Proceedings of the 2003 Winter Simulation Conference
S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, eds.

FROM SIMULATION TO VERIFICATION (AND BACK)*

Harald Rueß

Computer Science Laboratory
SRI International
Menlo Park, CA 94303, U.S.A.

Leonardo de Moura

Computer Science Laboratory
SRI International
Menlo Park, CA 94303, U.S.A.

ABSTRACT

Symbolic evaluation is the execution of software and software designs on inputs given as symbolic or explicit constants along with constraints on these inputs. Efficient symbolic evaluation is now feasible due to recent advances in efficient decision procedures and symbolic model checking. Symbolic evaluation can be applied to partially implemented descriptions and provides wider coverage and greater assurance than testing and traditional simulation alone. Unlike full formal verification, symbolic evaluation can be used in a partial manner that is more likely to succeed and yield some degree of assurance. Its main advantage is that it can be used within a smooth spectrum of analyses ranging from refutation based on explicit-state simulation to full-blown verification.

1 INTRODUCTION

Symbolic evaluation is the execution of a program (or even a specification) where some or all of the inputs are given in symbolic form. Symbolic evaluation is a basic technique in theorem proving and verification. For example, a greatest-common divisor (GCD) algorithm returns a common divisor can be verified by symbolically evaluating the GCD operation and showing that any common divisor for x and y is also a common divisor for y and $x - y$, for $x > y > 0$.

Symbolic evaluation has been especially successful for hardware designs (for example, (Hardin et al. 1998; Moore 1998)), but it is also effective for the verification of the correctness of compilation steps, in ensuring the safety of bytecode, and for checking that certain invariants are preserved. There are many other examples of the use of symbolic evaluation. For example, an interval analysis of a program can be carried out by symbolically computing the fixed points of the intervals that capture the range of the numeric variables. A simpler form of such analysis has been applied to the Ariane-5 launch control software since the initial debacle. A sorting program can be examined over a bounded size array to see if the output is indeed a sorted permutation of the input. This is obviously weaker than verifying the sorting program over arrays of arbitrary size, but perhaps more efficient at uncovering bugs.

Symbolic evaluation includes *testing* but has some added advantages. Most importantly, testing provides only partial coverage and yields very limited confidence in the correctness of the design, whereas symbolic simulation provides increased *coverage* since a symbolic evaluation covers a substantial range of concrete inputs. In addition, symbolic simulation does not require a full implementation and can be driven off a partial implementation or a high-level specification. It also provides increased coverage since a symbolic evaluation covers a substantial range of concrete inputs. Also, symbolic evaluation can be applied not only in the forward direction but also in the backward direction for computing preconditions from postconditions.

Symbolic evaluation is a key component of any useful verification system, and has been a standard part of theorem proving since the work of Boyer and Moore (Boyer and Moore 1979). Its main advantage is that it is largely automatic and can be used within a smooth spectrum of analyses ranging from testing to verification. In contrast, formal verification tends to be an all-or-nothing enterprise that yields few partial results, and is therefore not yet an economically viable technique for routine use.

Symbolic evaluation is very effective for essentially finite-state programs. For example, symbolic trajectory evaluation carries out symbolic simulation of hardware in a ternary domain of truth values with an unknown element. Model checking is a well-established technique for formal verification of reactive systems such as hard-

*This research was supported by NASA Langley Research Center Cooperative agreement NCC-1-399 under a subcontract from Honeywell, by NSA (Maryland Procurement Office) under Contract MDA904-02-C-1196, and by NSF under Contract EIA-0224465.

ware circuits and communication protocols. Systems are modeled as state machines and the specification is expressed in temporal logic. The reachable state space of a simple protocol, resource control algorithm, or hardware can be fully explored in symbolic terms, using a symbolic model checker (Burch et al. 1992; McMillan 1993b). Model checking techniques for reachability can also be used for some infinite state systems such as those with timers (Alur et al. 1993), hybrid combinations of discrete and continuous behavior (Alur et al. 1995), and data structures such as queues (Godefroid and Long 1996) and stacks (Abdulla et al. 1999). Abstraction can be used to reduce the symbolic evaluation of infinite-state systems to finite-state systems through the use of abstract interpretation (Clarke et al. 1994; Loiseaux et al. 1995; Saïdi and Graf 1997; Saïdi and Shankar 1999).

Bounded model checking (BMC) can be viewed as a restricted form of symbolic simulation in that the search for falsifying traces is restricted to traces of some given length (Clarke et al. 2001). The BMC problem can efficiently be reduced to a propositional satisfiability problem, and off-the-shelf propositional satisfiability (SAT) checkers are used to construct counterexamples from satisfying assignments. In this way, BMC extends ideas for using SAT checkers to generate plans (as witnesses of eventually reaching some goal) (Kautz and Selman 1992). Experience demonstrates that BMC can be effective for falsification in cases where there exist short falsifying traces (Clarke et al. 2001; Copty et al. 2001).

In deductive verification, the *invariance rule* for establishing invariance properties requires a 1-step symbolic simulation for establishing that a given safety property (one true of all reachable states) is indeed preserved on all transitions (Manna and Pnueli 1995). Application of the invariance rule usually requires creativity in coming up with a sufficiently strong inductive invariant. It is also nontrivial to detect bugs from failed induction proofs. Recent generalizations based on k-step symbolic simulation try to overcome these limitations (de Moura et al. 2002).

This concludes our brief, and necessarily incomplete, overview of the landscape of formal verification techniques based on symbolic simulation. These methods range from refutation and simulation-based methods to full-blown verification.

In the rest of this paper we explore these validation techniques and their relative merits in some more detail. As our running example, we formally model a priority-ceiling real-time scheduler and formally establish that certain deadlines are always met. For these experiments, we use SRI's SAL verification toolbox, which includes a powerful modeling language for specifying computational systems in a modular way. The

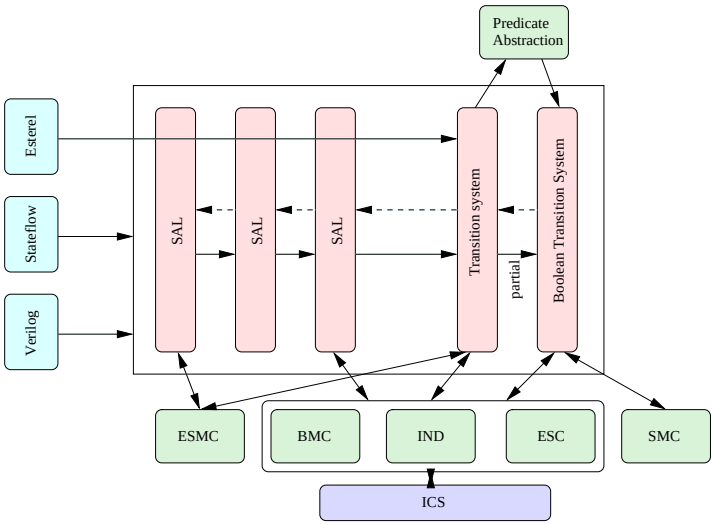

Figure 1: SAL Toolbus

SAL framework also integrates a number of validation and verification tools such as a slicer, an explicit-state simulator, a BDD-based, symbolic model checker, a bounded model checker for infinite-state systems based on a combination of propositional SAT solving and constraint solving, and an induction engine that combines refutation based on BMC with verification based on k-induction.

2 SYMBOLIC ANALYSIS LABORATORY

We have already seen a catalog of symbolic analysis techniques. The idea of *symbolic analysis* is to allow these techniques to coexist so that the analysis of a transition system can be carried out by successive applications of a combination of these techniques. SAL is such 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 (Bensalem et al. 2000). SAL provides a blackboard architecture for symbolic analysis where a collection of tools interact through a common intermediate language for transition systems. The individual analyzers (theorem provers, model checkers, static analyzers) are driven from this language, and the analysis results fed back to this intermediate level. This language also serves as the *target* for translators that extract the transition system description for popular programming languages such as Esterel, Java, and Stateflow (see Figure 1). An earlier overview of SAL can be found in (Bensalem et al. 2000), the SAL language is documented in (Dill et al. 2001), and the rationale behind symbolic analysis is explained in (Shankar 2000).

The SAL tools are available free of charge for noncommercial use at <sal.csl.sri.com>.

2.1 THE SAL LANGUAGE

A key part of the SAL framework is a language for describing transition systems. A variety of languages such as UNITY (Chandy and Misra 1988), SMV (McMillan 1993a), and *Reactive Modules* (Alur and Henzinger 1996) have been proposed in the literature, which are suitable for specifying transition systems. SAL has a lot in common with these languages, but it is also unique in that it includes a rich set of combinators for specifying large systems in a modular way.

A *module* is a self-contained specification of a transition system in SAL. Such 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, and invariant definitions. 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. Usually, several modules are collected in a context. Contexts also include type and constant declarations.

The scheduler module below, for example, receives a command as input and, depending on the values of the local variables, it decides on the next value of the output variable `turn`.

```
scheduler: MODULE =                                    1
BEGIN
  LOCAL clock : ClockRange
  LOCAL dispatch : ARRAY JobIdx OF ClockRange
  LOCAL  job_state : JobState
  OUTPUT turn : Turn
  LOCAL  rsrc : RSRC
  INPUT  cmd : Command
  INITIALIZATION
    ...
  TRANSITION
    ...
END
```

The definition of datatypes such as `ClockRange` and `Command`, the initial settings of variables, and transitions in terms of guarded commands are omitted here (for a more detailed description, see Figure 3).

Parametric modules allow the use of logical (state-independent) and type parameterization in the definition of modules. Most importantly, modules in SAL can be combined both synchronously and asynchronously.

The SAL language has been developed in collaboration with Stanford, Berkeley, Verimag, and SRI International.

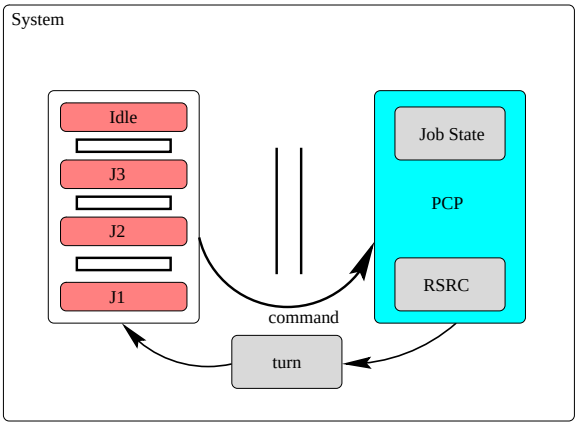

Figure 2: PCP Architecture

In the synchronous form of composition, modules react to inputs in zero time, as with combinational circuitry in hardware. Absence of causal loops in synchronous systems is ensured by generating proof obligations, rather than by more restrictive syntactic methods as in other languages. Asynchronously composed modules that are driven by independent clocks are modeled by means of interleaving the atomic transitions of the individual modules. SAL allows for mixtures of synchronous and asynchronous module composition. For example, it is natural to model a scheduler synchronously (`||`) composed with a set of jobs running asynchronously `[]` as depicted in Figure 2.

2.2 THE SAL VALIDATION TOOLBUS

The core of the SAL validation tools is a scriptable state space exploration toolkit for traversing state spaces associated with SAL specifications. Using the API of this toolkit, model checkers, simulators, static debuggers, symbolic simulators, and other state explorations can be encoded as Scheme scripts. For efficiency, these extensions are then compiled and linked with the SAL kernel.

SAL validation tools are not necessarily required to support the complete SAL language, as there is a staged translation of SAL into simpler fragments by source to source transformations (see Figure 1). These transformations include expression simplification, Skolemization of universally quantified expressions, and the expansion of module combinators. Finite-state SAL specifications, for example, are compiled into a Boolean transition system (circuit, net list) by converting state variables into bitvectors and abstractly interpreting operators in terms of bitvector expressions. The selection of verification tools below is a snapshot of the currently

available ones, but new verification tools can be added to the SAL toolbus due to its open-ended nature.

SAL-ESMC. Given a SAL module and a *linear temporal logic* (LTL) formula, the SAL explicit-state model checker translates the LTL formula into a SAL module for representing the associated Büchi automaton, which is then used as a *synchronous oberserver* for the system under consideration. Now, the given state space is explored for violations of the specified temporal logic formula, and a *counterexample* in the form of an execution path leading to such a violation, is constructed. In this way, ESMC can be seen as a standard simulator, but for the richness of the SAL language, which includes primed variables in the guard of transitions, simulation requires online scheduling. SAL-ESMC uses many of the optimizations for explicit-state simulators such as supertrace reduction (Holzmann 1998). Other popular techniques for dealing with the state explosion problem are partial order and symmetry reduction.

SAL-ESMC is in particularly useful in the initial steps of developing a model, since it detects many errors quickly. SAL-ESMC is rarely used for full verification, however, since even on finite-state systems, an enumerative check is unlikely to succeed because the size of the searchable state space can be exponential in the size of the program state. Still, enumerative model checking is an effective debugging or refutation technique that can often detect and display simple counterexamples when a property fails.

SAL-SMC. Given a SAL module of finite state space and an LTL formula, the SAL symbolic model checker decides whether the corresponding transition system indeed satisfies the formula. In the tradition of the SMV model checker, the finite transition relation is encoded using *binary decision diagrams* (BDDs), and symbolic simulation is realized by fixpoint computations on the BDD representations. SAL-SMC supports both forward and backward simulation.

Symbolic model checkers using BDD representations can sometimes process state spaces with more than 10^{1000} states. The problem, however, is that the size of the BDD representations may also explode during fixpoint computation. In some cases, symbolic model checking may fail to verify a small problem (say, with 10^7 states) because there is no compact BDD representation for the underlying transition relation. Therefore, SAL-SMC is usually used for verifying simplified and heavily abstracted models.

SAL-BMC. The use of Boolean satisfiability (SAT) solvers for verifying temporal logic properties has been explored through a technique known as *bounded model checking* (BMC) (Clarke et al. 2001). As with symbolic model checking, the state is encoded in terms of booleans. The program is unrolled a bounded number of steps for some bound k, and an LTL property is checked for counterexamples over computations of length k. Thus, a BMC problem corresponds to encoding all bounded simulation problems as a Boolean satisfiability problem. For example, to check whether a program with initial state I and next-state relation T violates the invariant φ in the first k steps, one checks, using a propositional SAT solver:

$$\begin{aligned} &I(s_0) \wedge \\ &T(s_0, s_1) \wedge T(s_1, s_2) \wedge \ \ldots \ \wedge T(s_{k-1}, s_k) \wedge \\ &(\neg\varphi(s_0) \vee \ \ldots \ \vee \neg\varphi(s_k)) \quad . \end{aligned}$$

This formula is satisfiable if and only if there exists a path of length at most k from the initial state s_0, which violates the invariant φ. This BMC methodology has been extended to BMC for infinite-state systems by translating the problem to a propositional constraint satisfication problem (de Moura et al. 2002; de Moura and Rueß 2002). The constraints involved might be linear arithmetic constraints, equalities over uninterpreted function symbols, array and bitvector constraints, or any combination thereof. Given a SAL specification with data types such as integers and arrays, an LTL formula with constraints on these datatypes, and an upper bound k, SAL-BMC decides whether there is a counterexample of length up to k to the hypotheses that a (possibly infinite) transition system satisfies its temporal specification. The corresponding constraint satisfaction problems are solved using the ICS decision procedures (Filliâtre et al. 2001). In this way, SAL-BMC is applicable for infinite-state verification problems, and it has been applied for continuous-time systems and special cases of hybrid systems (Sorea 2002).

It has been demonstrated that BMC can be more effective in falsifying hypotheses than traditional model checking (Clarke et al. 2001; Copty et al. 2001). Bounded model checking is therefore often used for refutation, where one systematically searches for counterexamples whose length is bounded by some integer k. The bound k is increased until a bug is found, or some precomputed *completeness threshold* or *diameter* (namely, the longest of all the shortest path to any reachable state) is reached. Unfortunately, the computation of completeness thresholds is usually prohibitively expensive and these thresholds may be too large to effectively explore the associated bounded search space. In addition, such completeness thresholds do not even exist for many infinite-state systems.

SAL-IND. The SAL induction tool combines refutation based on bounded model checking techniques with

verification based on a generalized induction rule, called k-induction (de Moura et al. 2003). This rule first requires demonstrating the invariance of a safety property in the first k states of any execution. Consequently, error traces of length k are detected. This induction rule also generalizes the usual invariance rule in that it requires showing that if the property under consideration holds in every state of every execution of length k, then every successor state also satisfies φ. As in BMC, the bound k is increased until either a violation is detected in the first k states of an execution or the property at hand is shown to be k-inductive. In the ideal case of attempting to prove correctness of an inductive property (that is, a property preserved on all transitions), 1-induction suffices and iteration up to a, possibly large, complete threshold, as in BMC, is avoided. Although k-induction is complete for finite systems, in practice, it usually works only for small values of $k < 20$.

Whenever k-induction fails to prove a property, there is a counterexample of length $k+1$ such that the first k states satisfy φ and the last state does not satisfy φ. If the first state of this trace is reachable, then φ is refuted. Otherwise, the counterexample is labeled *spurious*. By assuming the first state of this trace to be unreachable, a spurious counterexample is used automatically to obtain a strengthened invariant. Many infinite-state systems can only be proven with k-induction enriched with invariant strengthening, whereas for finite systems and many continuous-time systems the use of strengthening is an optimization in that it decreases the minimal k for which a k-induction proof succeeds (de Moura et al. 2003).

3 MODELING THE PRIORITY-CEILING PROTOCOL

We report on our work and experience in modeling and validating Dutertre's version (Dutertre 2000) of the priority-ceiling protocol (PCP) using SAL. The PCP protocol is particularly interesting, since scheduling is a critical component of real-time system that are being used in safety-critical applications such as Integrated Modular Avionics (IMA), and many real-world schedulers such as Honeywell's DEOS are based on simpler, but supposedly better understood, versions of PCP such as the highest locker protocol. In such a context, one must obtain strong guarantees of correctness, and rigorous development and verification methods are required.

Real-time scheduling involves the allocation of resources and time intervals to tasks in such a way that certain timeliness performance requirements are met. A scheduling problem is given in terms of a set of periodic tasks with given *period* length, *priority*, and *budget*, and a corresponding real-time scheduler needs to ensure that every task consumes its *budget* of processing time on a shared processor in each of its periods. Access to other shared resources such as common I/O channels is controlled by *semaphores* for ensuring mutual exclusive access to each of these resources. When synchronization primitives, such as semaphores, are used, there is a problem called *priority inversion* which causes low priority jobs to prevent higher priority jobs from running. For instance, a job j can be blocked when trying to lock a semaphore S if a job k of lower priority has locked S before j was dispatched. As a result, a job j of top priority can be unable to execute and a job k of lower priority than j can become active. This phenomenon may block j for long periods of time, since other jobs, with priority greater than k, may prevent k to execute and consequently to unlock S. So, the high-priority job j can then be delayed by the low-priority job k that locks S but also by any job of intermediate priority that might preempt k. Since high-priority jobs are usually the most urgent and may have tight deadlines, such unrestricted priority inversion can be disastrous. In the *Priority Ceiling Protocol*, the following approach is used: each semaphore S is assigned a fixed *ceiling* which is equal to the highest priority among the jobs that need access to S, and a job j executing $lock(S)$ is granted access to S if the priority of j is strictly higher that the ceiling of any semaphore locked by a job other than j. Otherwise, j becomes blocked and S is not allocated to j.

The scheduler and each of the jobs are represented as SAL modules. Each active job nondeterministically chooses to either lock or unlock a semaphore or to perform some local `step` computation (Figure 2). Thus, the actions of a job can be modeled using the abstract data type `Command` below.

2

```
Command: TYPE = DATATYPE
    cmd_lock(arg: Semaphore),
    cmd_unlock(arg: Semaphore),
    cmd_unlock_all,
    cmd_step
 END
```

Given the identifier of the currently active process, the current configuration `RSRC` of the semaphores, and the clock value, the PCP scheduler picks an executable job of highest precedence which is not blocked, and controls job selection through the `turn` variable. The skeleton of the SAL module for specifying this scheduler can be found in Figure 3. This module has local variables for a discrete clock (with a sufficiently large upper bound depending on the job configuration), the current dispatch times, and the current job states. At each clock tick, it receives a command from the currently active job and updates the resources `rsrc` depending on this

```
scheduler: MODULE =                                          3
BEGIN
  LOCAL clock : ClockRange
  LOCAL dispatch : ARRAY JobIdx OF ClockRange
  LOCAL  job_state : JobState
  OUTPUT turn : Turn
  LOCAL  rsrc : RSRC
  INPUT  cmd : Command
  INITIALIZATION
    clock = 0;
    dispatch = [[j : JobIdx] 0];
    job_state = [[j : JobIdx] 0];
    rsrc = rsrcCtx!initial_rsrc;
    turn = idle_turn
  TRANSITION
    clock' = adjust(clock + 1);
    job_state' =
      [[j : JobIdx]
         IF sleeping?(j, job_state) AND
            dispatch[j] = clock
         THEN 1
         ELSIF turn?(turn, j) THEN
           IF end_of_budget?(j, job_state) THEN 0
           ELSE job_state[j] + 1 ENDIF
         ELSE job_state[j] ENDIF ];
    dispatch' = ...
    rsrc' = ...

    [
     ([](j : JobIdx ):
        eligible?(j, rsrc, job_state')
          --> turn' = job_turn(j))
    []
     ELSE --> turn' = idle_turn
    ]
END
```

Figure 3: PCP Scheduler in SAL

command. Furthermore, the clock is incremented, and the state of each job is updated. Now, an eligible job `j` is selected to be active, depending on the old value of the resources and the new (!) state of the jobs. The `[j: JobIdx]` construct in this specification denotes simultaneous array updates, and `[]` denotes asynchronous composition.

The use of parametric transition systems in SAL allow us to investigate different task sets by simply instantiating the scheduler model without changing specifications. In particular, the PCP model is parameterized with respect to the number of tasks, the number of semaphores, and the specifics for each task. In this way, the PCP model can be reused for different scheduling problems by means of simple instantiation of parameters.

Although time is progressing indefinitely, the resulting system, for a given configuration, is essentially finite-state. Indeed, for the assumed periodicity of processes it suffices to consider time only up to the least common multiple of the task periods. Thus, we can restrict ourselves to the SAL validation tools for finite-state systems.

4 VALIDATING TIME PARTITIONING

Time-partitioning is a crucial property for hard real-time schedulers, particularly those in which application of different criticalities run on the same processor. In a time-partitioned operating system, the scheduler is responsible for ensuring that the actions of one job can not affect other jobs guaranteed access to CPU execution time. We say that a deadline has been missed for job `j` if the clock is at a period boundary for job `j` but the job `j` has not been put into sleeping mode. The corresponding theorem `deadline_missed`, expressed in LTL, formalizes that this condition is never been violated.

```
dl_missed?(                                                  4
    dispatch : ARRAY JobIdx OF ClockRange,
    job_state : JobState,
    clock : ClockRange): BOOLEAN
=
  (EXISTS (j : JobIdx) :
     dispatch[j] = clock AND
     NOT sleeping?(j, job_state));

deadline_missed : THEOREM
 system |-
  G(NOT(dl_missed?(dispatch, job_state, clock)));
```

Similarly important, at each clock tick, there should be at least one job ready to execute.

```
deadlock?(job_state:JobState,t:Turn):BOOLEAN = 5
  idle_turn?(t) AND
  (EXISTS (j: JobIdx) :
     ready_to_execute?(j, job_state));

deadlock_free : THEOREM
  system |- G(NOT deadlock?(job_state, turn));
```

We prove these properties for the three scheduling configurations in Figure 4 using various SAL verification tools.

Configuration 1 has three jobs with the given priorities, periods, budgets, and semaphores as given in Figure 4. This configuration leads to a scheduling problem with $209,737,024$ reachable states. This number is clearly beyond the capabilities of explicit-state model checking, but the deadlock property is easily

Configuration 1.

job	priority	period	budget	semaphores
1	100	8	3	$\{1,3\}$
2	50	12	4	$\{1,2\}$
3	25	20	5	$\{1,2,3\}$

Configuration 2.

job	priority	period	budget	semaphores
1	100	28	4	$\{1,3\}$
2	50	16	4	$\{1,2\}$
3	25	16	4	$\{1,2,3\}$

Configuration 3.

job	priority	period	budget	semaphores
1	100	10	3	$\{1\}$
2	75	16	4	$\{2\}$
3	50	8	7	$\{1,3\}$
4	50	12	6	$\{1,2\}$
5	25	20	5	$\{1,2,3\}$

Figure 4: Configurations

proved with symbolic model checking (both forward and backward reachability) and induction of depth $k = 1$.

SAL-SMC	(forward) 76.21 secs
	(backward) 4.24 secs
SAL-IND	($k = 1$) 6.4 secs

For this property, a proof using SAL-BMC without induction is much harder than with SAL-IND, since the diameter of the system to be explored is 194. For the inductiveness of the property under consideration, however, exploration of depth 1 suffices. The timeliness property does not hold for configuration 1, and counterexamples of length 16 are easily generated using forward symbolic model checking and bounded model checking.

SAL-SMC	(forward) 7.64 secs
	(backward) timeout
SAL-IND	7.7 secs

Configuration 2 only generates $4,992$ reachable states and the diameter is 112. Again, it is straightforward to establish deadlock-freeness using either model checking or induction.

SAL-SMC	(forward) $9,46$ secs
	(backward) 8.24 secs
SAL-IND	($k = 1$) 4.15 secs

Symbolic model checking using forward traversal proves the timeliness property. Both backward simulation and induction fail, but at least, bounded model checking establishes the property up to the diameter.

SAL-SMC	(forward) 9.68 secs
	(backward) timeout
SAL-BMC	(upto $k = 112$) 24.83 secs

In general, however, the diameter of a system is difficult to compute, and therefore it is unclear when to stop increasing the bound k in BMC. k-induction fails for this problem, since it has to be iterated up to the recurrence diameter (the length of the longest acyclic path), which usually is much larger than the diameter. In contrast, SAL-ESMC proves this property almost immediately.

Configuration 3 generates a rather huge problem space with $329,924,301,744$ reachable states and the diameter of the system is 437. Again, all symbolic methods establish deadlock-freeness, but this time symbolic forward traversal is less efficient than both the other methods.

SAL-SMC	(forward) 7055.6 secs
	(backward) 16.54 secs
SAL-IND	($k = 1$) 8.49 secs

Time-partitioning fails for this configuration, and both forward symbolic model checking and k induction produce a counterexample of length 8.

SAL-SMC	(forward) 17.32 secs
	(backward) timeout
SAL-IND	11.38 secs

Altogether, the best choice of verification technique usually depends on the characteristics of the problem at hand, and each verification technique has its particular weaknesses and strengths. However, they are complementary in that when one is weak the other is strong.

5 CONCLUSIONS

Highly efficient symbolic evaluation technology can be used to apply the whole spectrum of analysis of programs and specifications from testing and debugging to verification. We believe that the relevant technology consisting of decision procedures and constraint propagation has progressed to a point where it can be employed efficiently for symbolic evaluation. A major advantage of symbolic simulation is that it scales smoothly from explicit-state exploration to inductive verification.

Symbolic simulation in k-induction proofs, as developed in the SAL framework, for example, combines refutation and verification-based methods in a natural and useful way.

We have described a graded sequence of integrated formal analysis technologies in SAL, based on symbolic simulation, and demonstrated their effectiveness. In the early life cycle of a model, testing, debugging, and explicit-state exploration seem to be particularly effective for validation, whereas more heavy-weight verification tools are applied at later life cycles.

Compared to testing, symbolic simulation provides increased *coverage* and is applicable to partial models and high-level designs. On the other hand, symbolic simulation is often restricted too rather shallow exploration of state spaces compared to, say, random simulation. Combinations of explicit with symbolic-state exploration should make it possible to not only drastically increase coverage of explicit-state simulation but also to use localized symbolic simulation to drive simulations to territory in the state space that would otherwise remain unexplored. Much more intricate combinations seem to be possible. For example, the state space is divided into explicitly and symbolically represented set of sets, and simulation consists of a hybrid of explicit search and constraint solving. Symbolic simulation can also be used to generate "cheap" invariants, which themselves are used to restrict the search space for explicit exploration.

There is much more to a computational system than merely correctness, since it should also provide certain *quality of service*. In the priority-ceiling protocol, for example, an upper bound on the maximum time a process is blocked should be established. In some initial experiments, we developed SAL scripts based on explicit-state model checking for computing such maximum blocking time, but many more techniques from traditional simulation, in particular probabilistic methods, need to be incorporated.

ACKNOWLEDGMENTS. Many of the ideas and techniques presented here have been developed in close cooperation with our collegues S. Owre, J. Rushby, and N. Shankar.

REFERENCES

Abdulla, P. A., A. Annichini, S. Bensalem, A. Bouajjani, P. Habermehl, and Y. Lakhnech. 1999, July. Verification of infinite-state systems by combining abstraction and reachability analysis. See Halbwachs and Peled (1999), 146–159. 2

Alur, R., C. Courcoubetis, and D. Dill. 1993, May. Model-checking in dense real-time. *Information and Computation* 104 (1): 2–34. 2

Alur, R., C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. 1995, 6 February. The algorithmic analysis of hybrid systems. *TCS* 138 (1): 3–34. 2

Alur, R., and T. A. Henzinger. 1996, 27–30 July. Reactive modules. See IEEE Computer Society Press (1996), 207–218. 3

Bensalem, S., V. Ganesh, Y. Lakhnech, C. Muñoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari. 2000, June. An overview of SAL. In *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, ed. C. M. Holloway, 187–196. Hampton, VA: NASA Langley Research Center. Proceedings available at <http://shemesh.larc.nasa.gov/fm/Lfm2000/Proc/>. 2

Boyer, R. S., and J. S. Moore. 1979. *A computational logic*. New York, NY: Academic Press. 1

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

Chandy, K., and J. Misra. 1988. *Parallel program design: A foundation*. Addison Wesley. 3

Clarke, E. M., A. Biere, R. Raimi, and Y. Zhu. 2001. Bounded model checking using satisfiability solving. *Formal Methods in System Design* 19 (1): 7–34. 2, 4

Clarke, E. M., O. Grumberg, and D. E. Long. 1994, September. Model checking and abstraction. 16 (5): 1512–1542. 2

Copty, F., L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Vardi. 2001, July. Benefits of bounded model checking in an industrial setting. In *Computer-Aided Verification, CAV 2001*, Volume 2101 of *LNCS*, 436–453: Springer-Verlag. 2, 4

de Moura, L., and H. Rueß. 2002, May. Lemmas on demand for satisfiability solvers. Presented at SAT 2002, accepted for journal publication. Available at <http://www.csl.sri.com/users/demoura/sat02_journal.pdf>. 4

de Moura, L., H. Rueß, and M. Sorea. 2002, July. Lazy theorem proving for bounded model checking over infinite domains. In *International Conference on Automated Deduction (CADE'02)*, ed. A. Voronkov, Volume 2392 of *LNCS*, 438–455. Copenhagen, Denmark: Springer-Verlag. 2, 4

de Moura, L., H. Rueß, and M. Sorea. 2003, July. Bounded model checking and induction: From refutation to verification. In *Computer-Aided Verification, CAV '2003*, ed. W. A. Hunt, Jr. and F. Somenzi, Volume 2725 of *LNCS*, 14–26. Boulder, CO: Springer-Verlag. 5

Dill, D., T. Henzinger, S. Owre, and N. Shankar. 2001, March. The SAL language. Technical Report SRI-CSL-01-02, Computer Science Laboratory, SRI International, Menlo Park, CA. 2

Dutertre, B. 2000, December. Formal analysis of the priority ceiling protocol. In *Real Time Systems Symposium*. Orlando, FL: IEEE Computer Society. To appear. 5

Filliâtre, J.-C., S. Owre, H. Rueß, and N. Shankar. 2001, July. ICS: Integrated Canonization and Solving. In *Computer-Aided Verification, CAV '2001*, ed. G. Berry, H. Comon, and A. Finkel, Volume 2102 of *LNCS*, 246–249. Paris, France: Springer-Verlag. 4

Godefroid, P., and D. E. Long. 1996, 27–30 July. Symbolic protocol verification with queue BDDs. See IEEE Computer Society Press (1996), 198–206. 2

Halbwachs, N., and D. Peled. (Eds.) 1999, July. *Computer-aided verification, cav '99*, Volume 1633 of *LNCS*, Trento, Italy. Springer-Verlag. 8, 9

Hardin, D., M. Wilding, and D. Greve. 1998, June. Transforming the theorem prover into a digital design tool: From concept car to off-road vehicle. In *Computer-Aided Verification, CAV '98*, ed. A. J. Hu and M. Y. Vardi, Volume 1427 of *LNCS*, 39–44. Vancouver, Canada: Springer-Verlag. 1

Holzmann, G. 1998, March. Designing executable abstractions. In *Second Workshop on Formal Methods in Software Practice (FMSP '98)*, ed. M. Ardis, 103–109. Clearwater Beach, FL: Association for Computing Machinery. 4

IEEE Computer Society Press 1996, 27–30 July. *Proceedings, 11th annual ieee symposium on logic in computer science*, New Brunswick, New Jersey. IEEE Computer Society Press. 8, 9

Kautz, H. A., and B. Selman. 1992. Planning as satisfiability. In *European Conference on Artificial Intelligence*, 359–363. 2

Loiseaux, C., S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. 1995. Property preserving abstractions for the verification of concurrent systems. *FMSD* 6:11–44. 2

Manna, Z., and A. Pnueli. 1995. *Temporal verification of reactive systems: Safety*. Springer-Verlag. 2

McMillan, K. 1993a. *Symbolic model checking*. Kluwer Academic Publishers, Boston. 3

McMillan, K. L. 1993b. *Symbolic model checking*. Boston, MA: Kluwer Academic Publishers. 2

Moore, J. S. 1998, November. Symbolic simulation: An ACL2 approach. In *Formal Methods in Computer-Aided Design (FMCAD '98)*, ed. G. Gopalakrishnan and P. Windley, Volume 1522 of *LNCS*. Palo Alto, CA: Springer-Verlag. 1

Saïdi, H., and S. Graf. 1997, June. Construction of abstract state graphs with PVS. In *Computer-Aided Verification, CAV '97*, ed. O. Grumberg, Volume 1254 of *LNCS*, 72–83. Haifa, Israel: Springer-Verlag. 2

Saïdi, H., and N. Shankar. 1999, July. Abstract and model check while you prove. See Halbwachs and Peled (1999), 443–454. 2

Shankar, N. 2000, March. Symbolic analysis of transition systems. In *Abstract State Machines: Theory and Applications (ASM 2000)*, ed. Y. Gurevich, P. W. Kutter, M. Odersky, and L. Thiele, Volume 1912 of *LNCS*, 287–302. Monte Verità, Switzerland: Springer-Verlag. 2

Sorea, M. 2002. Bounded model checking for timed automata. *ENTCS* 68 (5). At: <http://www.elsevier.com/locate/entcs/volume68.html>. 4

AUTHOR BIOGRAPHIES

HARALD RUESS is a Computer Scientist at the Computer Science Laboratory of SRI International. His current work is concerned with the development and implementation of decision procedures, the application of formal methods for analyzing software and hardware systems, analysis of security protocols, and the logical foundation of evidential transactions. The address of his home page is <www.csl.sri.com/users/ruess>.

LEONARDO DE MOURA is a Computer Scientist at the Computer Science Laboratory of SRI International. He is mainly concerned with developing and implementing model checkers, simulators, and other verification tools. His e-mail address is <demoura@csl.sri.com>.