SAL 2*

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

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

1 Introduction

SAL (see `http://sal.csl.sri.com`) is an open suite of tools for analysis of state machines; it constitutes part of our vision for a **S**ymbolic **A**nalysis **L**aboratory that will eventually encompass SAL, the PVS verification system, the ICS decision procedures, and other tools developed in our group and elsewhere.

SAL provides a language similar to that of PVS, 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, 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 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, in addition to its role as an intermediate language, SAL is now a comprehensive specification language in its own right.

SAL's type system and expression language are similar to those of PVS, including higher types, predicate subtypes, datatypes, infinite types such as reals and integers (and their function types), 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

* This work was partially supported by the DARPA and USAF Rome Laboratory contract F33615-00-C-3043, NASA Langley Research Center contract NAS1-00079, National Science Foundation grant CCR-ITR-0326540, and by SRI International.

in the new state and may appear in guards and in the right-hand sides of assignments and nondeterministic selections as well as on their left-hand sides. 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 provide LTL as their assertion language, while the witness model checker supports CTL. (Both notations can be used to specify formulas in their common subset and SAL translates automatically to the form required by the analyzer concerned).

To support its role as an intermediate language, SAL is defined in XML. 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, 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. Many 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 does not perform the deeper checks needed for some of SAL's richer constructs: these require proof obligations similar to TCCs in PVS (although SAL TCCs within modules need merely be invariants, not 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 (inf BMC) 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 and provide access to its 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 with Holger Pfeifer and Wilfried Steiner concerning fault-tolerant startup of the Time-Triggered Architecture (TTA), we 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. 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 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 or GRASP. In our TTA startup example, the SAL bounded model checker would often solve problems having hundreds of thousands of DAG nodes in their SAT representations (and more than 600 variables in a BDD representation) in a few minutes.

The inf BMC 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 ICS. Although ICS is competitive as a pure SAT solver, it is actually 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 coproducts (i.e., disjoint sums), propositional calculus and propositional sets, and restricted forms of lambda calculus, bitvectors, and arrays. Like its finite counterpart, the inf BMC model checker can advance its depth of search incrementally and can perform k-induction. Counterexamples are presented symbolically. Although inf BMC uses ICS as its default satisfiability procedure, it can also be instructed to use UCLID, SVC, CVC, or CVC-Lite, albeit with restrictions (e.g., UCLID decides less theories than ICS) and without counterexamples.

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. For example, with Bruno Dutertre, we have developed a timed formulation for TTA startup: instances with up to 10 nodes (whose representation uses 24 real and 99 discrete variables) can be verified in a few minutes using inf BMC to perform 1-induction on a series of lemmas. Instances of Fischer's real time mutual exclusion algorithm with as many as 39 nodes have been verified in the same way.

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 scripting, all the capabilities described above can be employed within user-written Scheme functions. For example, with Grégoire Hamon we have 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.

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 with all the capabilities described is freely available for noncommercial research purposes (i.e., roughly, research that will be openly published) from `http://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 `http://fm.csl.sri.com`, from which you can find links to our Roadmap, papers, examples, and a tutorial illustrating all our tools.

For want of space, all references have been omitted here; they are present in an expanded version pf the paper available at
`http://www.csl.sri.com/~rushby/abstracts/sal-cav04`.