An Overview of SAL[*]

To Appear in the Fifth Langley Formal Methods Workshop, June 2000

Saddek Bensalem[†] Vijay Ganesh[‡] Yassine Lakhnech[†] Cesar Muñoz[§] Sam Owre[¶]
Harald Rueß[¶] John Rushby[¶] Vlad Rusu[||] Hassen Saïdi[**] N. Shankar[¶]
Eli Singerman[††] Ashish Tiwari[‡‡]

Abstract

To become practical for assurance, automated formal methods must be made more scalable, automatic, and cost-effective. Such an increase in scope, scale, automation, and utility can be derived from an emphasis on a systematic separation of concerns during verification. SAL (Symbolic Analysis Laboratory) attempts to address these issues. It is a framework for combining different tools to calculate properties of concurrent systems. The heart of SAL is a language, developed in collaboration with Stanford, Berkeley, and Verimag, for specifying concurrent systems in a compositional way. Our instantiation of the SAL framework augments PVS with tools for abstraction, invariant generation, program analysis (such as slicing), theorem proving, and model checking to separate concerns as well as calculate properties (i.e., perform symbolic analysis) of concurrent systems. We describe the motivation, the language, the tools, their integration in SAL/PVS, and some preliminary experience of their use.

1 Introduction

To become practical for debugging, assurance, and certification, formal methods must be made more cost-effective. Incremental improvements to individual verification techniques will not suffice. It is our basic premise that a significant advance in the effectiveness and automation of verification of concurrent systems is possible by engineering a systematic separation of concerns through a truly integrated combination of static analysis, model checking, and theorem proving techniques. A key idea is to change the perception (and implementation) of model checkers and theorem provers from tools that perform verifications to ones that calculate *properties* such as slices, abstractions and invariants. In this way, big problems are cut down to manageable size, and properties of big systems emerge from those of reduced subsystems obtained by slicing, abstraction, and composition. By iterating through several such steps, it becomes possible to incrementally accumulate properties that eventually enable computation of a substantial new property—which in turn enables accumulation of further properties. By interacting at the level of properties and abstractions, multiple analysis tools can be used to derive properties that are beyond the capabilities of any individual tool.

SAL (Symbolic Analysis Laboratory) addresses these issues. It is a framework for combining different tools for abstraction, program analysis, theorem proving, and model checking toward the calculation of properties (symbolic analysis) of concurrent systems expressed as transition systems. The heart of SAL is an intermediate language, developed in collaboration with Stanford, Berkeley, and Verimag for specifying concurrent systems in a compositional way. This language will serve as the target for translators that extract the transition system description for popular programming languages such as Esterel, Java, or Verilog. The intermediate language also serves as a common description from which different analysis tools can be driven by translating the intermediate language to the input format for the tools and translating the output of these tools back to the

[*] This research was performed in the Computer Science Laboratory, SRI International, Menlo Park CA USA, and supported by DARPA through USAF Rome Laboratory contract F30602-96-C-0204, by NASA Langley Research Center contract NAS1-20334, and by the National Science Foundation contract CCR-9509931.

[†] VERIMAG, Grenoble, France
[‡] Stanford University, Stanford CA
[§] ICASE, NASA Langley, Hampton VA
[¶] Computer Science Laboratory, SRI International, Menlo Park CA
[||] IRISA, Rennes, France
[**] Systems Design Laboratory, SRI International, Menlo Park CA
[††] Intel, Haifa, Israel
[‡‡] SUNY Stony Brook, NY

SAL intermediate language.

This paper is structured as follows. In Section 2 we describe the motivation and rationale behind the design of the SAL language and give an overview of its main features. The main part, Section 3, describes SAL components including slicing, invariant generation, abstraction, model checking, simulation, and theorem proving together with their integration into the SAL toolset. Section 4 concludes with some remarks.

2 The SAL Common Intermediate Language

Mechanized formal analysis starts from a description of the problem of interest expressed in the notation of the tool to be employed. Construction of this description often entails considerable work: first to recast the system specification from its native expression in C, Esterel, Java, SCR, UML, Verilog, or whatever, into the notation of the tool concerned, then to extract the part that is relevant to the analysis at hand, and finally to reduce it to a form that the tool can handle. If a second tool is to be employed for a different analysis, then a second description of the problem must be prepared, with considerable duplication of effort. With m source languages and n tools, we need $m*n$ translators. This situation naturally suggests use of a common intermediate language, where the numbers of tools required could be reduced to $m + n$ translators.

The intermediate language must serve as a medium for representing the state transition semantics of a system described in a source language such as Java or Esterel. It must also serve as a common representation for driving a number of back-end tools such as theorem provers and model checkers. A useful intermediate language for describing concurrent systems must attempt to preserve both the structure and meaning of the original specification while supporting a modular analysis of the transition system.

For these reasons, the SAL intermediate language is a rather rich language. In the sequel, we give an overview of the main features of the SAL type language, the expression language, the module language, and the context language. For a precise definition and semantics of the SAL language, including comparisons to related languages for expressing concurrent systems, see [31].

The type system of SAL supports basic types such as booleans, scalars, integers and integer subranges, records, arrays, and abstract datatypes. Expressions are strongly typed. The expressions consist of constants, variables, applications of Boolean, arithmetic, and bit-vector operations (bit-vectors are just arrays of Booleans), and array and record selection and updates.

```
mutex : CONTEXT =
BEGIN

PC: TYPE = {trying, criti-
cal, sleeping}

mutex [tval:boolean] : MODULE =
BEGIN
INPUT  pc2: PC, x2: boolean
OUTPUT pc1: PC, x1: boolean

INITIALIZATION
  TRUE -->  pc1 = sleeping;
            x1  = tval

TRANSITION
   pc1 = sleeping
      --> pc1' = trying;
           x1' = (x2=tval)
  []
   pc1 = trying AND
    (pc2=sleeping OR x1= (x2/=tval))
      --> pc1' = critical
  []
   pc1 = critical
      --> pc1' = sleeping;
           x1' = (x2=tval)
END

system: MODULE =
  HIDE x1,x2
   (mutex[FALSE]
    || RENAME pc2 TO pc1,
              x2 TO x1,
              pc1 TO pc2,
              x1 TO x2
         mutex[TRUE])

mutualExclusion: THEOREM
  system |-
     AG(NOT(pc1=critical
            AND pc2=critical))

eventually1: LEMMA
     system |- EF(pc1=critical)
eventually2: LEMMA
     system |- EF(pc2=critical)

END
```

Figure 1. Mutual Exclusion

Conditional expressions are also part of the expression language and user-defined functions may also be introduced.

A module is a self-contained specification of a transition system in SAL. Usually, several modules are collected in a context. Contexts also include type and constant declarations. A transition system *module* consists of a *state* type, an *initialization condition* on this state type, and a binary *transition relation* of a specific form on the state type. The state type is defined by four pairwise disjoint sets of *input*, *output*, *global*, and *local* variables. The input and global variables are the *observed* variables of a module and the output, global, and local variables are the *controlled* variables of the module. It is good pragmatics to name a module. This name can be used to index the local variables so that they need not be renamed during composition. Also, the properties of the module can be indexed on the name for quick lookup.

Consider, for example, the SAL specification of a variant of Peterson's mutual exclusion algorithm in Figure 1. Here the state of the module consists of the controlled variables corresponding to its own program counter `pc1` and boolean variable `x1`, and the observed variables are the corresponding `pc2` and `x2` of the other process.

The transitions of a module can be specified variable-wise by means of *definitions* or transition-wise by *guarded commands*. Henceforth, primed variables `X'` denote next-state variables. A definition is of the form `X = f(Y, Z)`. Both the initializations and transitions can also be specified as guarded assignments. Each guarded command consists of a guarded formula and an assignment part. The guard is a boolean expression in the current controlled (local, global, and output) variables and current-state and next-state input variables. The assignment part is a list of equalities between a left-hand side next-state variable and a right hand side expression in both current-state and next-state variables.

Parametric modules allow the use of logical (state-independent) and type parameterization in the definition of modules. Module `mutex` in Figure 1, for example, is parametric in the Boolean `tval`. Furthermore, modules in SAL can be combined by either synchronous composition `||`, or asynchronous composition `[]`. Two instances of the `mutex` module, for example, are conjoined synchronously to form a module called `system` in Figure 1. This combination also uses *hiding* and *renaming*. Output and global variables can be made local by the `HIDE` construct. In order to avoid name clashes, variables in a module can be renamed using the `RENAME` construct.

Besides declaring new types, constants, or modules, SAL also includes constructs for stating module properties and abstractions between modules. CTL formulas are used, for example, in Figure 1 to state safety and liveness properties about the combined module `system`.

The form of composition in SAL supports a compositional analysis in the sense that any module properties expressed in linear-time temporal logic or in the more expressive universal fragment of CTL* are preserved through composition. A similar claim holds for asynchronous composition with respect to stuttering invariant properties where a stuttering step is one where the local and output variables of the module remain unchanged.

Because SAL is an environment where theorem proving as well as model checking is available, absence of causal loops in synchronous systems is ensured by generating proof obligations, rather than by more restrictive syntactic methods as in other languages. Consider the following definitions:

```
X = IF A THEN NOT Y ELSE C ENDIF
Y = IF A THEN B ELSE X ENDIF
```

This pair of definitions is acceptable in SAL because we can prove that `X` is *causally dependent* on `Y` only when `A` is *true*, and vice-versa only when it is *false*—hence there is no causal loop. In general, *causality checking* generates proof obligations asserting that the conditions that can trigger a causal loop are unreachable.

3 SAL Components

SAL is built around a blackboard architecture centered around the SAL intermediate language. Different backend tools operate on system descriptions in the intermediate language to generate properties and abstractions. The core of the SAL toolset includes the usual infrastructure for parsing and type-checking. It also allows integration of translators and specialized components for computing and verifying properties of transition systems. These components are loosely coupled and communicate through well-defined interfaces. An invariant generator may expect, for example, various application specific flags and a SAL base module, and it generates a corresponding assertion in the context language together with a justification of the invariant. The SAL toolset keeps track of the dependencies between generated entities, and provides capabilities similar to proof-chain analysis in theorem proving systems like PVS.

The main ingredients of the SAL toolset are specialized components for computing and verifying properties of transition systems. Currently, we have integrated various components providing basic capabilities for analyzing SAL specifications, including

- Validation based on theorem proving, model checking, and animation;
- Abstraction and invariant generation;
- Generation of counterexamples;
- Slicing.

We describe these components in more detail below.

3.1 Backend translations

We have developed translators from the SAL intermediate language to PVS, SMV, and Java for validating SAL specifications by means of theorem proving (in PVS), model checking (in SMV), and animation (in Java). These compilers implement *shallow structural embeddings* [26] of the SAL language; that is, SAL types and expressions are given a semantics with respect to a model defined by the logic of the target language. The compilers performs a limited set of semantic checks. These checks mainly concern the use of state variables. More complex checks, as for example type checking, are left to the verification tools.

3.1.1 Theorem Proving: SAL to PVS

PVS is a specification and verification environment based on higher-order logic [27]. SAL contexts containing definitions of types, constants, and modules, are translated into PVS theories. This translation yields a semantics for SAL transition systems. Modules are translated as parametric theories containing a record type to represent the state type, a predicate over states to represent the initialization condition, and a relation over states to represent the transition relation. Figure 2 describes a typical translation of a SAL module in PVS. Notice that initializations as well as transitions may be nondeterministic.

Compositions of modules are embedded as logical operations on the transition relations of the corresponding modules: disjunction for the case of asynchronous composition, conjunction for the case of synchronous composition. Hiding and renaming operations are modeled as morphisms on the state types of the modules. Logical properties are encoded via the temporal logic of the PVS specification language.

```
module[para:Parameters] : THEORY
BEGIN
  State : TYPE = [#
    input  : InputVars,
    output : OutputVars,
    local  : LocalVars
  #]

  state,next : VAR State

  initialization(state):boolean =
    (guard_init_1 AND
     output(state) = ... AND
     local(state) = ...)
    OR ... OR (guard_init_n AND ...)

  transitions(state, next):boolean =
    (guard_trans_1 AND
     output(next) =
          output(state) WITH [...]
     local(next) =
          local(state) WITH [... ])
    OR ... OR
    (guard_trans_m AND ...)
    OR
    (NOT guard_trans_1 AND ... AND
     NOT guard_trans_m AND
     output(next) = output(state)
     local(next) = local(state))
```

Figure 2. A SAL module in PVS

3.1.2 Model Checking: SAL to SMV

SMV is a popular model checker with its own system description language [25]. SAL modules are mapped to SMV modules. Type and constant definitions appearing in SAL contexts are directly expanded in the SMV specifications. Output and local variables are translated to variables in SMV. Input variables are encoded as parameters of SMV modules.

The nondeterministic assignment of SMV is used to capture the arbitrary choice of an enabled SAL transition. Roughly speaking, two extra variables are introduced. The first is assigned nondeterministically with a value representing a SAL transition. The guard of the transition represented by this variable is the first guard to be evaluated. The second variable loops over all transitions starting from the chosen one until it finds a transition which is enabled. This mechanism assures that every transition satisfying the guard has an equal chance to being fired in the first place. Composition of SAL modules and logical properties are directly translated via the specification language of SMV.

3.1.3 Animation: SAL to Java

Animation of SAL specifications is possible via compilation to Java. However, not all the features of the SAL language are supported by the compiler. In particular, the expression language that is supported is limited to that of Java. For example, only integers and booleans are accepted as basic types. Elements of enumeration types are translated as constants and record types are represented by classes.

The state type of a SAL module is represented by a class containing fields for the input, output, and local variables. In order to simulate the nondeterminism of the initialization conditions, we have implemented a random function that arbitrary chooses one of the initialization transition satisfying the guard.

Each transition is translated as a Java thread class. At execution time, all the threads share the same state object. We assume that the Java virtual Machine is nondeterministic with respect to execution of threads. The main function of the Java translation creates one state object and passes the object as an argument to the thread object constructors. It then starts all the threads. Safety properties are encoded by using the exception mechanism of Java, and are checked at run time.

3.1.4 Case Study: Flight Guidance System

Mode confusion is a concern in aviation safety. It occurs when pilots get confused about the actual states of the flight deck automation. NASA Langley conducts research to formally characterize mode confusion situations in avionics systems. In particular, a prototype of a Flight Guidance System (FGS) has been selected a case study for the application of formal techniques to identify mode confusion problems. FGS has been specified in various formalisms (see [23] for a comprehensive list of related work). Based on work by Lüttgen and Carreño, we have developed a complete specification of FGS in SAL. The specification has been automatically translated to SMV and PVS, where it has been analyzed. We did not experience any significant overhead in model checking translated SAL models compared to hand-coded SMV models. This case study is available at `http://www.icase.edu./~munoz/sources.html`.

3.2 Invariant Generation

An *invariant* of a transition system is an assertion—a predicate on the state—that holds of every reachable state of the transition system. An *inductive invariant* is a assertion that holds of the initial states and is preserved by each transition of the transition system. An inductive invariant is also an invariant but not every invariant is inductive.

Let $\mathtt{SP}(\mathcal{T},\phi)$ denote the formula that represents the set of all states that can be reached from any state in ϕ via a single transition of the system $\mathcal{T}$, and Θ denote the formula that denotes the initial states. A formula ϕ is an inductive invariant for the transition system $\mathcal{T}$ if (i) $\Theta \rightarrow \phi$; (ii) $\mathtt{SP}(\mathcal{T},\phi) \rightarrow \phi$.

We recall that for a given transition system $\mathcal{T}$ and a set of states described by formula ϕ, the notation $\mathtt{SP}(\mathcal{T},\phi)$ denotes the formula that characterizes all states reachable from states ϕ using exactly one transition from $\mathcal{T}$. If Θ denotes the initial state, then it follows from the definition of invariants that any fixed-point of the operator $F(\phi) = \mathtt{SP}(\mathcal{T},\phi) \vee \Theta$ is an invariant.

Notice that the computation of strongest postconditions introduces existentially quantified formulas. Due to novel theorem proving techniques in PVS2.3 that are based on the combination of a set of ground decision procedures and quantifier elimination we are able to effectively reason about these formulas in many interesting cases.

It is a simple observation that not only is the greatest fixed point of the above operator an invariant, but every intermediate ϕ_i generated in an iterated computation procedure of greatest fixed point also is an invariant.

$$\begin{array}{rcl} \phi_0 & : & \mathtt{true} \\ \phi_{i+1} & : & \mathtt{SP}(\mathcal{T},\phi_i) \vee \Theta \end{array}$$

A consequence of the above observation is that we do not need to detect when we have reached a fixed point in order to output an invariant.

As a technical point about implementation of the above greatest fixed point computation in SAL, we mention that we break up the (possibly infinite) state space of the system into finitely many (disjoint) control states. Thereafter, rather than working with the global invariants ϕ_i, we work with local invariants that hold at particular control states. The iterative greatest fixed point computation can now be seen as a method of generating invariants based on *affirmation* and *propagation* [6].

Note that rather than computing the greatest fixed point, if we performed the least fixed point computation, we would get the strongest invariant for any given system. The problem with least fixed points is that their computation does not converge as easily as those of greatest fixed points. Unlike greatest fixed points, the intermediate predicates in the computation of the least fixed point are not invariants. We are currently investigating approaches based on widening to compute invariants in a convergent manner using least fixed points [8].

The techniques described so far are noncompositional since they examine all the transitions of the given system. We use a novel composition rule defined in [29] allowing local invariants of each of the modules to be composed into global invariants for the whole system. This composition rule allows us to generate stronger invariants than the invariants generated by the techniques described in [6, 7]. The generated invariants allows us to obtain boolean abstractions of the analyzed system using the incremental analysis techniques presented in [29].

3.3 Slicing

Program analyses like slicing can help remove code irrelevant to the property under consideration from the input transition system which may result in a reduced state-space, thus easing the computational needs of subsequent formal analysis efforts. Our slicing tool [18] accepts an input transition system which may be synchronously or asynchronously composed of multiple modules written in SAL and the property under verification. The property under verification is converted into a slicing criterion and the input transition system is sliced with respect to this slicing criterion. The slicing criterion is merely a set of local/output variables of a subset of the modules in the input SAL program that are not relevant to the property. The output of the slicing algorithm is another SAL program similarly composed of modules wherein irrelevant code manipulating irrelevant variables from each module has been sliced out. For every input module there will be an output module, empty or otherwise. In a nutshell the slicing algorithm does a dependency analysis of each module and computes backward transitive closure of the dependencies. This transitive closure would take into consideration only a subset of all transitions in the module. We call these transitions observable and the remaining transitions are called τ or silent transitions. We replace silent transitions with skips.

We are currently investigating reduction techniques that are simpler than slicing and also ones that are more aggressive. One example is the cone-of-influence reduction where the slicing criterion is a set of variables V, and the reduction computes a transition system that includes all the variables in the transitive closure of V given by the dependencies between variables [21]. In comparison with slicing, the cone-of-influence reduction is insensitive to control and is therefore easier to compute but generally not as efficient at pruning irrelevance. Slicing preserves program behavior with respect to the slicing criterion. One could obtain a more dramatic reduction by admitting slices that admitted more behaviors by introducing nondeterminism. Such aggressive slicing would be needed for example to abstract away from the internal behavior of a transition system within its critical section for the purpose of verifying mutual exclusion. Slicing for concurrent systems with respect to temporal properties has been investigated by Dwyer and Hatcliff [16].

3.4 Connecting InVeSt with SAL

So far we have described specialized SAL components that provide core features for the analysis of concurrent systems, but we have also integrated the standalone InVeSt [5] into the SAL framework. Besides compositional techniques for constructing abstraction and features for generating counterexamples from failed verification attempts, InVeSt introduces alternative methods for invariant generation to SAL. InVeSt not only serves as a backend tool for SAL but also has been connected to the IF laboratory [10], Aldebaran [9], TGV [17] and Kronos [15].

The salient feature of InVeSt is that it combines the algorithmic with the deductive approaches to program verification in two different ways. First, it integrates the principles underlying the algorithmic (e.g. [11, 28]) and the deductive methods (e.g. [24]) in the sense that it uses fixed point calculation as in the algorithmic approach but also the reduction of the invariance problem to a set of first-order formulas as in the deductive approach. Second, it integrates the theorem prover PVS [27] with the model checker SMV [25] through the automatic computation of finite abstractions. That is, it provides the ability to automatically compute finite abstractions of infinite state systems which are then analyzed by SMV or, alternatively, by the model checker of PVS. Furthermore, InVeSt supports the proof of invariance properties using the method based on induction and auxiliary invariants (e.g. [24]) as well as a method based on abstraction techniques [2, 12–14, 21, 22]. InVeSt uses PVS as a backend tool and depends heavily on its theorem proving capabilities for deciding the myriad verification conditions.

3.4.1 Abstraction

InVeSt provides also a capability that computes an abstract system from a given concrete system and an abstraction function. The method underlying this technique is presented in [4]. The main features of this method is that it is automatic and compositional. It computes an abstract system $S^a = S^1_\alpha \parallel \cdots \parallel S^n_\alpha$, for a given system $S = S^1 \parallel \cdots \parallel S^n$ and abstraction function α, such that S simulates S_α is guaranteed by the construction. Hence, by known preservation results, if S_α satisfies an invariant φ then S satisfies the invariant $\alpha^{-1}(\varphi)$. Since the produced abstract system is not given by a graph but in a programming language, one still can apply all the known methods for avoiding the state explosion problem while analyzing S_α. Moreover, it generates an abstract system which has the same structure as the concrete one. This gives the ability to apply further abstractions and techniques to reduce the state explosion problem and facilitates the debugging of the concrete system. The computed abstract system is optionally represented in the specification language of PVS or in that of SMV.

The basic idea behind our method of computing abstractions is simple. In order to construct an abstraction of S, we construct for each concrete transition τ_c an abstract transition τ_a. To construct τ_a we proceed by elimination starting from the universal relation, which relates every abstract state to every abstract state, and eliminate pairs of abstract states in a conservative way, that is, it is guaranteed that after elimination of a pair the obtained transition is still an abstraction of τ_c. To check whether a pair (a, a') of abstract states can be eliminated we have to check that the concrete transition τ_c does not lead from any state c with $\alpha(c) = a$ to any state c' with $\alpha(c') = a'$. This amounts to proving a Hoare triple. The elimination method is in general too complex. Therefore, we combine it with three techniques that allow many fewer Hoare triples to be checked. These techniques are based on partitioning the set of abstract variables, using substitutions, and a new preservation result which allows to use the invariant to be proved during the construction process of the abstract system.

We implemented our method using the theorem prover PVS [27] to check the Hoare triples generated by the elimination method. The first-order formulas corresponding to these Hoare triples are constructed automatically and a strategy that is given by the user is applied. In [1] we developed also a general analysis methodology for *heterogeneous* infinite-state models, extended automata operating on variables which may range over several different domains, based on combining abstraction and symbolic reachability analysis.

3.4.2 Generation of Invariants

There are two different way to generate invariants in InVeSt. First, we use calculation of pre-fixed points by applying the body of the backward procedure a finite number of times and use techniques for the automatic generation of invariants (cf. [3]) to support the search for auxiliary invariants. The tool provides strategies which allow derivation of *local invariants*, that is, predicates attached to control locations and which are satisfied whenever the computation reaches the corresponding control point. InVeSt includes strategies for deriving local invariants for sequential systems as well as a composition principle that allows combination of invariants generated for sequential systems to obtain invariants of a composed system. Consider a composed system $S_1 \parallel S_2$ and control locations l_1 and l_2 of S_1 and S_2, respectively. Suppose that we generated the local invariants P_1 and P_2 at l_1 and l_2, respectively. Let us call P_i *interference independent*, if P_i does not contain a free variable that is written by S_j with $j \neq i$. Then, depending on whether P_i is interference independent we compose the local invariants P_1 and P_2 to obtain a local invariant at (l_1, l_2) as follows: if P_i is interference independent, then we can affirm that P_i is an invariant at (l_1, l_2) and if both P_1 and P_2 are interference dependent, then $P_1 \vee P_2$ is an invariant at (l_1, l_2). This composition principle proved to be useful in the examples we considered. However, examples showed that predicates obtained by this composition principle can become very large. Therefore, we also consider the alternative option where local invariants are not composed until they are needed in a verification condition. Thus, we assign to each component of the system two lists of local invariants. The first corresponds to interference independent local invariants and the second to interference dependent ones. Then, when a verification condition is considered, we use heuristics to determine which local invariants are useful when discharging the verification condition. A useful heuristic concerns the case when the verification condition is of the form $(pc(1) = l_1 \wedge pc(2) = l_2) \Rightarrow \phi$, where $pc(1) = l_1 \wedge pc(2) = l_2$ asserts that computation is at the local control locations l_1 and l_2. In this case, we combine the local invariants associated to l_1 and l_2 and add the result to the left hand side of the implication.

Second, we use abstraction generating invariants at the concrete level: Let $\mathcal{S}_{\alpha_1}$ the result of the abstraction of a concrete system $\mathcal{S}$, the set of reachable states denoted by $Reach(\mathcal{S}_{\alpha_1})$ is an invariant of $\mathcal{S}_{\alpha_1}$ (the strongest one including the initial configurations in fact). We developed a method that extract the formula which characterizes the reachable states from the BDD. Hence, $\alpha_1^{-1}(Reach(\mathcal{S}_{\alpha_1}))$ is an invariant of the concrete model $\mathcal{S}$. This invariant can be used to strengthen φ and show that it is an invariant of $\mathcal{S}$.

3.4.3 Analysis of Counterexamples

The generation of the abstract system is *completely automatic* and compositional as we consider transition by transition. Thus, for each concrete transition we obtain an abstract transition (which might be nondeterministic). This is a very important property of our method, since it enables the debugging of the concrete system or alternatively enhancing the abstraction function. Indeed, the constructed abstract system may not satisfy the desired property, for three possible reasons:

1. The concrete system does not satisfy the invariant,
2. The abstraction function is not suitable for proving the invariant, or
3. The proof strategies provided are too weak.

Now, a model checker such as SMV provides a trace as a counterexample, if the abstract system does not satisfy the abstract invariant. Since we have a clear correspondence between abstract and concrete transitions, we can examine the trace and find out which of the three reasons listed above is the case. In particular if the concrete system does not satisfy the invariant then we can transform the trace given by SMV to a concrete trace, thus generating a concrete counterexample.

3.5 Predicate/Boolean Abstraction

In addition to the InVeSt abstraction mechanisms, we implemented boolean abstraction of SAL specifications. We use the boolean abstraction scheme defined in [19] that uses predicates over concrete variables as abstract variables to abstract infinite or large state systems into finite state systems analyzable by model checking. The advantage of using boolean abstractions can be summarized as follows:

- Any abstraction to a finite state system can be expressed as a boolean abstraction.
- The abstract transition relation can be represented symbolically using Binary Decision Diagram (BDDs). Thus, efficient symbolic model checking [25] can be effectively applied.
- We have defined in [30] an efficient algorithm for the construction of boolean abstractions. We also designed an efficient refinement technique that allows us to refine automatically an already constructed abstraction until the property of interest is proved or a counter-example is generated.
- Abstraction followed by model checking and successive refinement is an efficient and more powerful alternative to invariant generation techniques such as the ones presented in [6, 7].

3.5.1 Automatic Construction of Boolean Abstractions

The automatic abstraction module takes as input a SAL basemodule and a set of predicates defining the boolean abstraction. Using the algorithm in [30] we automatically construct the corresponding abstract transition system. This process relies heavily on the PVS decision procedures.

```
...
INPUT   x: integer
OUTPUT  y, z: integer

INITIALIZATION
  TRUE -->  INIT(x) = 0;
            INIT(y) = 0;
            INIT(z) = y;

TRANSITION
  NOT(x > 0)  -->  y' = y + 1
  []   z > 0  -->  z' = y - 1, y' = 0
...
```

Figure 3. Concrete Module.

Figure 3 and 4 display a simple SAL module and its abstraction where the boolean variables B1, B2 and B3 correspond to the predicates $x > 0$, $y > 0$, and $z > 0$. Notice that the assignment to B3 is nondeterministically chosen from the set {TRUE, FALSE}.

```
...
INPUT  B1: boolean
OUTPUT B2,B3: boolean

INITIALIZATION
  TRUE -->  INIT(B1) = FALSE;
            INIT(B2) = FALSE;
            INIT(B3) = FALSE;

TRANSITION
    NOT(B1) --> B2'=F
 [] B3 --> B2'=T, B3'= { TRUE, FALSE }
...
```

Figure 4. Abstract Module.

3.5.2 Explicit Model Checking

Finite-state SAL modules can be translated to SMV for model checking as explained above. However, model checkers usually do not allow to access their internal data structures where intermediate computation steps of the model-checking process can be exploited. For this reason, we implemented an efficient explicit-state model checker for SAL systems obtained by boolean abstraction. The abstract SAL description is translated into an executable Lisp code that performs the explicit state model checking procedure allowing us to explore about twenty thousand states a second. This procedure builds an abstract state graph that can be exploited for further analysis. Furthermore, additional abstractions can be applied on the fly while the abstract state graph is being built.

3.5.3 Automatic Refinement of Abstractions

When model checking fails to establish the property of interest, we use the results developed in [29, 30] to decide whether the constructed abstraction is too coarse and needs to be refined, or that the property is violated in the concrete system and that the ge nerated counterexample corresponds indeed to an execution of the concrete system violating the property. This is done by examining the generated abstract state graph. The refinement technique computes the precondition to a transition where nondeterministic assignments occur. The preconditions corresponding to the cases where the variables get either `TRUE` or `FALSE` define two predicates that are used as new abstract variables. The following transition from the example

```
B3 --> B2'=TRUE, B3'= {TRUE, FALSE}
```

can be automatically refined to

```
B3 --> B2'=TRUE, B3'=B4 ,
       B4'=FALSE, B5' = FALSE
```

where `B4` and `B5` correspond to the predicates `y=1` and `y>1`, respectively.

4 Conclusions

SAL is a tool that combines techniques from static analysis, model checking, and theorem proving in a truly integrated environment. Currently, its core is realized as an extension of the PVS system and has a well-defined interface for coupling specialized analysis tools. So far, we have been focusing on developing and connecting back-end tools for validating SAL specifications by means of animation, theorem proving, and model checking, and also for computing abstractions, slices, and invariants of SAL modules. There are as yet no automated translators into the SAL language. Primary candidates are translators for source languages such as Java, Verilog, Esterel, Statecharts, or SDL. Since SAL is an open system with well-defined interfaces, however, we hope others will write those if the rest of the system proves effective.

We are currently completing the implementation of the SAL prototype which includes a parser, typechecker, a slicer, an invariant generator, the connection to InVeSt, and translators to SMV and PVS. We expect to release the prototype SAL system in mid-2000.

Although our experience with the combined power of several forms of mechanized formal analysis in the SAL system is still rather limited, we predict that proofs and refutations of concurrent systems that currently require significant human effort will soon become routine calculations.

References

[1] P. A. Abdulla, A. Annichini, S. Bensalem, A. Bouajjani, P. Habermehl, and Y. Lakhnech. Verification of infinite-state systems by combining abstraction and reachability analysis. In Halbwachs and Peled [20], pages 146–159.

[2] S. Bensalem, A. Bouajjani, C. Loiseau, and J. Sifakis. Property preserving simulations. In G. v. Bochmann and D. K. Probst, editors, *Computer Aided Verification'92*, volume 663 of *LNCS*, pages 260–273. Springer-Verlag, 1992.

[3] S. Bensalem and Y. Lakhnech. Automatic generation of invariants. *Formal Methods in System Design*, 15(1):75–92, July 1999.

[4] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems automatically and compositionally. In A. J. Hu and M. Y. vardi, editors, *Computer Aided Verification*, volume 1427 of *LNCS*, pages 319–331. Springer-Verlag, 1998.

[5] S. Bensalem, Y. Lakhnech, and S. Owre. InVeSt: A tool for the verification of invariants. In A. J. Hu and M. Y. vardi, editors, *Computer Aided Verification*, volume 1427 of *LNCS*, pages 505–510. Springer-Verlag, 1998.

[6] S. Bensalem, Y. Lakhnech, and H. Saïdi. Powerful techniques for the automatic generation of invariants. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 323–335, New Brunswick, NJ, July/Aug. 1996. Springer-Verlag.

[7] N. Bjørner, I. A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.

[8] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In D. Bjørner, M. Broy, and I. V. Pottosin, editors, *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, pages 128–141, 1993. Vol. 735 of *Lecture Notes in Computer Science*, Springer-Verlag.

[9] M. Bozga, J. Fernandez, A. Kerbrat, and L. Mounier. Protocol verification with the Aldebaran toolset. *Software Tools and Technology Transfer journal*, 1998.

[10] M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J. Krimm, and L. Mounier. IF: An Intermediate Representation and Validation Environment for Timed Asyn-

chronous Systems. In *Proceedings of FM'99, Toulouse, France*, LNCS, 1999.

[11] E. Clarke, E. Emerson, and E. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *10th ACM symp. of Prog. Lang.* ACM Press, 1983.

[12] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5), 1994.

[13] D. Dams. *Abstract interpretation and partition refinement for model checking*. PhD thesis, Technical University of Eindhoven, 1996.

[14] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems: Abstractions preserving ACTL*, ECTL* and CTL*. In *Proceedings of the IFIP WG2.1/WG2.2/WG2.3 (PROCOMET)*. IFIP Transactions, North-Holland/Elsevier, 1994.

[15] C. Daws, A. Olivero, and S. Yovine. Verifying ET-LOTOS programs with KRONOS. In *Proc. FORTE'94*, Berne, Switzerland, Oct. 1994.

[16] M. B. Dwyer and J. Hatcliff. Slicing software for model construction. In *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, Jan. 1999.

[17] J.-C. Fernandez, C. Jard, T. Jéron, L. Nedelka, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In R. Alur and T. A. Henzinger, editors, *Proceedings of the 8th International Conference on Computer-Aided Verification (Rutgers University, New Brunswick, NJ, USA)*, volume 1102 of *LNCS*. Springer Verlag, 1996. Also available as INRIA Research Report RR-2987.

[18] V. Ganesh, H. Saïdi, and N. Shankar. Slicing SAL. Draft, 1999.

[19] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Conference on Computer Aided Verification CAV'97*, LNCS 1254, Springer Verlag, 1997.

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

[21] R. Kurshan. *Computer-Aided Verification of Coordinating Processes, the automata theoretic approach*. Princeton Series in Computer Science. Princeton University Press, 1994.

[22] D. E. Long. *Model Checking, Abstraction, and Compositional Reasoning*. PhD thesis, Carnegie Mellon, 1993.

[23] G. Lüttgen and V. Carreño. Analyzing mode confusion via model checking. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking (SPIN '99)*, volume 1680 of *Lecture Notes in Computer Science*, pages 120–135, Toulouse, France, September 1999. Springer-Verlag.

[24] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[25] K. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, Boston, 1993.

[26] C. Muñoz and J. Rushby. Structural embeddings: Mechanization with method. In *Proceedings of the World Congress on Formal Methods FM 99*, volume 1708 of *LNCS*, pages 452–471, 1999.

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

[28] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Int. Sym. on Programming*, volume 137 of *LNCS*, pages 337–351. Springer-Verlag, 1982.

[29] H. Saïdi. Modular and incremental analysis of concurrent software systems. In *14th IEEE International Conference on Automated Software Engineering*, Oct. 1999.

[30] H. Saïdi and N. Shankar. Abstract and model check while you prove. In Halbwachs and Peled [20], pages 443–454.

[31] The SAL Group. The SAL intermediate language. Available at: `http://sal.csl.sri.com/`, 1999.