Tool description to be presented at CAV 2004.

SAL 2*

Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, N. Shankar,
Maria Sorea, Ashish Tiwari

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025 USA

Abstract. SAL 2 augments the specification language and explicit-state model checker of SAL 1 with high-performance symbolic and bounded model checkers, and with novel *infinite bounded* and *witness* model checkers. The bounded model checker can use several different SAT solvers, while the infinite bounded model checker similarly can use several different ground decision procedures. SAL 2 provides a scriptable API for its basic model checking and analysis functions that can be used to extend the system. All four new model checkers are implemented using this interface.
Its high-level specification language and wide range of model checkers make SAL convenient for those seeking a ready-to-use solution, while its scriptability and flexible choice of backend analyzers should make it attractive to those seeking an experimental platform.

1 Introduction

SAL (see sal.csl.sri.com) is a suite of tools for analysis of state machines that constitutes part of our vision for a **S**ymbolic **A**nalysis **L**aboratory that will eventually encompass SAL, ICS, PVS, and other tools developed in our group and elsewhere.

SAL provides a language similar to that of our PVS verification system, but specialized for the specification of state machines; it was first released with an explicit-state model checker as SAL 1 in July 2002. SAL 2, which was released in December 2003 and updated to SAL 2.1 in January 2004, adds high-performance symbolic and bounded model checkers, and novel *infinite bounded* and *witness* model checkers. Both the bounded model checkers can additionally perform verification by k-induction, and the capabilities of all the model checkers and their components are available through an API that is scriptable in Scheme.

2 The Language

The SAL language was originally conceived as an intermediate language and was developed in collaboration with the research groups of David Dill at Stanford

* This work was partially supported by the DARPA SEC program through USAF Rome Laboratory contract F33615-00-C-3043, by NASA Langley Research Center through contract NAS1-00079, and by SRI International.

and Tom Henzinger at UC Berkeley. Since then, our version of the language has evolved, principally through the addition of a richer type system, including structured types and subtypes, so that it is now a comprehensive specification language in its own right.

SAL's type system and expression language are similar to those of PVS, including predicate subtypes, datatypes, infinite types such as reals and integers, recursive function definitions, and quantification. State machines are specified as parameterized modules with state variables explicitly identified as input, output, local, or global. The transition relation of a module may be specified using both guarded commands and SMV-style variable-wise invariants. Primes are used to indicate the values of variables in the new state and may appear on the right as well as left hand sides of assignments. Modules may be composed both synchronously and asynchronously (and in combinations of these) to yield systems; a renaming construction allows inputs and outputs of different modules to be "wired up" appropriately.

The assertion language is not primitive in SAL but is defined in libraries associated with the analyzer concerned. Three of the model checkers that constitute the analyzers in SAL 2.1 provide LTL as their assertion language, while the witness model checker supports CTL.

As befits its role as an intermediate language, SAL is defined as an XML DTD. Parsers and prettyprinters are provided for a human-readable ASCII representation, and for a Lisp-like LSAL syntax that is useful in scripting and is translated directly into internal representations by the Scheme scripting interface. Because the language is so rich, it is easy to translate most other state machine languages into SAL. We have a translator from the Stateflow notation of Matlab/Simulink [1], and we expect that ourselves and others will soon provide translators from other popular languages.

3 Preprocessing and Compilation

Because SAL is a rich language, compiling it into the representations used in the deductive cores of its analysis tools (e.g., as BDDs, or as propositional or ICS SAT problems) is a substantial task. All the SAL analysis tools share a common set of preprocessing and compilation routines that perform extensive optimizations. These include partial evaluation, common subexpression elimination, and slicing (i.e., cone of influence reduction). For the finite-state model checkers, arithmetic values and operators are compiled into bitvectors and binary "circuits" respectively, with comparable representations for other SAL types. Reverse translations allow counterexamples to be presented to the user as traces through the original SAL specification with variable assignments expressed in their original SAL types. LTL assertions are translated to optimized Büchi automata. Most of these transformations and optimizations can be controlled by the user.

SAL 2 provides a lightweight typechecker, called the SAL well-formedness checker, that 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 for some of the richer constructions: these require proof obligations similar to TCCs in PVS (although SAL TCCs need merely be invariants, rather than universally valid as in PVS) and will be supported by the full SAL typechecker, which is based on that of PVS.

4 Model Checkers

SAL 2 provides high performance symbolic and bounded model checkers (SMC and BMC, respectively) for systems defined over finite state types, and a novel "infinite bounded" model checker (infBMC) that can handle infinite as well as finite state types; SAL 2.1 added the "witness" model checker (WMC) that performs finite-state CTL model checking using a new symbolic method.

The SMC and WMC symbolic model checkers use the CUDD BDD package (see http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html) and provide extensive options for controlling the ordering and dynamic reordering of variables. The representation of the transition relation as a BDD and the evaluation of the transformed assertion use many optimizations and deliver performance comparable to other state-of-the-art symbolic model checkers, most of which start from much more primitive notations. In a case study concerning a fault-tolerant distributed algorithm [2], we have routinely analyzed systems with many hundreds of state bits and hundreds of billions of reachable states in tens of minutes using commodity workstations.

The WMC model checker implements a novel approach that constructs both symbolic witnesses (positive) and counterexamples (negative) for assertions in full CTL [3]. This symbolic evidence is useful in abstraction-refinement, vacuity checking and controller synthesis, and also allows explicit (trace or tree-like) witnesses and counterexamples to be easily extracted.

The BMC model checker uses a propositional SAT solver to search for counterexamples no longer than some specified "depth" (i.e., length); the model checker can be instructed to advance the depth incrementally, so that it will find the shortest counterexample, and it can also verify properties by k-induction (optionally using other formulas as lemmas). By default, SAL uses ICS as its SAT solver, but it can optionally be instructed to use zChaff [4] or GRASP [5]. In our case study mentioned above, the SAL bounded model checker would often solve problems having hundreds of thousands of nodes in their SAT representation (and more than 600 variables in a BDD representation) in a few minutes.

The infBMC model checker uses the standard formulation of bounded model checking, but instead of translating into a purely propositional SAT problem, it translates to the theory supported by the ICS decision procedure [6]. ICS is a decision procedure and satisfiability solver for the combination of ground (i.e., unquantified) real and integer linear arithmetic, 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. Using real or unbounded integer state types, SAL can represent infinite state systems such as hybrid or timed automata,

and other formulations of continuous or real-time behavior. An n-process version of Fischer's real time mutual exclusion algorithm, for example, is easily represented in SAL. Like its finite counterpart, the infBMC model checker can advance the depth of its search incrementally and can perform k-induction [7]. Instances of Fischer's algorithm, for example, can be verified using the infBMC model checker to perform 1-induction on a series of lemmas. Although the infBMC model checker uses ICS as its default satisfiability procedure, it can also be instructed to use UCLID [8], SVC [9], CVC [10], or CVC-Lite, though with restrictions (since UCLID decides less theories than ICS) and without counterexamples.

5 Scripting and the SAL Simulator

The preprocessing and model checking components of SAL can be accessed through an API defined in Scheme. The actual model checkers are simply Scheme scripts defined over this API. Users can write their own scripts to perform specialized analyses using the full resources of SAL. The SAL Simulator provides a convenient environment in which to develop such scripts: it is essentially a read-eval-print loop with the SAL libraries preloaded. Used as a simulator, it allows users interactively to explore a specification by executing selected transitions, filtering the current set of states, or finding a path to a state satisfying a given assertion. Used as an environment for developing scripts, all the capabilities described above can be used in Scheme functions defined by the user. For example, Gregoire Hamon has used this capability to develop a prototype test case generator for Stateflow that first uses symbolic model checking to find a path to some previously unvisited state or transition, then alternates slicing and bounded model checking to extend the path to additional unvisited targets.

6 Plans for Further Development

SAL 1, which is still available, provides an explicit-state model checker for a subset of the language supported in SAL 2. We intend to redevelop this model checker and to integrate it with the others in forthcoming versions of SAL. We will also integrate the extensions for specifying and abstracting hybrid systems developed by Ashish Tiwari [11].

Over the longer term, we intend to integrate SAL with PVS (so that, for suitable specifications, it will be possible to translate SAL into PVS, and vice-versa), and to evolve both into an open scriptable environment for symbolic analysis in which numerous tools, developed by ourselves and others, will interact through a SAL *Tool Bus*. The tool bus will extend the SAL language with XML representations for the many artifacts and intermediate products of analysis: for example, invariants, abstractions, counterexamples, test cases and their outputs.

7 Current Status and Availability

SAL 2.1 with all the capabilities described is freely available for noncommercial research purposes from `sal.csl.sri.com`. Binary versions of the system, which

require an automatically-generated license key, may be downloaded for Linux, Solaris, MacOS X, and Cygwin (for Windows). The SAL binaries also install the ICS executable. The SAL and ICS source code is available with a signed license agreement. The top-level page for tools developed by our group is fm.csl.sri.com, from which you can find links to our Roadmap, papers, examples, and a tutorial illustrating all our tools.

References

1. Hamon, G., Rushby, J.: An operational semantics for Stateflow. In Wermelinger, M., Margaria-Steffen, T., eds.: Fundamental Approaches to Software Engineering: 7th International Conference (FASE). Lecture Notes in Computer Science, Barcelona, Spain, Springer-Verlag (2004) 229–243 To appear; available at http://www.csl.sri.com/users/rushby/abstracts/fase04.
2. Steiner, W., Rushby, J., Sorea, M., Pfeifer, H.: Model checking a fault-tolerant startup algorithm: From design exploration to exhaustive fault simulation. To be presented at DSN '04 (2004) Available at http://www.csl.sri.com/~rushby/abstracts/startup-verification.
3. Shankar, N., Sorea, M.: Counterexample-driven model checking. Technical Report SRI-CSL-03-04, Computer Science Laboratory, SRI International, Menlo Park, CA (2003) Available at http://www.csl.sri.com/users/sorea/reports/wmc.ps.gz.
4. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proceedings of the 38th Design Automation Conference (DAC'01). (2001)
5. Marques-Silva, J.P., Sakallah, K.A.: GRASP - A New Search Algorithm for Satisfiability. In: Proceedings of IEEE/ACM International Conference on Computer-Aided Design. (1996) 220–227
6. de Moura, L., Rueß, H., Sorea, M.: Lazy theorem proving for bounded model checking over infinite domains. In Voronkov, A., ed.: International Conference on Automated Deduction (CADE'02). Volume 2392 of Lecture Notes in Computer Science., Copenhagen, Denmark, Springer-Verlag (2002) 438–455
7. de Moura, L., Rueß, H., Sorea, M.: Bounded model checking and induction: From refutation to verification. In Hunt, Jr., W.A., Somenzi, F., eds.: Computer-Aided Verification, CAV '2003. Volume 2725 of Lecture Notes in Computer Science., Boulder, CO, Springer-Verlag (2003) 14–26
8. Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In: Computer Aided Verification. Volume 2404 of Lecture Notes in Computer Science., Springer-Verlag (2002) 78–92
9. Barrett, C.W., Dill, D.L., Levitt, J.R.: A decision procedure for bit-vector arithmetic. In: Proceedings of the 35th Design Automation Conference. (1998) San Francisco, CA.
10. Stump, A., Barrett, C.W., Dill, D.L.: CVC: a cooperating validity checker. In: Computer Aided Verification. Volume 2404 of Lecture Notes in Computer Science., Springer-Verlag (2002) 500–504
11. Tiwari, A.: Approximate reachability for linear systems. In Maler, O., Pnueli, A., eds.: Hybrid Systems: Computation and Control, 6th International Workshop, HSCC 2003. Volume 2623 of Lecture Notes in Computer Science., Prague, Czech Republic, Springer-Verlag (2003) 514–525