-19/W2.pdf; this paper is also available at http://www.csl.sri.com/users/tiwari/pdpar03.html. 15

[Tiw03b] Ashish Tiwari. Approximate reachability for linear systems. In Maler and Pnueli [MP03], pages 514–525. 19

[TK02] Ashish Tiwari and Gaurav Khanna. Series of abstractions for hybrid automata. In C.J. Tomlin and M.R. Greenstreet, editors, *Hybrid Systems: Computation and Control, 5th International Workshop, HSCC 2002*, volume 2289 of *Lecture Notes in Computer Science*, pages 465–478, Stanford, CA, March 2002. Springer-Verlag. 19

[TRSS01] Ashish Tiwari, Harald Rueß, Hassen Saïdi, and N. Shankar. A technique for invariant generation. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001*, volume 2031 of *Lecture Notes in Computer Science*, pages 113–127, Genova, Italy, April 2001. Springer-Verlag. 18