Symbolic Analysis of Transition Systems*

Invited paper at the ASM2000 Workshop

Natarajan Shankar

Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
{shankar, owre}@csl.sri.com
URL: http://www.csl.sri.com/~shankar/
Phone: +1 (650) 859-5272 Fax: +1 (650) 859-2844

Abstract. We give a brief overview of the *Symbolic Analysis Laboratory* (SAL) project. SAL is a verification framework that is directed at analyzing properties of transition systems by combining tools for program analysis, model checking, and theorem proving. SAL is built around a small intermediate language that serves as a semantic representation for transition systems that can be used to drive the various analysis tools.

The transition system model of a program consists of a state type, an initialization predicate on this state type, and a binary next-state relation. The execution of a program starts in a state satisfying the initialization predicate so that each state and its successor state satisfy the next-state relation. Transition systems are a simple low-level model that have none of the semantic complications of high-level programming languages. Constructs such as branches, loops, and procedure calls can be modelled within a transition system through the use of explicit control variables. The transition system model forms the basis of several formalisms for several popular formalisms including UNITY [CM88], TLA [Lam94], SPL [MP92], and ASMs [Gur95]. It also underlies verification tools such as SMV [McM93], Murphi [Dil96], and STeP [MT96].

If we focus our attention on the verification of properties of transition systems, we find that even this simple model poses some serious challenges. The verification of transition systems is performed by showing that the system satisfies an invariance or progress property, or that it refines another transition system. It is easy to write out proof rules for the verification of such properties but the actual application of these proof rules requires considerable human ingenuity. For example, the verification of invariance properties requires that the invariant be inductive, i.e., preserved by each transition. A valid invariant might need to be strengthened before it can be shown to be inductive. Fairness constraints and progress measures have to be employed for demonstrating progress

* This work was funded by the Defence Advanced Research Projects Agency under Contract NO. F30603-96-C-0204, and NSF Grants No. CCR-9712383 and CCR-9509931. The SAL project is a combined effort between SRI International, Stanford University, and the University of California, Berkeley.

properties. It takes a fair amount of effort and ingenuity to come up with suitable invariant strengthenings and progress measures.

Methods like model checking [CGP99] that are based on state-space exploration have the advantage that they are largely automatic and seldom require the fine-grain interaction seen with deductive methods. Since these methods typically explore the reachable state space (i.e., the strongest invariant), there is no need for invariant strengthening. Progress measures are also irrelevant since the size of the whole state space is bounded. However, model checking methods apply only to a limited class of systems that possess small, essentially finite state spaces.

Theorem proving or model checking are not by themselves adequate for effective verification. It is necessary to combine the expressiveness of the deductive methods with the automation given by model checking. This way, small, finite-state systems can be directly verified using model checking. For larger, possibly infinite-state systems, theorem proving can be used to construct *property-preserving abstractions* over a smaller state space. Such abstractions convert data-specific characteristics of a computation into control-specific ones. The finite-state model constructed by means of abstraction can be analyzed using model checking. It is easy to actually compute the properties of a system from a finite-state approximation and map these properties back to the original system.

We give an overview of an ongoing effort aimed at constructing a general framework for the integration of theorem proving, model checking, and program analysis. We use the term *symbolic analysis* to refer to the integration of these analysis techniques since they all employ representations based on symbolic logic to carry out a symbolic interpretation of program behavior. The framework also emphasizes *analysis*, i.e., the extraction of a large number of useful properties, over *correctness* which is the demonstration of a small number of important properties. The framework is called the *Symbolic Analysis Laboratory* (SAL). We motivate the need for symbolic analysis and describe the architecture and intermediate language of SAL.

1 A Motivating Example

We use a very simple and artificial example to illustrate how symbolic analysis can bring about a synergistic combination of theorem proving, model checking, and program analysis. The example consists of a transition system with a state contain a (control) variable PC ranging over the scalar type $\{inc, dec\}$, and two integer variables B and C. Initially, control is in state inc and the variables B and C are set to zero. There are three transition rules shown below as guarded commands:

1. When $PC = inc$, then B is incremented by two, C is set to zero, and control is transferred to state dec.

$$PC = inc \longrightarrow B' = B + 2; C' = 0; PC' = dec;$$

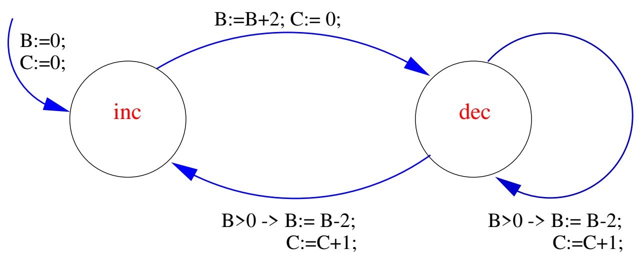

Fig. 1. A Simple Transition System: Twos

2. When $PC = dec$, B is decremented by two, and C is incremented by one, and control is transferred to state $decl$.

$$PC = dec \wedge B > 0 \longrightarrow B' = B - 2; C' = C + 1; PC' = inc;$$

3. Same as transition rule 2, but control stays in dec.

$$PC = dec \wedge B > 0 \longrightarrow B' = B - 2; C' = C + 1;$$

There is also an implicit stuttering transition from state dec to itself when none of the guards of the other transitions holds, i.e., when $B \leq 0$. Since the inc state has a transition with a guard that is always true, there is no need for a stuttering transition on inc. The transition system is shown diagrammatically in Figure 1.

The transition system $Twos$ satisfies a number of interesting invariants.

1. B is always an even number.
2. B and C are always non-negative.
3. B is always either 0 or 2.
4. B is always 0 in state inc.
5. C is always either 0 or 1.
6. In state dec, $B = 2$ iff $C = 0$.

The purpose of symbolic analysis is to find and validate such properties with a high degree of automation and minimal human guidance and intervention. While efficient automation is essential for analyzing large transition systems, the intended outcome of symbolic analysis is human insight. The analysis should therefore not rule out human interaction.

2 Some Symbolic Analysis Techniques

We enumerate some symbolic analysis techniques and assess their utility on the $Twos$ example. For this purpose, we focus on the invariant (1) below.

$$B = 0 \vee B = 2 \tag{1}$$

Note that the transition system *Twos* is a potentially infinite state system since variables B and C range over the integers.

Some mathematical preliminaries are in order. A transition system P is given by a pair $\langle I_P, N_P \rangle$ consisting of an initialization predicate on states I_P, and a binary next-state relation on states N_P. We constrain the next-state relation N to be total so that $\forall s : \exists s' : N(s, s')$. The metavariables s, s' range over states. We treat a set of states as equivalent to its characteristic predicate. The boolean connectives $\wedge$, $\vee$, $\supset$, are lifted from the booleans to the level of predicates and correspond to the set-theoretic operations $\cap$, $\cup$, and $\subseteq$, respectively. An assertion is a predicate on states. The metavariables ϕ, ψ range over assertions. A predicate transformer is a map from predicates to predicates. A monotone predicate transformer τ preserves the subset or implication ordering on predicates so that if $\phi \supset \psi$, then $\tau(\phi) \supset \tau(\psi)$. The fixed point of a monotone predicate transformer τ is an assertion ϕ such that $\phi \equiv \tau(\phi)$. As a consequence of the Tarski–Knaster theorem, every monotone predicate transformer has a least fixed point $lfp(\tau)$ and a greatest fixed point $gfp(\tau)$ such that

$$lfp(\tau) = \tau(lfp(\tau)) \supset gfp(\tau) = \tau(gfp(\tau)).$$

Let $\perp$ represent the empty set of states, $\top$ the set of all states, and ω the set of natural numbers. If the state space is finite, then the least fixed point $lfp(\tau)$ can be calculated as

$$\perp \vee \tau(\perp) \vee \tau^2(\perp) \vee \ldots \vee \tau^n(\perp)$$

for some n, and similarly, $gfp(\tau)$ can be calculated as

$$\top \wedge \tau(\top) \wedge \tau^2(\top) \wedge \ldots \wedge \tau^n(\top),$$

for some n.

If τ is $\vee$-continuous (i.e., $\tau(\bigvee_{i\in\omega} \phi_i) = \bigvee_{i\in\omega} \tau(\phi_i)$ for ϕ_i such that whenever $i < j$, $\phi_i \supset \phi_j$), then

$$lfp(\tau) = \bigvee_{i\in\omega} \tau^i(\perp) \tag{2}$$

Similarly, if τ is $\wedge$-continuous (i.e., $\tau(\bigwedge_{i\in\omega} \phi_i) = \bigwedge_{i\in\omega} \tau(\phi_i)$ for ϕ_i such that whenever $i < j$, $\phi_j \supset \phi_i$), then

$$gfp(\tau) = \bigwedge_{i\in\omega} \tau^i(\top) \tag{3}$$

Equations (2) and (3) provide an iterative way of computing the least and greatest fixed points but these on infinite-state spaces, the computations might not converge in a bounded number of steps.

Typical examples of monotone predicate transformers include

1. Strongest postcondition of a transition relation N, $sp(N)$, which is defined as
$$sp(N)(\phi) \equiv (\exists s' : \phi(s') \wedge N(s', s)).$$

2. Strongest postcondition of a transition system P, $sp(P)$ is defined as

$$sp(P)(\phi) \equiv I_P \vee sp(N_P)(\phi).$$

3. Weakest precondition of a transition relation N, $wp(N)$ is defined as

$$wp(N)(\phi) \equiv (\forall s' : N(s, s') \supset \phi(s)).$$

2.1 Invariant Proving

The invariance rule is the most heavily used proof rule in any program logic [Hoa69,Pnu77]. Given a transition system P as a pair $\langle I_P, N_P \rangle$, consisting of an initialization I_P and a next-state relation N_P, the invariance rule usually has the form:

$$\frac{\begin{array}{c}\vdash \psi(s) \supset \phi(s) \\ \vdash I_P(s) \supset \psi(s) \\ \vdash \psi(s_0) \wedge N_P(s_0, s_1) \supset \psi(s_1)\end{array}}{P \models \textbf{invariant } \phi}$$

In this rule, the assertion ψ is a strengthening of the assertion ϕ. Such a strengthening is needed since the assertion ϕ may not be inductive, i.e., satisfy the premises $\vdash I_P(s) \supset \phi(s)$ and $\vdash \phi(s_0) \wedge N_P(s_0, s_1) \supset \phi(s_1)$.

In the *Twos* example, the invariant (1) is not inductive. It fails because it is not preserved by transition 1 since we cannot establish

$$\begin{array}{ll}\vdash & (PC = inc \wedge (B = 0 \vee B = 2)) \\ & \wedge\ (PC = inc \wedge B' = B + 2 \wedge C' = 0 \wedge PC' = dec) \\ & \supset (B' = 0 \vee B' = 2).\end{array}$$

The invariant has to be strengthened with the observation that when $PC = inc$, B is always 0 so that it now reads

$$B = 0 \vee (PC \neq inc \wedge B = 2). \tag{4}$$

The strengthened invariant (4) is inductive. The need for invariant strengthening in program proofs is the key disadvantage of the deductive methods with respect to model checking. Quite a lot of effort is needed to turn a putative invariant into an inductive one. Once an invariant has been strengthened in this manner, it can contain a large number of conjuncts that generate a case explosion in the proof. Much of the focus of symbolic analysis is on supplementing deductive verification with the means of automatically obtaining useful invariants and invariant strengthenings.

2.2 Enumerative Model Checking

The early approaches to model checking were based on the feasibility of computing fixed point properties for finite-state systems. The reachable states of a finite-states can be computed by starting from the set of initial states and exploring the states reachable in n consecutive transitions. Any property that holds on all the reachable states is a valid invariant. There are many variations on this basic theme. Many modern enumerative model checkers such as Murphi [Dil96] and SPIN [Hol91] carry out a depth-first search exploration of the transition graph while maintaing a hash-table to record states that have already been visited. In SPIN, the LTL model checking problem is transformed into one of emptiness for ω-automata, i.e., automata that recognize infinite strings [VW86,GPVW95].

In enumerative model checking, properties written in a branching-time temporal logic CTL can be verified in time proportional to $N \times F$ where N is the size of the transition graph and F the size of the temporal formula. Model checking linear-time temporal logic formulas is more expensive and takes time proportional to $N \times 2^F$ where N is the size of the model and F is of the formula.

The *Twos* example succumbs rather fortuitously to enumerative model checking. Even though the potential state space of *Twos* is unbounded, only a bounded part of the state space is reachable since B is either 0 or 2, and C is either 0 or 1. The success of enumerative model checking is somewhat anomalous since this method is unlikely to terminate on typical infinite-state systems. Even on finite-state systems, an enumerative check is unlikely to succeed because the size of the searchable state space can be exponential in the size of the program state. Still, enumerative model checking is an effective debugging technique that can often detect and display simple counterexamples when a property fails.

2.3 Symbolic Model Checking

The use of symbolic representation for the state sets was proposed in order to combat the state explosion problem in enumerative model checking [BCM+92,McM93]. A symbolic representation for boolean functions based on binary decision diagrams (BDDs) [Bry86] has proved particularly successful. A finite state can be represented as a bit-vector. Then sets of bit-vectors are just boolean functions and can be represented as BDDs. In particular, the initial set, a given invariant claim, the transition relation, and the reachable state set, can all be represented as BDDs. The BDD operations can be used to compute images of state sets with respect to the transition relation. This allows predicate transformers such as strongest postcondition and weakest precondition to be applied to the BDD representation of a state set. The reachable state set can be computed by means of a fixed point iteration of the strongest postcondition computation starting from the initial state set. Every intermediate iteration of the reachable state set is also represented as a BDD. There are several advantages to the use of BDDs. Sometimes even sets of large cardinality might have compact symbolic representations. BDDs are a canonical representation for boolean functions so that equivalence tests are cheap. BDDs are especially good at handling the boolean quantification that is needed in the image computations. Automata-theoretic methods can also be represented in symbolic form. Some symbolic model checkers include SMV [McM93]

Such symbolic representations do require the state to be explicitly finite. This means that the *Twos* example cannot be coded directly in a form that can be directly understood by a symbolic model checker. Some work has to be done in order to reduce the problem to finite-state form so that it can be handled by a symbolic model checker.

2.4 Invariant Generation

Automatic invariant generation techniques have been studied since the 1970s [CH78,GW75,KM76,SI77], and more recently in the work of Bjørner, Browne, and Manna [BBM97], and Bensalem, Lakhnech, and Saïdi [BLS96,Saï96,BL99].

As in model checking, the basic operation in invariant generation is that of taking the strongest postcondition or weakest precondition of a state set X with respect to the transition relation N. Some of the techniques for computing invariants are described briefly below.

Least Fixed Point of the Strongest Postcondition. The invariant computed here corresponds to the reachability state set. It is computed by starting with an initial symbolic representation of the initial state set given by the program. This set is successively enlarged by taking its image under the strongest postcondition operation until a fixed point is reached, i.e., no new elements are added to the set. We term this method LFP-SP. It yields a symbolic representation of the set of reachable states which is the strongest invariant. However, LFP-SP computation often does not terminate since the computation might not converge to a fixed point in a finite number of steps. Take, for example, a program that successively increments by one, a variable x that is initially zero. This program has a least fixed point, i.e., x is in the set of natural numbers, but the iterative computation does not converge.

For the *Twos* example, the LFP-SP computation does terminate with the desired invariant as seen in the calculation below.

$$\begin{aligned}
\mathit{Inv}^0 &= (PC = \mathit{inc} \wedge B = 0 \wedge C = 0) \\
\mathit{Inv}^1 &= \mathit{Inv}^0 \vee (PC = \mathit{dec} \wedge B = 2 \wedge C = 0) \\
\mathit{Inv}^2 &= \mathit{Inv}^1 \vee (B = 0 \wedge C = 1) \\
\mathit{Inv}^3 &= (B = 0 \wedge C = 1) \vee (PC = \mathit{dec} \wedge B = 2 \wedge C = 0) \\
&= \mathit{Inv}^2
\end{aligned}$$

The resulting invariant easily implies the strengthened inductive invariant (4). The LFP-SP computation terminates precisely because the reachable state set is bounded. In more typical examples, approximation techniques based on *widening* will be needed to accelerate the convergence of the least fixed point computation.

Greatest Fixed Point of the Strongest Postcondition. The greatest fixed point iteration starts with the entire state space and strengthens it in each iteration by excluding states that are definitely unreachable. This approach, which we call GFP-SP, yields a weaker invariant than the least fixed point computation. The GFP-SP computation also need not terminate. Even when it does terminate, the resulting invariant might not be strong enough. In the case of the program with single integer variable x that is initially zero and incremented by one in each transition, the GFP-SP computation returns the trivial invariant **true**. However the GFP-SP method has the advantage that it can be made to converge more easily than the LFP-SP method, and any intermediate step in the computation already yields a valid invariant.

The greatest fixed point invariant computation for *Twos* (ignoring the variable C) can be carried out as follows. Here $Inv^i(pc)$ represents the i iteration of the invariant for control state pc.

$$\begin{aligned}
Inv^0(inc) &= (B = 0 \vee B \geq -1) = (B \geq -1)\\
Inv^0(sub) &= \mathbf{true}\\[1em]
Inv^1(inc) &= (B \geq -1)\\
Inv^1(sub) &= (B \geq 1 \vee B \geq -1) = (B \geq -1)\\[1em]
Inv^2(inc) &= (B \geq -1)\\
Inv^2(sub) &= (B \geq -1)
\end{aligned}$$

The invariant $B \geq -1$ is not all that useful since this information contributes nothing to the invariants that we wish to establish. Still, the GFP-SP method is not without value. It is especially useful for propagating known invariants. For example, if we start the iteration with invariant (1), then we can use the GFP-SP method to deduce that the strengthened invariant (4).

Greatest Fixed Point of the Weakest Precondition. Both LFP-SP and GFP-SP compute inductive invariants that are valid, whereas the method GFP-WP takes a putative invariant and strengthens it in order to make it inductive. The computation starts with a putative invariant S, and successively applies the weakest precondition operation wp(P)(S) to it. If this computation terminates, then either the resulting assertion is a strengthening of the original invariant that is also inductive, or the given invariant is shown to be invalid.

With the *Twos* example, the weakest precondition with respect to the putative invariant (1) yields the strengthened invariant (4).

2.5 Abstract Interpretation.

Many of the invariant generation techniques are already examples of abstract interpretation which is a general framework for lifting program execution from the concrete domain of values to a more abstract domain of properties. Examples of abstract interpretation include sign analysis (positive, negative, or zero) of variables, interval analysis (computing bounds on the range of values a variable can take), live variable analysis (the value of a variable at a control point might be used in the computation to follow), among many others.

We can apply an interval analysis to the *Twos* example. Initially, the interval for B is $[0, 0]$ for $PC = inc$. This yields an interval of $[2, 2]$ for B when $PC = dec$. In the next step, we have an approximation of $[0, 2]$ for B when $PC = dec$, and $[0, 0]$ when $PC = inc$. The next round, we get an approximation of $[-1, 0]$ for the range of B when $PC = inc$, and $[0, 2]$ for the range of B when $PC = dec$. At this point the computation converges, but the results of the analysis are still too approximate and do not discharge the invariant (1).

2.6 Property Preserving Abstractions

Since model checking is unable to cope with systems with infinite or large state spaces, abstraction has been studied as a technique for reducing the state space [CGL94,LGS+95,SG97]. In data abstraction, a variable over an infinite or large type is reduced to one over a smaller type. The smaller type is essentially a quotient with respect to some equivalence relation of the larger type. For example, a variable ranging over the integers can be reduced to boolean form by considering only the parity (odd or even) of the numbers. *Predicate abstraction* is an extension of data abstraction that introduces boolean variables for predicates over a set of variables. For example, if x and y are two integer variables in a program, it is possible to abstract the program with respect to the predicates such as $x < y$, $x = y$. These variables are then replaced by boolean variables p and q such that p corresponds to the $x < y$ and q corresponds to $x = y$. Even though predicate abstraction introduces only boolean variables, it is possible to simulate a data abstraction of a variable to one of finite type by using a binary encoding of the finite type.

In general, an abstraction is given by means of a concretization map γ such that $\gamma(a)$ for an abstract variable a returns its concrete counterpart. In the case of the abstraction where $x < y$ is replaced by p and $x = y$ by q, $\gamma(a) = (x < y)$ and $\gamma(b) = (x = y)$. The more difficult direction is computing an abstraction $\alpha(C)$ given a concrete predicate C. The construction of α requires the use of theorem proving as described below.

There are also two ways of using abstractions in symbolic analysis. In one approach, the abstract reachability set [SG97,DDP99] is constructed by the following iteration

$$ARG(P)(s) = lfp(\alpha(I_P) \vee \alpha \circ sp(P) \circ \gamma).$$

We can then check if p is an invariant of P by verifying $\gamma(ARG(P)) \supset p$.

A second way of using abstraction is by actually constructing the abstracted version of the program and the property of interest [BLO98,CU98,SS99]. This can be more efficient since the program and property are usually smaller than the abstract reachability graph.

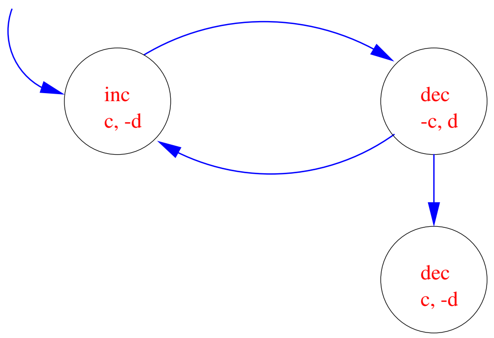

Fig. 2. Abstract *Twos*

In the *Twos* example, the predicate abstraction is suggested by the predicates $B = 0$ and $B = 2$ in the putative invariant. The abstract transition system by replacing the predicate $B = 0$ by c and $B = 2$ by d is shown in Figure 2.

The abstract transition system computed using predicate abstraction can easily be model checked to confirm that invariant (1) holds. The stronger invariant (4) can also be extracted from the reachable state space of the abstract transition system.

Predicate abstraction affords an effective integration of theorem proving and model checking where the former is used to construct a finite-state property-preserving abstraction that can be analyzed using the latter. The abstraction loses information so that a property can fail to hold in the abstract system even when its concrete counterpart is valid for the concrete system. In this case, the abstraction has to be refined by introducing further predicates for abstraction [BLO98,CGJ+00].

3 SAL: A Symbolic Analysis Laboratory

We have already seen a catalog of symbolic analysis techniques. The idea of a symbolic analysis laboratory is to allow these techniques to coexist so that the analysis of a transition system can be carried out by successive applications of a combination of these techniques [BGL+00]. With such a combination of analysis techniques, one could envisage a verification methodology where

1. A cone-of-influence reduction is used to discard irrelevant variables.
2. Invariant generation is used to obtain small but useful invariants.
3. These invariants are used to obtain a reasonably accurate abstraction to a finite-state transition system.
4. Model checking is used to compute useful invariants of the finite-state abstraction.
5. The invariants computed by model checking over the abstraction are used propagated using invariant generation techniques.

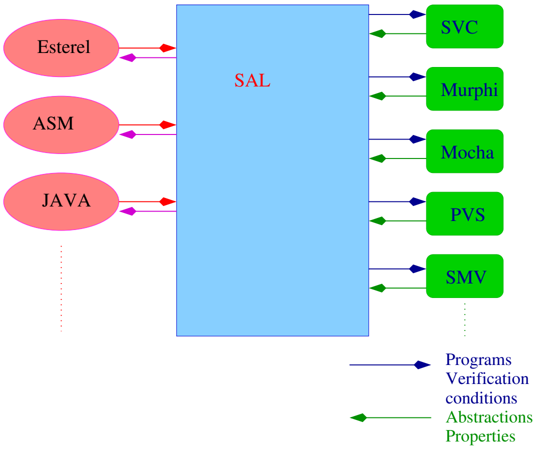

Fig. 3. The Architecture of SAL

6. This cycle can be repeated until no further useful information is forthcoming.

SAL provides a blackboard architecture for symbolic analysis where a collection of tools interact through a common intermediate language for transition systems. The individual analyzers (theorem provers, model checkers, static analyzers) are driven from this intermediate language and the analysis results are fed back to this intermediate level. In order to analyze systems that are written in a conventional source language, the transition system model of the source program has to be extracted and cast in the SAL intermediate language.[1] The model extracted in the SAL intermediate language essentially captures the transition system semantics of the original source program.

The SAL architecture is shown in Figure 3 The SAL architecture is constrained so that the different analysis tools do not communicate directly with each other, but do so through the SAL intermediate language. The interaction between the tools must therefore be at a coarse level of granularity, namely in terms of transition systems, their properties, and property-preserving transformations between transition systems. Allowing the tools to communicate directly to each other would require a quadratic number of different maps (for a given number of tools) between these analysis tools.

3.1 The SAL Intermediate Language

The intermediate language for SAL[2] serves as

[1] We are currently working on a translator from a subset of Verilog to SAL, and another from a subset of Java to SAL.

[2] The SAL intermediate language was designed in collaboration with Prof. David Dill of Stanford, Prof. Tom Henzinger at UC Berkeley, and several colleagues at SRI, Stanford, and UC Berkeley.

1. The target of translations from source languages.
2. The source for translations to the input formats of different analysis tools.
3. A medium for communication between different analysis tools.

The SAL intermediate language is based on languages and models such as SMV [McM93], Murphi [Dil96], Reactive Modules [AH99], ASM [Gur95], UNITY [CM88], and TLA [Lam94], among others. The unit of specification in SAL is a context which contains declarations of types, constants, transition system modules, and assertions. A SAL module is a transition system unit. A basic SAL module is a state transition system where the state consists of *input*, *output*, *local*, and *global* variables, where

- An input variable to a module can be read but not written by the module.
- An output variable to a module can be read and written by the module, and only read by an external module.
- A local variable to a module can be read and written by the module, but is not read or written by the module.
- A global variable to a module can be read and written by the module as well as by an external module

A basic module also specifies the initialization and transition steps. These can be given by a combination of definitions or guarded commands. A definition is of the form $x = \mathit{expression}$ or $x' = \mathit{expression}$, where x' refers to the new value of variable x in a transition. A definition can also be given as a selection of the form $x' \in \mathit{set}$ which means that the new value of x is nondeterministically selected from the value of of *set*. A guarded command is of the form $g \longrightarrow S$, where g is a boolean guard and S is a list of definitions of the form $x' = \mathit{expression}$ or $x' \in \mathit{set}$.

As in synchronous language such as Esterel [BG92] and Lustre [HCRP91], SAL allows *synchronous*, i.e., Mealy machine, interaction so that the new value of a local or output variable can be determined by the new value of a variable. Such interaction introduces the possibility of a causal cycle where each variable is defined to react synchronously to the other. Such causal cycles are ruled out by using static analysis to generate proof obligations demonstrating that such cycles are not reachable. The UNITY and ASM models do not admit such synchronous interaction since the new values of a variable in a transition are completely determined by the old values of the variables. SMV allows such interaction but the semantics is not clearly specified, particularly when causal cycles are possible. The Reactive Modules [AH99] language uses a static partial ordering on the variables that breaks causal loops by allowing synchronous interaction in one direction of the ordering but not the other. In TLA [Lam94], two modules are composed by conjoining their transition relations. TLA allows synchronous interaction where causal loops can be resolved in any manner that is compatible with the conjunction of the transition relations is satisfied.

SAL modules can be composed

- *Synchronously*, so that $M_1 \| M_2$ is a module that takes M_1 and M_2 transitions in lockstep, or

– Asynchronously, so that $M_1 \, [] \, M_2$ is a module that takes an interleaving of M_1 and M_2 transitions.

There are rules that govern the usage of variables within a composition. Two modules engaged in a composition must not share output variables and nor should the output variables of one module overlap with the global variables of another. The modules can can share input and global variables, and the input variables of one module can be the output or global variables of the other. Two modules that share a global variable cannot be composed *synchronously*, since this might create a conflict when both modules attempt to write the variable synchronously. The rules governing composition allow systems to be analyzed modularly so that system properties can be composed from module properties [AH99].

The N-fold synchronous and asynchronous compositions of modules are also expressible in SAL. Module operations include those for hiding and renaming of variables. Any module defined by means of composition and other module operations can always be written as a single basic module, but with a significant loss of succinctness.

SAL does not contain features other than the rudimentary ones described above. There are no constructs for synchronization, synchronous message passing, or dynamic process creation. These have to explicitly implemented by means of the transition system mechanisms available in SAL. While these features are useful, their introduction into the language would place a greater burden on the analysis tools.

The SAL language is thus similar in spirit to Abstract State Machines [Gur95] in that both serve as basic conceptual models for transition systems. However, machines described in SAL are not abstract compared with those in ASM notation since SAL is intended as a front-end to various popular model checking and program analysis tools.

4 Conclusions

Powerful automated verification technologies have become available in the form of model checkers for finite, timed, and hybrid systems, decision procedures, theorem provers, and static analyzers. Individually, these technologies are quite limited in the range of systems or properties they can handle with a high degree of automation. These technologies are complementary in the sense that one is powerful where the other is weak. Static analysis can derive properties by means of a syntactic analysis. Model checking is best suited for control-intensive systems. Theorem proving is most appropriate for verifying mathematical properties of the data domain. Symbolic analysis is aimed at achieving a synergistic integration of these analysis techniques. The unifying ideas are

1. The use of transition systems as a unifying model, and
2. Fixed point computations over symbolic representations as the unifying analysis scheme.

3. Abstraction as the key technique for reducing infinite-state systems to finite-state form.

Implementation work on the SAL framework is currently ongoing. The preliminary version of SAL consists of a parser, typechecker, causality checker, an invariant generator, translators from SAL to SMV and PVS, and some other tools. SAL is intended as an experimental framework for studying the ways in which different symbolic analysis techniques can be combined to achieve greater automation in the verification of transition systems.

Acknowledgments. Many collaborators and colleagues have contributed ideas and code to the SAL language and framework, including Saddek Bensalem, David Dill, Tom Henzinger, Luca de Alfaro, Vijay Ganesh, Yassine Lakhnech, Cesar Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, Eli Singerman, Mandayam Srivas, Jens Skakkebæk, and Ashish Tiwari.

References

[AH96] Rajeev Alur and Thomas A. Henzinger, editors. *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[AH99] R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[BBM97] Nikolaj Bjørner, I. Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, and implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[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*, Hampton, VA, June 2000. NASA Langley Research Center. To appear.

[BL99] Saddek Bensalem and Yassine Lakhnech. Automatic generation of invariants. *Formal Methods in Systems Design*, 15(1):75–92, July 1999.

[BLO98] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In Hu and Vardi [HV98], pages 319–331.

[BLS96] Saddek Bensalem, Yassine Lakhnech, and Hassen Saïdi. Powerful techniques for the automatic generation of invariants. In Alur and Henzinger [AH96], pages 323–335.

[Bry86] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[CGJ$^+$00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Computer-Aided Verification*, Lecture Notes in Computer Science. Springer-Verlag, 2000. To appear.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CGP99] E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables. In *5th ACM Symposium on Principles of Programming Languages*. Association for Computing Machinery, January 1978.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, 1988.

[CU98] M. A. Col on and T. E. Uribe. Generating finite-state abstractions of reactive systems using decidion procedures. In Hu and Vardi [HV98], pages 293–304.

[DDP99] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In Halbwachs and Peled [HP99], pages 160–171.

[Dil96] David L. Dill. The Murϕ verification system. In Alur and Henzinger [AH96], pages 390–393.

[GPVW95] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. 15th Work. Protocol Specification, Testing, and Verification*, Warsaw, June 1995. North-Holland.

[Gur95] Yuri Gurevich. Evolving algebras 1993: Lipari guide. In Egon Börger, editor, *Specification and Validation Methods*, International Schools for Computer Scientists, pages 9–36. Oxford University Press, Oxford, UK, 1995.

[GW75] S. M. German and B. Wegbreit. A synthesizer for inductive assertions. *IEEE Transactions on Software Engineering*, 1(1):68–75, March 1975.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[Hoa69] C. A. R. Hoare. An axiomatic basis of computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[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.

[HV98] Alan J. Hu and Moshe Y. Vardi, editors. *Computer-Aided Verification, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, Vancouver, Canada, June 1998. Springer-Verlag.

[KM76] S. Katz and Z. Manna. Logical analysis of programs. *Communications of the ACM*, 19(4):188–206, April 1976.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM TOPLAS*, 16(3):872–923, May 1994.

[LGS$^+$95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:11–44, 1995.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1993.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Volume 1: Specification.* Springer-Verlag, New York, NY, 1992.

[MT96] Zohar Manna and The STeP Group. STeP: Deductive-algorithmic verification of reactive and real-time systems. In Alur and Henzinger [AH96], pages 415–418.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th Symposium on Foundations of Computer Science*, pages 46–57, Providence, RI, November 1977. ACM.

[Saï96] Hassen Saïdi. A tool for proving invariance properties of concurrent systems automatically. In *Tools and Algorithms for the Construction and Analysis of Systems TACAS '96*, volume 1055 of *Lecture Notes in Computer Science*, pages 412–416, Passau, Germany, March 1996. Springer-Verlag.

[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.

[SI77] N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *4th ACM Symposium on Principles of Programming Languages*, pages 132–143, January 1977.

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

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings 1st Annual IEEE Symp. on Logic in Computer Science*, pages 332–344. IEEE Computer Society Press, 1986.