Model Checking for the Practical Verificationist:

A User's Perspective on SAL

Lee Pike
Galois, Inc.
leepike@galois.com

ABSTRACT

SRI's Symbolic Analysis Laboratory (SAL) is a high-level language-interface to a collection of state-of-the-art model checking tools. SAL contains novel and powerful features, many of which are not available in other model checkers. In this experience report, I highlight some of the features I have particularly found useful, drawing examples from published verifications using SAL. In particular, I discuss the use of higher-order functions in model checking, infinite-state bounded model checking, compositional specification and verification, and finally, mechanical theorem prover and model checker interplay. The purpose of this report is to expose these features to working verificationists and to demonstrate how to exploit them effectively.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: [formal methods, model checking]

1. INTRODUCTION

SRI's Symbolic Analysis Laboratory (SAL)[1] is bad news for interactive mechanical theorem provers. SAL is so automated yet expressive that for many of the verification endeavors I might have previously used a mechanical theorem prover, I would now use SAL. The purpose of this brief report is to persuade you to do the same.

To convince the reader, I highlight SAL's features that are especially useful or novel from a practitioner's perspective. My goals in doing so are (1) to begin a dialogue with other SAL users regarding how best to exploit the tool, (2) to show off SAL's features to verificationists not yet using SAL, and (3) to provide user feedback to spur innovation in the dimensions I have found most novel and beneficial, as a user.

[1]SAL is open source under a GPL license and the tool, documentation, a user-community wiki, etc. are all available at `http://sal.csl.sri.com`.

To appear in the proceedings of *Automated Formal Methods* (AFM07), November 2007, Atlanta, Georgia.

With my coauthors, I have had the opportunity to use SAL in a number of applied verifications [3, 4, 5, 20, 21, 22].[2] These works draw from the domains of distributed systems, fault-tolerant protocols, and asynchronous hardware protocols (the most notable omission is the domain of software, although the techniques reported are not domain-specific).

Let me also say what are *not* the intentions of this report. This report is not a manual or user's guide; such artifacts are available from the SAL website (I do, however, strive to make the report self-contained, even for the reader not experienced with SAL). Also, I do not compare and contrast SAL to other model checkers. In particular, I am not claiming that each feature highlighted is unique to SAL, but I do claim that their combination in one tool is unique.

The outline for the remainder of this report is as follows. In Section 2, I briefly overview SAL's language and toolset. In Section 3, I describe how higher-order functions, as implemented in SAL, can be used in a model checking context to simplify the specification of state machines. In Section 4, I describe SAL's infinite-state bounded model checker, a particularly novel and powerful model checker, and I describe how it can be used to prove safety-properties of real-time systems with an order of magnitude reduction in proof steps as compared to mechanical theorem proving. Section 5 describes how to judiciously use synchronous and asynchronous composition to ease the proof effort required in infinite-state bounded model checking and to decompose environmental constraints from system specifications. Despite the power of SAL, a model checker sometimes is not enough; in Section 6, I describe cases in which SAL can be effectively used in tandem with a mechanical theorem prover. Conclusions are provided in Section 7.

2. SAL OVERVIEW

SAL has a high-level modeling language for specifying state machines. A state machine is specified by a *module*. A module consists of a set of state variables (declared to be *input*, *output*, *global*, or *local*) and guarded transitions. A guarded transitions is *enabled* if its guard—some expression that evaluates to a boolean value—is true. Of the enabled transitions in a state, one is nondeterministically executed. When a transition is exercised, the next-state values are

[2]My coathors for these works include Geoffrey Brown, Steve Johnson, Paul Miner, and Wilfredo Torres-Pomales. The specifications associated with these works are all available from `http://www.cs.indiana.edu/~lepike`.

assigned to variables; for example, consider the following guarded transition:

```
H --> a' = a - 1;
      b' = a;
      c' = b' + 1;
```

If the guard `H` holds and the transition is exercised, then in the next state, the variable `a` is decremented by one, the variable `b` is updated to the previous value of `a`, and the variable `c` is updated to the new value of `b`, plus one. In the language of SAL, "`;`" denotes statement separation, not sequential composition (thus, variable assignments can be written in any order). If no variables are updated in a transition (i.e., `H -->`), the state idles.

Modules can be composed both synchronously (`||`) and asynchronously (`[]`), and composed modules communicate via shared variables. In a synchronous composition, a transition from each module is simultaneously applied; a synchronous composition is deadlocked if either module has no enabled transition. Furthermore, a syntactic constraint on modules requires that no two modules update the same variable in a synchronous composition. In an asynchronous composition, an enabled transition from exactly one of the modules is nondeterministically applied.

The language is typed, and predicate sub-types can be declared. Types can be both interpreted and uninterpreted, and base types include the reals, naturals, and booleans. Array types, inductive data-types, and tuple types can be defined. Both interpreted and uninterpreted constants and functions can be specified.

One of the greatest practical benefits of SAL is that a variety of useful tools are associated with the same input language. SAL 3.0 includes a BDD-based model checker, a SAT based model checker (capable of performing k-induction proofs), and infinite-state bounded model checker that is integrated with the Yices satisfiability modulo theories (SMT) solver, a BDD-based simulator, a BDD-based deadlock checker, and an automated test-case generator. Other tools can be built on SAL's API.

3. HIGHER-ORDER FUNCTIONS

The first feature of SAL I cover is higher-order functions. What use are higher-order functions in a model checker? Model checkers are about building state machines, and higher-order functions are typically associated with "stateless" programming. The practicality of higher-order functions is well-known in the programming and mechanical theorem proving communities [12], and these advantages apply just as well to specifying the functional aspects of a model. Furthermore, higher-order functions are just as indispensable for specifying a model's stateful aspects; I give two supporting examples below. First, I show how sets can be specified with higher-order functions, allowing the easy specification of nondeterministic systems. Second, I show how to replace guarded transitions with higher-order functions; the benefit of doing so is that it allows one to decompose the environment and system specifications or to make assumptions explicit in proofs.

3.1 Sets and Nondeterminism

The first example is drawn from work done with Geoffrey Brown to verify real-time physical-layer protocols [3, 5]. Suppose I want to nondeterministically update some value to be within a parameterized closed interval of real-time (modeled by the real number line). We can define a higher-order function that takes a minimum and a maximum value and returns the set of real values in the closed interval between them. The function has the return type of `REAL -> BOOLEAN`, which is the type of a set of real numbers.

```
timeout( min : REAL
       , max : REAL) : [REAL -> BOOLEAN] =
  {x : REAL |     min <= x
              AND x <= max};
```

I use the identifier '`timeout`' to pay homage to *timeout automata*, a theory and corresponding implementation in SAL developed by Bruno Dutertre and Maria Sorea for specifying and verifying real-time systems using infinite-state k-induction (see Section 4) [10].

Then, in specifying the guarded transitions in a state machine, we can simply call the function with specific values (let `H` and `I` be some predicates on the real-time argument). The `IN` operator specifies that its first argument, a variable of some arbitrary type `T`, nondeterministically takes a value in its second argument, a set of type `T -> BOOLEAN`.

```
   H(t) --> t' IN timeout(1, 2);
[] I(t) --> t' IN timeout(3, 5);
```

With higher-order functions, nondeterministic transitions can be specified succinctly, as above, rather than specifying an interval in each transition.[3]

3.2 Specifying Transitions

Another benefit of higher-order functions is that they can be used to "pull" constraints out of the state machine specification. The motivations for doing this include (1) to simplify specifications, (2) to make assumptions and constraints more explicit in proofs, and (3) to decompose environmental constraints from state machine behavior.

To illustrate this idea, we will model check a simple distributed system built from a set of nodes (of the uninterpreted type `NODE`) and a set of one-way channels between nodes. Furthermore, suppose that the creation of channels is dynamic, and we wish to make this explicit in our state-machine model of the system. We might prove properties like, "If there is a channel pointing from node `n` to node `m`, then there is no channel pointing from `m` to `n`," (i.e., channels are unidirectional) or "No channel points from a node in one subset of nodes to a node in another subset of nodes."

[3]In this and subsequent specifications, we sometimes state just the guarded transitions where the remainder of the guard specification is irrelevant or can be inferred from context.

To build the model, we will record the existence of channels using a NODE by NODE matrix of booleans. For convenience, we define CHANS to be the type of these matrices:

```
CHANS : TYPE = ARRAY NODE OF ARRAY NODE OF BOOLEAN;
```

For some matrix chans of type CHANS, we choose, by convention, to let chans[a][b] mean that there is a channel pointing from node a to node b.

We define a function newChan that takes two nodes and adds a channel between them. Indeed, having higher-order functions at our disposal, we define the function in curried form.

```
newChan(a : NODE, b : NODE) : [CHANS -> CHANS] =
  LAMBDA (chans : CHANS) :
    chans WITH [a][b] := TRUE;
```

The function newChan returns a function, and that function takes a set of channels and updates it with a channel from a to b.

Now we will build a state machine containing three asynchronous transitions. Two of the transitions introduce new channels into the system depending on the current system state, and the final one simply stutters (i.e., maintains the system state). Let H and I be predicates over sets of channels (i.e., functions of type CHANS -> BOOLEAN), and let m, n, o, p, ... be constant node-identifiers (i.e., constants of type NODE).

```
   H(chans) --> chans' =  newChan(n, m)
                         (newChan(p, q)
                         (chans));
[] I(chans) --> chans' =  newChan(q, n)(chans);
[] ELSE -->
```

Alternatively, we can write an equivalent specification using a predicate rather than defining three transitions in the state machine. First, we define a function that takes a current channel configuration and returns a set of channel configurations—i.e., chanSet returns a predicate parameterized by its argument.

```
chanSet(chans : CHANS) : [CHANS -> BOOLEAN] =
  {x : CHANS |
        (H(chans) => x =  newChan(n, m)
                         (newChan(p, q)
                         (chans)))
    AND (I(chans) => x =  newChan(q, n)(chans))
    AND (   (NOT I(chans) AND NOT H(chans))
         => FORALL (a, b : NODE) :
              x[a][b] = chans[a][b]))};
```

Now, we can specify the state-machine with the following single transition:

```
TRUE --> chans' IN chanSet(chans);
```

In all states, chans is updated to be some configuration of channels from the possible configurations returned by chanSet(chans). Why might one wish to "pull" transitions out of a state-machine specification and into a predicate? One reason would be to decompose the transitions enabled by the state machine itself from the environmental constraints over the machine. For example, suppose that in our example distributed system, the nodes themselves are not responsible for creating new channels—some third party does so. In this case, specifying channel creation in the transitions of the state-machine module itself belies the distinction between the distributed system and its environment. (See Section 5.2 for an industrial application of this idea.)

We can even go one step further and remove the constraints entirely from the model. First, we modify the previous transition so that it is completely unconstrained: under any condition (i.e., a guard of TRUE), it returns any configuration of channels. (We also add an auxiliary history variable, the sole purpose of which is to record the set of channels in the previous state; we use the variable shortly.

```
TRUE --> chans' IN {x : CHANS | TRUE};
         chansHist' = chans
```

Now suppose we were to prove some property about the machine; for instance, suppose we wish to prove that all channels between two nodes are unidirectional. We might state the theorem as follows.[4]

```
Thm : THEOREM system |-
  G(FORALL (a,b : NODE) :
      chans[a][b] => NOT chans[b][a]);
```

With a completely under-specified state machine, the theorem fails. We are forced to add as an explicit hypothesis that the state variable chans belongs to the set of channels chanSet(chansHist) generated in the previous state.[5]

```
Thm : THEOREM system |-
  LET inv : BOOLEAN = chanSet(chansHist)(chans)
  IN W( (inv => FORALL (a, b : NODE) :
                  chans[a][b] => NOT chans[b][a])
       , NOT inv);
```

[4]This and subsequent SAL theorems, lemmas, etc. are stated in the language of Linear Temporal Logic (LTL), a common model-checking logic. In the following theorem, the G operator states that its argument holds in all states on an arbitrary path, and LTL formulas are implicitly quantified over all paths in the system. See the SAL documentation for more information.

[5]Due to the semantics of LTL, we cannot simply add chanSet(chansHist)(chans) (call it inv) as a hypothesis. This is because false positives propagate over transitions: there is a path on which inv fails for one or more steps (so the implication holds), and because the state machine was under-specified, we can then reach a state in which inv holds but the unidirectional property fails. To solve this problem, we use the *weak until* LTL operator W. Intuitively, W states that on any arbitrary path, its first argument either holds forever, or it holds in all states until its second argument holds, at which point neither need to hold.

Being forced to add the hypothesis can be a virtue: the environmental assumptions now appear in the theorem rather than being implicit in the state machine. The difference between assumptions being implicit in the model or explicit in the theorem is analogous to postulating assumptions as axioms in a theory or as hypotheses in a proof—a classic tradeoff made in mechanical theorem proving. SAL allows a verificationist to have the same freedoms in a model checking environment.

4. PRACTICAL INVARIANTS

Bounded model checkers have historically been used to find counterexamples, but they can also be used to prove invariants by induction over the state space [7]. SAL supports *k-induction*, a generalization of the induction principle, which can prove some invariants that may not be strictly inductive. The technique can be applied to both finite-state and infinite-state systems. In both cases, a bounded model checker is used. For infinite-state systems, the bounded model checker is combined with a *satisfiability modulo theories* (SMT) solver [8, 26]. For shorthand, I refer to infinite-state bounded model checking via k-induction as *inf-bmc* in the remainder of this paper.

The default SMT solver used by SAL is SRI's own Yices solver, which is a SMT solver for the satisfiability of (possibly quantified) formulas containing uninterpreted functions, real and integer linear arithmetic, arrays, fixed-size bit vectors, recursive datatypes, tuples, records, and lambda expressions [9]. Yices has regularly been one of the fastest and most powerful solvers at the annual SMT competitions [2].

4.1 Generalized Induction

To define k-induction, let $(S, I, \rightarrow)$ be a transition system where S is a set of states, $I \subseteq S$ is a set of initial states, and $\rightarrow$ is a binary transition relation. If k is a natural number, then a *k-trajectory* is a sequence of states $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_k$ (a 0-trajectory is a single state). Let k be a natural number, and let P be property. The k-induction principle is then defined as follows:

- *Base Case*: Show that for each k-trajectory $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_k$ such that $s_0 \in I$, $P(s_j)$ holds, for $0 \leq j < k$.
- *Induction Step*: Show that for all k-trajectories $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_k$, if $P(s_j)$ holds for $0 \leq j < k$, then $P(s_k)$ holds.

The principle is equivalent to the usual transition-system induction principle when $k = 1$. In SAL, the user specifies the depth at which to attempt an induction proof, but the attempt itself is automated.

For example, consider the following state machine defined in SAL:

```
counter1 : MODULE =
BEGIN
  OUTPUT cnt : INTEGER
  OUTPUT b : BOOLEAN
INITIALIZATION
  cnt = 0;
  b = TRUE;
TRANSITION
[   b    --> cnt' = cnt + 2;
             b' = NOT b
 [] ELSE --> cnt' = cnt - 1;
             b' = NOT b
]
END;
```

The module produces an infinite sequence of integers and boolean values. It's behavior is as follows:

cnt	:	0	2	1	3	2	4	...
b	:	*T*	*F*	*T*	*F*	*T*	*F*	...

Now suppose we wish to prove the following invariant holds:

```
Cnt1Clm : CLAIM counter1 |- G(cnt >= 0);
```

While `Cnt1Clm` is an invariant, it is not inductive (i.e., $k = 1$). To see why, consider the induction step, and consider (an unreachable) state in which `b` is false and `cnt` is zero. This state satisfies `Cnt1Clm`, but in one step, `cnt` equals -1, and the invariant fails. However, in any two steps (i.e., $k = 2$), the claim holds.

4.2 Disjunctive Invariants

Unfortunately, k-induction is exponential in the size of k, so at some point, an invariant will likely need to be manually strengthened. I find the method of building up invariants using *disjunctive invariants* [24] to be particularly suited to SAL. A disjunctive invariant is built up by adding disjuncts, each of which is an invariant for some system configuration. In SAL, disjunctive invariants can quickly be built up by examining the counterexamples returned by SAL in failed proof attempts. Disjunctive invariants contrast with the traditional approach of strengthening an invariant by adding *conjuncts*. Each conjunct in a traditional invariant needs to hold in every system configuration, unlike in a disjunctive invariant.

Consider the following simple example:

```
counter2 : MODULE =
BEGIN
  OUTPUT cnt : INTEGER
  OUTPUT b : BOOLEAN
INITIALIZATION
  cnt = 0;
  b = TRUE;
TRANSITION
[   b    --> cnt' = (-1 * cnt) - 1;
             b' = NOT b
 [] ELSE --> cnt' = (-1 * cnt) + 1;
             b' = NOT b
]
END;
```

The module produces an infinite sequence of integers and boolean values. It's behavior is as follows:

cnt	:	0	−1	2	−3	4	−5	...
b	:	*T*	*F*	*T*	*F*	*T*	*F*	...

Suppose we wished to prove an invariant that captures the behavior of the state machine. Rather than consider every configuration of the machine, we might begin with an initial approximation that only characterizes states in which `b` is true:

```
Cnt2Clm : CLAIM counter2 |- G(b AND cnt >= 0);
```

The conjecture fails. SAL automatically returns a counterexample in which both `b` is false and `cnt` is less than zero. Guided by the counterexample, we can now augment the original conjecture with a disjunct to characterize the configuration in which `b` is false.

```
Cnt2Clm : CLAIM counter2 |-
  G(   (b AND cnt >= 0)
    OR (NOT b AND cnt < 0));
```

The stated conjecture is proved by SAL. Of course, the example presented is quite simple, but the technique allows one to build up invariants of complex specifications in piecemeal fashion by considering only one configuration at a time and allowing SAL's counterexamples to show remaining configurations.

Using k-induction and disjunctive invariants, Geoffrey Brown and I were able to dramatically reduce the verification effort of physical-layer protocols, such as the Biphase Mark protocol (used in CD-player decoders, Ethernet, and Tokenring) and the 8N1 protocol (used in UARTs) [5]. The verification of BMP presented herein results in an orders-of-magnitude reduction in effort as compared to the protocol's previous formal verifications using mechanical theorem proving. Our verification required 3 invariants, whereas a published proof using the mechanical theorem prover PVS required 37 [28]. Using infinite-bmc induction, proofs of the 3 invariants were completely automated, whereas the PVS proof initially required some 4000 user-supplied proof directives, in total. Another proof using PVS is so large that the tool required 5 hours just to *check* the manually-generated proof whereas the SAL proof is generated automatically in seconds [13]. BMP has also been verified by J. Moore using Nqthm, a precursor to ACL2, requiring a substantial proof effort (Moore cites the work as being one of the "best ideas" of his career) [18].[6] Geoffrey and I spent only a couple of days to obtain our initial results; much more time was spent generalizing the model and writing up the results.

In Section 5.1, we describe techniques to exploit k-induction effectively.

[6] `http://www.cs.utexas.edu/users/moore/best-ideas/`.

5. COMPOSITION

Similar to SMV [16] and other model checkers, SAL allows state machines to be composed both synchronously and asynchronously, and a composed state machine may contain both synchronously and asynchronously composed subcompositions. For example, supposing that `A`, `B`, `C`, and `D` were modules, the following is a valid composition (assuming the modules satisfy the variable-update constraints for synchronous composition mentioned in Section 2):

```
E : MODULE = (A [] B) [] (C || D);
```

The judicious use of synchronous composition can simplify specifications and ease the verification effort, taking further advantage of SAL's tools.

In the following, I first provide an example emphasizing how to use synchronous composition to reduce the proof effort for k-induction. The second emphasizes how to judiciously use composition to decompose environmental constraints from the state machine itself, allowing for simple specification refinements.

5.1 Cheap Induction Proofs

In proofs by k-induction (described in Section 4), k specifies the length of trajectories considered in the base case and induction step. With longer trajectories, weaker invariants can be proved. Unfortunately, the cost of k induction is exponential in the value of k, since a SAT-solver is used to unroll the transition relation. Thus, models that reduce unessential interleavings make k-induction proofs faster.

Let me give a simple example explaining this technique first and then describe how I used it in an industrial verification. Recall the module `counter1` from Section 4.1. We will modify its transitions slightly so that it deadlocks when the counter is greater than 2 (the sole purpose of which is to avoid dealing with fairness conditions in the foregoing state-machine composition):

```
   b AND cnt <= 2       --> cnt' = cnt + 2;
                            b' = NOT b
[] (NOT b) AND cnt <= 2 --> cnt' = cnt - 1;
                            b' = NOT b
```

The behavior of the generated state machine is as follows:

cnt	:	0	2	1	3	(deadlock)
b	:	*T*	*F*	*T*	*F*	(deadlock)

Now suppose we wish to prove that the `cnt` variable is always nonnegative.

```
cntThm : THEOREM nodes |- G(cnt >= 0);
```

This property is k-inductive for $k = 2$ (for the reasons mentioned in Section 4.1). Now we are going to compose some instances of the `node` module together. SAL provides some

convenient syntax for doing this, but first, we must parameterize the above module. We being by defining an index type `[1..I]` denoting the sequence 1, 2, ..., I.

```
I : NATURAL = 5;
INDICES : TYPE = [1..I];
```

Now we modify the declaration of the `node` module from

```
node : MODULE =
BEGIN
  ...
```

as above to a module parameterized by indices:

```
node[i: INDICES]: MODULE =
BEGIN
  ...
```

where the remainder of the module declaration is exactly as presented above. Now we can automatically compose instances of the module together with the following SAL declaration (an explanation of the syntax follows):

```
nodes : MODULE =
  WITH OUTPUT cnts : ARRAY INDICES OF INTEGER
    (|| (i : INDICES) : RENAME cnt TO cnts[i]
                        IN node[i]);
```

The module `nodes` is the synchronous composition of instances of the `node` module, one for each index in `INDICES`. The `nodes` module has only one state variable, `cnts`. This variable is an array, and its values are taken from the `cnt` variables in each module. Thus, `cnts[j]` is value of `cnt` from the jth node module.

We can modify slightly the theorem proved about a single module to cover all of the modules in the composition:

```
cntsThm : CLAIM nodes |-
  G(FORALL (i : INDICES) : cnts[i] >= 0);
```

The theorem is proved using k-induction at $k = 2$. Indeed, in the synchronous composition, it is proved for $k = 2$ for any number of nodes (i.e., values of `I`). I proved `cntsThm` for two through thirty nodes on a PowerBook G4 Mac with one gigabyte of memory, and the proofs all took about 1-2 seconds.

On the other hand, if we change the synchronous composition in the `nodes` module above to an asynchronous composition,[7] the cost increases exponentially. Intuitively, we have to increase the value of k to account for the possible interleavings in which each module's `cnt` and `b` variables are updated. For two nodes (`I` = 2), proving the theorem `cntsThm` requires at minimum $k = 6$. On the same PowerBook, the proof takes about one second. For `I` = 3, the proof requires $k = 10$, and the proof takes about three and one-half seconds. For `I` = 4, we require $k = 14$ and the proof takes just over 10 minutes; for `I` = 5, we require $k = 18$, and I stopped running the experiment after five hours of computation!

This technique has a practical aspect. To verify a reintegration protocol in SAL using inf-bmc, I used these techniques [21]. A *reintegration protocol* is a protocol designed for fault-tolerant distributed systems—in particular, I verified a reintegration protocol for SPIDER, a time-triggered fly-by-wire communications bus being designed at NASA Langley [27]. The protocol increases system survivability by allowing a node that has suffered a transient fault (i.e., it is not permanently damaged) to regain state consistent with the non-faulty nodes and reintegrate with them to deliver system services.

In the model of the system, I initially began with a highly-asynchronous model. I realized, however, that much of the asynchronous behavior could be synchronous without affecting the fidelity of the model. One example is the specification of non-faulty nodes (or *operational nodes*) being observed by the reintegrating node. In this model, their executions are independent of each other, and their order of execution is not relevant to the verification (we do not care which operational node executes first, second, etc., but only that each operational node executes within some window). Thus, we can update the state variables of each operational node synchronously. Each maintains a variable ranging over the reals (its timeout variable—see Section 3.1) denoting at what real-time it is modeled to execute, but their transitions occur synchronously.

An anonymous reviewer of this report noted that the technique appears to be akin to a partial-order reduction [6] applied to inf-bmc, and asked if such a reduction could be realized automatically. For the simple example presented, I believe it would be possible to do so, but as far as I know, generalizations would be an open research question.

To summarize, while synchronous and asynchronous composition are not unique to SAL, their impact on k-induction proofs is a more recent issue. Since k-induction is especially sensitive to the lengths of trajectories to prove an invariant, synchronous composition should be employed when possible.

5.2 Environmental Decomposition

Another use of synchronous composition is to decompose the environment model from the system model. The purpose of an environmental model is to constrain the behavior of a system situated in that environment. In the synchronous composition of modules `A` and `B`, if either module deadlocks, the composition `A || B` deadlocks. Thus, environmental constraints can be modeled by having the environment deadlock the entire system on execution paths outside of the environmental constraints.

[7] The theorem `cntsThm` would have had to include fairness constraints if the asynchronously-composed modules did not deadlock after some number of steps.

For example, Geoffrey Brown and I used this approach in the verification and refinement of physical-layer protocols [3]. Physical-layer protocols are cooperative protocols between a transmitter and a receiver. The transmitter and receiver are each hardware devices driven by separate clocks. The goal of the protocols is to pass a stream of bits from the transmitter to the receiver. The signal must encode not only the bits to be passed but the transmitter's clock signal so that the receiver can synchronize with the transmitter to the extent necessary to capture the bits without error.

In the model, we specify three modules: a transmitter (`rx`), a receiver (`tx`), and a constraint module (`constraint`) simulating the environment. The entire system is defined as the composition of three modules, where the transmitter and receiver are asynchronously composed, and the constraint module is synchronously composed with the entire system:

```
system : MODULE = (tx [] rx);
systemEnv : MODULE = system || constraint;
```

In this model, the constraint module separates out from the system the effects of *metastability*, a phenomenon in which a flip flop (i.e., latch) does not settle to a stable value (i.e., "1" or "0") within a clock cycle. Metastability can arise when a signal is asynchronous (in the hardware-domain sense of the word); that is, it passes between clock regions. One goal of physical-layer protocols is to ensure that the probability of metastable behavior is sufficiently low.

In the module `rx`, the receiver's behavior is under-specified. In particular, we do not constrain the conditions under which metastability may occur. The receiver captures the signal sent with the boolean variable `rbit`. The receiver is specified with a guarded transition like the following (the guard has been elided) allowing `rbit` to nondeterministically take a value of `FALSE` or `TRUE`, regardless of the signal sent to it by the transmitter:

```
... -->  rbit' IN {FALSE, TRUE};
```

The under-specified receiver is constrained by its environment. The constraint module definition is presented below, with extraneous details (for the purposes of this discussion) elided.

```
constraint : MODULE =
    ...
DEFINITION
  stable = NOT changing OR tclk - rclk < TSTABLE;
    ...
TRANSITION
   rclk' /= rclk AND (stable => rbit' = tdata) -->
[] ...
```

In the module, we define the value of the state variable `stable` to be a fixed function of other state variables (modeling the relationship between the transmitter's and receiver's respective clocks). The variable `stable` captures the sufficient constraints to prevent metastability. We give a representative transition in the constraint module. The transition's guard is a conjunction. The first condition holds if the receiver is making a transition (the guard states that the reciever's clock is being updated—this is a timeout automata model, as mentioned in Section 3.1). The second conjunct enforces a relation between the signal the transmitter sends (`tdata`) and the value captured by the receiver (`rbit`): if `stable` holds, then the receiver captures the signal. In other words, the constraint module prunes the execution paths allowed by the `system` module alone in which the value of `rbit` is completely nondeterministic. Finally, note that because no state variables follow `-->` in the transition, no state variables are updated by the environment.

So what are the benefits of the decomposition? One example is refinement. Brown and I wished to refine the physical-layer protocols we specified. These protocols are real-time protocols. Unfortunately, we could not easily compose the real-time specifications with synchronous (i.e., finite-state) hardware specifications of the transmitter and the receiver. Doing so would require augmenting the invariant about real-time behavior with invariants about the synchronous hardware. Ideally, we could decompose the correctness proof of the protocol with the correctness proofs of the hardware in the transmitter and receiver, respectively.

Therefore, we developed a more abstract finite-state, discrete-time model of the protocols. The finite-state model could be easily composed with the other finite-state specifications of the synchronous hardware within a single clock domain; i.e., the transmitter's encoder could be composed with a specification of the remainder of the transmitter's hardware, and the receiver's decoder could be composed with a specification of the remainder of the receiver's hardware. The entire specification could then be verified using a conventional model checker, like SAL's BDD-based model checker.

The goal was to carry out a *temporal refinement* to prove that the real-time implementation refined the discrete-time specification. Using inf-bmc, we verified the necessary refinement conditions. These conditions demonstrate that the implementation is more constrained (i.e., has fewer behaviors) than its specification along the lines of Abadi and Lamport's classic refinement approach [1].

In SAL, most of the refinement was "for free." For example, recall that a synchronous composition `A || B` constrains the possible behaviors of module `A` and `B`. Thus, by definition, `A || B` is a refinement of `A`.

Thus, to prove that the real-time model

```
system : MODULE = (tx [] rx);
systemEnv : MODULE = (tx [] rx) || constraint;
```

refines the discrete-time model

```
system_dt : MODULE = tx_dt [] rx_dt;
```

(where `dt` stands for "discrete time"), we simply had to prove that `tx` refines `tx_dt` and that `rx` refines `rx_dt`. We did not have to refine the `constraint` module, as one would intuitively expect, since it is orthogonal to the system itself.

6. THE MARRIAGE OF MODEL CHECKING AND THEOREM PROVING

Sometimes, even the powerful tools provided by SAL are not enough. In this section, I describe three ways in which I have used SAL in tandem with a mechanical theorem prover to take advantage of the best of both worlds. The two examples include using SAL to discover counterexamples to failed proof conjectures and verifying a theory of real-time systems in a mechanical theorem prover, and then using SAL to prove that an implementation satisfies constraints from the theory.

6.1 Counterexample Discovery

Sometimes verifications require the full interactive reasoning-power of mechanical theorem proving. This is the case when, for example, the specification or proof involves intricate mathematics (that does not fall within a decidable theory), or the specification is heavily parameterized (e.g., proving a distributed protocol correct for an arbitrary number of nodes).

Although rarely discussed in the literature, most attempts to prove conjectures using interactive mechanical theorem proving fail. Only after several iterations of the following steps

1. attempting to prove a theorem,
2. then discovering the theorem is false or the proof is too difficult,
3. revising the specification or theorem accordingly,
4. and repeating from Step 1

is a theorem finally proved.

Provided the theorem prover is sound and the conjecture is not both true and unprovable—a possibility in mathematics—there are two possible reasons for a failed proof attempt. First, the conjecture may be true, but the user lacks the resources or insight to prove it. Second, the conjecture may be false. It can be difficult to determine which of these is the case.

Proofs of correctness of algorithms and protocols often involve nested case-analysis. A proof obligation that cannot be completed is often deep within the proof, where intuition about the system behavior—and what constitutes a counterexample—wanes. The difficulty is also due to the nature of mechanical theorem proving. The proof steps issued in such a system are fine-grained. Formal specifications make explicit much of the detail that is suppressed in informal models. The detail and formality of the specification and proof makes the discovery of a counterexample more difficult.

Paul Miner, Wilfredo Torres-Pomales, and I ran against this very problem when trying to verify the correctness of a fault-tolerant protocol for a distributed real-time fault-tolerant

```
[-1]  good?(r_status!1(r!1))
[-2]  asymmetric?(b_status!1(G!1))
[-3]  IC_DMFA(b_status!1, r_status!1, F!1)
[-4]  all_correct_accs?(b_status!1, r_status!1, F!1)
  |-------
[1]   trusted?(F!1`BR(r!1)(G!1))
[2]   declared?(F!1`BB(b2!1)(G!1))
{3}   (FORALL (p_1: below(R)):
         (trusted?(F!1`RB(b1!1)(p_1)) =>
            NOT asymmetric?(r_status!1(p_1))))
       &
       (FORALL (p_1: below(R)):
          (trusted?(F!1`RB(b2!1)(p_1)) =>
             NOT asymmetric?(r_status!1(p_1))))
[4]   declared?(F!1`BB(b1!1)(G!1))
[5]   robus_ic(b_status!1, r_status!1,
                F!1`BB(b1!1)(G!1), F!1`RB(b1!1))
               (G!1, msg!1, b1!1)
       =
       robus_ic(b_status!1, r_status!1,
                F!1`BB(b2!1)(G!1), F!1`RB(b2!1))
               (G!1, msg!1, b2!1)
```

Figure 1: Unproved PVS Sequent

bus [22]. The protocol we were verifying was an interactive consistency protocol designed for NASA's SPIDER architecture [27]. We were verifying the protocol in PVS.

The protocol suffered a bug in its design: the bug occurrs if two Byzantine faults [15] (allowing unconstrained faulty behavior) occur simultaneously. Such an occurrence is a rare pathological case that escaped our pencil-and-paper analysis.

During the course of formally verifying the protocol, Torres-Pomales independently discovered the bug through "engineering insight." Nevertheless, as a case-study in distilling counterexamples from a failed proof, we decided to press on in the proof until a single leaf in the proof tree remained. To give the reader an idea about what the unproven leaf looked like, we present the PVS sequent if Figure 1 (it is described in detail elsewhere [22]).

The unproven leaf, however, does not give a good idea as to whether a counterexample actually exists and if one does, what that counterexample is. Therefore, building on the specification and verification of a similar protocol done by John Rushby in SAL [25], we formulated the unproven leaf as an LTL conjecture in SAL (Figure 2) stating that in all states, the unproven leaf from the PVS specification indeed holds.

Note the similarity between the PVS sequent and the SAL conjecture afforded by the expressiveness of SAL's language. The main difference between the sequent and conjecture are the use of arrays in SAL rather than curried functions in PVS and that the number of nodes are fixed in the finite-state SAL specification.

For a fixed number of nodes, SAL easily returns a concrete counterexample showing how a state can be reached in which the theorem is false.

```
counterex : CLAIM system |-
  G(   (pc = 4 AND
        r_status[1] = good AND
        G_status = asymmetric AND
        IC_DMFA(r_status, F_RB, F_BR, G_status) AND
        all_correct_accs(r_status, F_RB,
                         G_status, F_BR, F_BB))
    =>
       (F_BR[1] = trusted OR
        F_BB[2] = declared OR
        ((FORALL (r: RMUs): F_RB[1][r] = trusted =>
              r_status[r] /= asymmetric)
         AND
         (FORALL (r: RMUs): F_RB[2][r] = trusted =>
              r_status[r] /= asymmetric)) OR
        F_BB[1] = declared OR
        robus_ic[1] = robus_ic[2]));
```

Figure 2: SAL Formulation in LTL of the Unproved Sequent

While our case-study highlights the benefit of interactivity between model checking and theorem proving, further work is required. The case-study suffers at least the following shortcomings:

- The approach is too interactive and onerous. It requires manually specifying the protocol and failed conjecture in a model checker and manually correcting the specification in the theorem prover.
- The approach depends on the counterexample being attainable with instantiated parameters that are small enough to be model checked. As pointed out by an anonymous reviewer of this report, that the counterexample was uncovered with small finite values accords with Daniel Jackson's "small scope hypothesis" [14]. For the case presented, we could have uncovered the error through model checking alone, but our goal was to prove the protocols correct for any instantiation of the parameters, as we were in fact able to do, once the protocol was mended [17].
- We would like a more automated approach to verify the parameterized protocol specification in the first place than is possible using mechanical theorem proving alone.

A more automated connection between PVS and SAL would be a good start to satisfying many of these desiderata.

6.2 Real-Time Schedule Verification

In this final example, I used PVS to specify and verify a general mathematical theory, and then I used SAL to automatically prove that various hardware realizations satisfied the theory's constraints. Specifically, I used PVS to extend a general theory about time-triggered systems. *Time-triggered systems* are distributed systems in which the nodes are independently-clocked but maintain synchrony with one another. Time-triggered protocols depend on the synchrony assumption the underlying system provides, and the protocols are often formally verified in an untimed or synchronous model based on this assumption. An untimed model is simpler than a real-time model, but it abstracts away timing assumptions that must hold for the model to be valid.

John Rushby developed a theory of time-triggered system in PVS [23]. The central theorem of that work showed that under certain constraints, a time-triggered system simulates a synchronous system. Some of the constraints (or axioms, as they were formulated in PVS) were inconsistent. I mended these inconsistencies and extended the theory to include a larger class of time-triggered protocols [20].

A theorem prover was the right tool for this effort, as the theory is axiomatic and its proofs rely on a variety of arithmetic facts (e.g., properties of floor and ceiling functions, properties of absolute values, etc.). Furthermore, using PVS, I could formally prove the theory consistent by showing a model exists for it using theory interpretations, as implemented in PVS [19].

Once the theory was developed, I wished to prove that specific hardware schedules satisfied the real-time constraints of the theory. To do so, I essentially lifted the schedule constraints (i.e., the axioms) from the PVS specification into SAL, given the similarity of the languages. Then I built a simple state machine that emulated the evolution of the hardware schedules through the execution of the protocol. I finally proved theorems stating that in the execution of the state machine, the constraints are never violated. This verification also used inf-bmc (Section 4), since the constraints were real-time constraints.

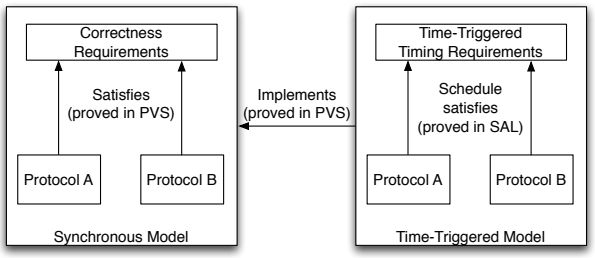

Figure 3: Time-Triggered Protocol Verification Strategy

The upshot of the approach is a formally-verified connection between the untimed specification and the hardware realization of a time-triggered protocol with respect to its timing parameters, as shown in Figure 3.

Stepping back, the above approach is an example of the judicious combination of mechanical theorem proving and model checking. While theorem proving requires more human guidance, it was appropriate for formulating and verifying the theory of time-triggered systems because the theory required substantial mathematical reasoning, and we only have to develop the theory once. To prove the timing characteristics of the implementations are correct, model checking was appropriate because the proofs can be automated, and the task must be repeated for each implementation or

optimization for a single implementation.

7. CONCLUSION

My goal in this paper was to advocate for and demonstrate the utility of the advanced features of SAL. I hope this report serves as a "cookbook" of sorts for the uses of SAL I have described.

In addition to the features and techniques I have demonstrated herein, other applications have been developed. As one example, Grégoire Hamon, Leonardo de Moura, and John Rushby prototyped a novel automated test-case generator in SAL [11]. The prototype is a few-dozen line Scheme program that calls the SAL API for its model checkers. Still other uses can be found on the SAL wiki at `http://sal-wiki.csl.sri.com/index.php/Main_Page`, which should continue to provide the community with additional SAL successes.

Acknowledgments

Many of the best ideas described herein are from my coauthors. I particularly thank Geoffrey Brown for his fruitful collaboration using SAL. I was first inspired to use SAL from attending Bruno Dutertre's seminar at the National Institute of Aerospace. John Rushby's SAL tutorial [25] helped me enormously to learn to exploit the language. I received detailed comments from the workshop's anonymous reviewers and from Levent Erkök at Galois, Inc. Much of the research cited herein was completed while I was a member of the NASA Langley Research Center Formal Methods Group.

8. REFERENCES

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[2] C. Barrett, L. de Moura, and A. Stump. Design and results of the 2nd satisfiability modulo theories competition (SMT-COMP 2006). *Formal Methods in System Design*, 2007. Accepted June, 2007. To appear. A preprint is available at `http://www.smtcomp.org/`.

[3] G. Brown and L. Pike. Temporal refinement using smt and model checking with an application to physical-layer protocols. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE'2007)*, pages 171–180. OmniPress, 2007. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/refinement.html`.

[4] G. M. Brown and L. Pike. Easy parameterized verification of biphase mark and 8N1 protocols. In *The Proceedings of the 12th International Conference on Tools and the Construction of Algorithms (TACAS'06)*, pages 58–72, 2006. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/bmp.html`.

[5] G. M. Brown and L. Pike. "easy" parameterized verificaton of cross domain clock protocols. In *Seventh International Workshop on Designing Correct Circuits DCC: Participants' Proceedings*, 2006. Satellite Event of ETAPS. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/dcc.html`.

[6] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[7] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In R. Alur and D. Peled, editors, *Computer-Aided Verification, CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500, Boston, MA, July 2004. Springer-Verlag.

[8] L. de Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification. In *Computer-Aided Verification, CAV'03*, volume 2725 of *LNCS*, 2003.

[9] B. Dutertre and L. de Moura. Yices: an SMT solver. Available at `http://yices.csl.sri.com/`, August 2006.

[10] B. Dutertre and M. Sorea. Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 3253 of *Lecture Notes in Computer Science*, pages 199–214, Grenoble, France, Sept. 2004. Springer-Verlag. Available at `http://fm.csl.sri.com/doc/abstracts/ftrtft04`.

[11] G. Hamon, L. deMoura, and J. Rushby. Generating efficient test sets with a model checker. In *2nd International Conference on Software Engineering and Formal Methods*, pages 261–270, Beijing, China, Sept. 2004. IEEE Computer Society.

[12] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[13] D. V. Hung. Modelling and verification of biphase mark protocols using PVS. In *Proceedings of the International Conference on Applications of Concurrency to System Design (CSD'98), Aizu-wakamatsu, Fukushima, Japan, March 1998*, pages 88–98. IEEE Computer Society Press, 1998.

[14] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[15] Lamport, Shostak, and Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, July 1982. Available at `http://citeseer.nj.nec.com/lamport82byzantine.html`.

[16] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[17] P. Miner, A. Geser, L. Pike, and J. Maddalon. A unified fault-tolerance protocol. In Y. Lakhnech and S. Yovine, editors, *Formal Techniques, Modeling and Analysis of Timed and Fault-Tolerant Systems (FORMATS-FTRTFT)*, volume 3253 of *LNCS*, pages 167–182. Springer, 2004. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/unified.html`.

[18] J. S. Moore. A formal model of asynchronous communication and its use in mechanically verifying a biphase mark protocol. *Formal Aspects of Computing*, 6(1):60–91, 1994.

[19] S. Owre and N. Shankar. Theory interpretations in PVS. Technical Report SRI-CSL-01-01, SRI, International, April 2001. Available at `http://pvs.csl.sri.com/documentation.shtml`.

[20] L. Pike. Modeling time-triggered protocols and verifying their real-time schedules. In *Proceedings of Formal Methods in Computer Aided Design (FMCAD'07)*. IEEE, 2007. Available at `http://www.`

cs.indiana.edu/~lepike/pub_pages/fmcad.html. To appear.

[21] L. Pike and S. D. Johnson. The formal verification of a reintegration protocol. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 286–289, New York, NY, USA, 2005. ACM Press. Available at http://www.cs.indiana.edu/~lepike/pub_pages/emsoft.html.

[22] L. Pike, P. Miner, and W. Torres. Model checking failed conjectures in theorem proving: a case study. Technical Report NASA/TM–2004–213278, NASA Langley Research Center, November 2004. Available at http://www.cs.indiana.edu/~lepike/pub_pages/unproven.html.

[23] J. Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. *IEEE Transactions on Software Engineering*, 25(5):651–660, September 1999.

[24] J. Rushby. Verification diagrams revisited: Disjunctive invariants for easy verification. In E. A. Emerson and A. P. Sistla, editors, *Computer-Aided Verification, CAV '2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 508–520, Chicago, IL, July 2000. Springer-Verlag. Available at http://www.csl.sri.com/users/rushby/abstracts/cav00.

[25] J. Rushby. SAL tutorial: Analyzing the fault-tolerant algorithm OM(1). Technical Report CSL Technical Note, SRI International, 2004. Available at http://www.csl.sri.com/users/rushby/abstracts/om1.

[26] J. Rushby. Harnessing disruptive innovation in formal verification. In *4th IEEE International Conference on Software Engineering and Formal Methods (SEFM)*. IEEE Computer Society, 2006. Available at http://www.csl.sri.com/users/rushby/abstracts/sefm06.

[27] W. Torres-Pomales, M. R. Malekpour, and P. Miner. ROBUS-2: A fault-tolerant broadcast communication system. Technical Report NASA/TM-2005-213540, NASA Langley Research Center, 2005.

[28] F. W. Vaandrager and A. L. de Groot. Analysis of a Biphase Mark Protocol with Uppaal and PVS. Technical Report NIII-R0455, Nijmegen Institute for Computing and Information Science, 2004.