Checking Formal Specifications under Simulation

William Canfield*
CAE+
9130 Jollyville Road #340
Austin, TX 78759
canfield@cae-plus.com

E. Allen Emerson
Dept. of Computer Sciences
University of Texas at Austin
Austin, TX 78712
emerson@cs.utexas.edu

Avijit Saha
International Business Machines
11400 Burnet Road
Austin, TX 78758
saha@austin.ibm.com

Abstract

"Verification" of large multiprocessor designs currently relies heavily on simulation. Formal techniques such as model checking are typically only applied to small parts of the system, due to issues of computational and notational complexity. With these two facts in mind, we have designed a platform which aims to help bridge the gap between formal verification and simulation. We present a temporal logic specification language which includes constructs for specifying system behavior at a high level of abstraction, and discuss its use in simulation and model checking.

1 Motivation

As the complexity of computer systems increases, design verification efforts run into a mathematical Catch 22: the industry standby, *simulation*, can no longer exercise the design enough to provide confidence in its correctness; automated formal approaches such as *model checking* [2] are derailed by the state explosion problem. Since neither simulation nor model checking can currently solve the entire verification problem, industrial verification efforts often employ both methods: formally verifying small pieces of a system, and running billions of cycles of simulation to test the entire system.

We are implementing a platform which reduces the amount of effort duplicated by these two approaches to system verification. By using temporal logic formulas as specifications in a cycle simulation environment, we give simulation a more formal basis, and the same formulas can then be used to model-check the design or parts of it. The temporal logic we have chosen is able to express the kinds of behaviors that are normally checked during simulation: this will also aid the formal verification effort by allowing a designer to compose very abstract specifications, then mechanically refine them for model checking. Such a methodology reduces the risk of typographical errors and errors of omission.

It may seem paradoxical to apply temporal logic specifications—which can specify infinite behaviors of a system—to finite simulation traces. For instance, a finite trace cannot in general verify the formula $\mathbf{G}P$ ("always P"). However, a trace that reaches a state where P is false, shows that the system does not satisfy $\mathbf{G}P$. Another way to view the problem: given a finite run R and a specification Φ, decide if it is possible that R is a *prefix* of an infinite run which satisfies Φ. We have implemented a program to test such specifications in a simulation environment.

The remainder of this paper is organized as follows. In Section 2 we describe the specification language we use, including its syntax and semantics. Section 3 presents the trace checking algorithm, and proves its correctness; complexity issues are taken up in Section 4. Finally, Section 5 summarizes the contributions of the work, and outlines the direction of our future research.

2 Specification language

Our simulation trace checking program, "SHERLOCK", is implemented on top of a cycle simulator, which simulates the operation of the system under test in response to a sequence of inputs. SHERLOCK polls the simulator for the signal or register values it needs at each simulated cycle, and checks it against a specification for the system. This section describes SHERLOCK's specification language.

2.1 Syntax and semantics

In the taxonomy given in [1], the logic we use is a linear, integer-time, point-based temporal logic with bounded and unbounded future-time operators. It is augmented with bounded operators and such abstractions as auxiliary variables, data arrays, and first-in first-out (FIFO) buffers. While it is not truly a first-order logic, it permits operations on formula variables which makes it more powerful than a propositional

*Employed by IBM when this research was done.

logic. In effect, SHERLOCK permits two kinds of specification: high-level, abstract specifications, which model the input-output behavior of a system; and low-level specifications, which verify that the internal operation of the system works as expected. Our choice of variable-changing operations has largely been motivated by elements of designs we have been testing; for examples of other temporal logics which allow such operations, see [6, 5].

We consider a specification to be an unordered set of formulas, which we often refer to as *rules*. It is possible to write a specification as a single large formula, but we expect a modular set of rules to be easier to compose and maintain.

All values in the logic are represented internally as unsigned binary integers (even if they are written in decimal notation); in what follows, any reference to *size* or *number of bits* of an object refers to its binary representation.

2.1.1 Syntax

Data objects are the building blocks of the logic, but they are not themselves formulas, since they have no truth value.

- **symbols** represent elements of the design, such as signal wires and data registers.
- **constants** are non-negative integers of at most 32 bits.
- **variables** provide storage which is accessible from any rule of the specification.

Statements are the smallest formulas of the logic:

- **comparisons** of similarly-sized data objects (e.g., "$PROC3\_OUT = deq(OUTQ\_3)$").
- **assignments** to variables (e.g., "$ARRAY[COUNT] \leftarrow PROC3\_IN$").
- **enqueue** statements (e.g., $enq(INQ\_3,\ PROC3\_IN)$).
- **increment** statements allow ordered access to variable arrays. An increment statement may have an optional "period" (e.g., if the value of $COUNT$ is 2, $inc(COUNT, 3)$ sets it to 0).

Boolean formulas If Φ_1 and Φ_2 are formulas, $\Phi_1 \vee \Phi_2$ and $\neg\Phi_1$ are also formulas.

Temporal formulas If Φ_1 and Φ_2 are formulas, and n is a positive integer, $\Phi_1 \mathbf{U} \Phi_2$ (read "Φ_1 *until* Φ_2"), $\Phi_1 \mathbf{U}^{\leq n} \Phi_2$, and $\Phi_1 \mathbf{U}^{=n} \Phi_2$ are also formulas. The integer n is a *time bound*; we say that $\Phi_1 \mathbf{U} \Phi_2$ is an *unbounded* formula, and the other two are *bounded*.

We use the usual abbreviations for boolean and temporal formulas: $\Phi_1 \wedge \Phi_2$ means $\neg(\neg\Phi_1 \vee \neg\Phi_2)$; $\Phi_1 \Rightarrow \Phi_2$ means $\neg\Phi_1 \vee \Phi_2$; $\mathbf{F}\Phi_1$ (read "*eventually* Φ_1") means $true\mathbf{U}\Phi_1$; and $\mathbf{G}\Phi_1$ (read "*always* Φ_1") means $\neg\mathbf{F}\neg\Phi_1$. Bounded versions of $\mathbf{G}$ and $\mathbf{F}$ are constructed from the corresponding bounded until in the same way. We will use the symbol $\mathcal{S}$ to refer to the set of formulas conforming to the syntax given here.

2.1.2 Semantics

Definitions: for the purposes of this semantics, a *system* M is comprised of a set of symbols called *signals* (representing the values of wires or bundles of wires)[1]. Each signal has a single natural-number value at any time; the values of signals are observed only at discrete points of time. The interval between two consecutive observation points is called a *cycle*; cycles are numbered in monotonically increasing order, starting at zero. We often speak of the value of a signal *at* some cycle n: by this we mean the value of the signal at the observation point at the end of cycle n. The *state* of M at a cycle is the valuation of its signals at that cycle.

A *trace* or *run* of M is a finite sequence $t_0, \ldots, t_{n-1}$ such that t_i is the observed state of M at cycle i. A *specification* S of M is an unordered collection $\{\Phi_0, \ldots, \Phi_{m-1}\}$ of formulas conforming to the syntax given above, such that each symbol (in the sense of Section 2.1.1) of each Φ_i is a signal belonging to M. We say that a trace T *violates* a specification S if, for some $\Phi \in S$, Φ is $false$ at cycle 0 (the first cycle), for the sequence of states given by T. The truth of a formula is defined by induction over the formula structure, as follows:

Data objects The value of a symbol can change only at the end of a cycle. A variable can be changed during a cycle, but there is also a means to declare a variable to have *read-before-write* (respectively, *write-before-read*) semantics, meaning that if a rule updates the variable, any rule which reads the variable at the same cycle obtains the old (respectively, new) value.

Statements All statements have a value of either $true$ or $false$ at the time they are evaluated. A

[1] We expect M to be a state machine with a well-defined transition relation, but since the logic deals only with the signal values, it need not be so.

comparison $a\ R\ b$ is *true* iff the a and b stand in relation R to one another. Assignments and other state-changing operations always evaluate to *true*.

Boolean formulas These have the usual semantics, but are evaluated "lazily". For instance, if Φ is *false*, a formula like $\Phi \Rightarrow$ "$X \leftarrow 0$" evaluates to *true* without the variable X being updated.

Temporal formulas The formula $\Phi_1\ \mathbf{U}\ \Phi_2$is *true* at cycle i iff $\Phi_2$is *true* at some cycle j such that $j \geq i$, and $\Phi_1$is *true* at every cycle k, for $i \leq k < j$. The bounded formulas have similar meanings, but with greater restrictions on the value of j: for $\Phi_1\ \mathbf{U}^{\leq n}\ \Phi_2$, replace "$j \geq i$" by "$i \leq j \leq i+n$"; for $\Phi_1\ \mathbf{U}^{=n}\ \Phi_2$, replace "$j \geq i$" by "$j = i+n$".

Rule violation is defined in terms of the first cycle of the trace, which corresponds to the usual way of evaluating temporal logic formulas relative to a Kripke structure (i.e., evaluating them with respect to the start state). However, since SHERLOCK operates on finite traces, there are *three* possibilities for the truth value of a rule Φ at cycle 1: *true*, *false*, or *unknown*. As noted in Section 1, a rule such as **GP** cannot be shown to be *true* by a finite trace: if it is not shown to be *false* by the trace, then its value is unknown. The significance of unfulfilled rules must be interpreted by the user of the program.

2.2 Other language issues

We have included in the language parameterized rule *templates*, which greatly simplify the task of specifying a system with many symmetric components. For example, a specification for part of a memory controller we tested involved 94 rules, but due to the regularity of the design, it was implemented as 15 templates, invoked with suitable sets of parameters[2]. The use of templates greatly improves the human-readability of a specification, and lessens the chance of typographical error.

Although SHERLOCK allows state-changing operations to be mingled freely with assertions about the system, we have found that it is best to separate these two functions: writing some rules which serve only to update variables, and others which specify the expected behavior of the system.

[2] Incidentally, as another rough measure of the conciseness of the language, that specification was written as about 400 lines of text; a C program which checked the same behaviors was 1600 lines long.

3 Formula-checking algorithm

If there were no temporal operators in the logic, checking the truth of a formula at some cycle of a simulation would simply involve calculating a boolean expression relating the symbols of the formula; the result would be known immediately. With temporal operators, however, it is most often the case that the truth of a formula at some cycle depends on the state of the system at one or more instants in the future. The following identities are needed (let $\Phi_1, \Phi_2 \in \mathcal{S}$, and $n > 1$):

$$\begin{array}{rcl} \Phi_1\mathbf{U}\Phi_2 & \equiv & \Phi_2 \vee (\Phi_1 \wedge \mathbf{X}(\Phi_1\mathbf{U}\Phi_2)) \\ \Phi_1\mathbf{U}^{\leq n}\Phi_2 & \equiv & \Phi_2 \vee (\Phi_1 \wedge \mathbf{X}(\Phi_1\mathbf{U}^{\leq n-1}\Phi_2)) \\ \Phi_1\mathbf{U}^{=n}\Phi_2 & \equiv & \Phi_1 \wedge \mathbf{X}(\Phi_1\mathbf{U}^{=n-1}\Phi_2) \\ \Phi_1\mathbf{U}^{\leq 0}\Phi_2 & \equiv & \Phi_2 \\ \Phi_1\mathbf{U}^{=0}\Phi_2 & \equiv & \Phi_2 \end{array} \tag{1}$$

Here $\mathbf{X}$ is the *next-time* operator; if $\Phi \in \mathcal{S}$, $\mathbf{X}\Phi$ means "the evaluation of Φ at the next cycle"; though it is not part of the syntax of the logic, it is equivalent to $\mathbf{F}^{=1}\Phi$. We use $\mathbf{X}$ as a meta-symbol, in order to highlight the distinction between return values of subroutines of the algorithm and subformulas of the rule being checked. An important characteristic of $\mathbf{X}$ is that it distributes freely over $\wedge$ and $\vee$.

3.1 Outline of the algorithm

Our algorithm runs in a cycle simulation environment. At each cycle, the simulator calls a SHERLOCK routine which requests the values of all signals belonging to active rules and marks those rules which need to be checked. A rule previously found to be *true* or *false* needn't be checked; another optimization is to pass over a rule if none of its symbols or variables have changed value since the last cycle (unless it has an unfulfilled bounded subexpression). Finally, *check_rules* (Figure 1) is called.

The function call to *eval*() in *check_rules* actually represents a class of functions, one for each operator or statement type in the syntax; the type of the root node of *rule*[*i*].*to_check* determines which one is called. Each operator evaluation routine performs an *eval*() on one or both of its arguments, according to the semantics given in Section 2.1.2. This recursive evaluation terminates at the leaf nodes of the rule, which have a value of *true* or *false* at every cycle. Evaluation of a non-leaf node Ψ may yield a value of *true* or *false*, but it may also yield a formula Ψ', such that $\mathbf{X}\Psi' \equiv \Psi$. If evaluation of the root node of a formula yields such an expression, then the newly constructed formula is checked in the place of the original at the next cycle.

The subexpression evaluation step is illustrated by Figure 1; the other $eval()$ routines are analogous. Note that the expression returned in the last line of *evaluate_until* may be simplified: if R is $false$, the *or* is unnecessary; if L is $true$, the *and* can be dropped. Subroutines for evaluating other types of subformulas work in a similar manner. Bounded temporal operators must also check that their time bounds are adhered to.

3.2 Correctness

Assuming that the class of $eval()$ routines faithfully implements the semantics of the logic, it is not difficult to believe that *check_rules* operates correctly, but the following point must be addressed: given a formula Φ which, at some cycle, does not evaluate to $true$ or $false$ but rather to Φ', we must be sure that $\mathbf{X}\Phi' \equiv \Phi$. First we state the following identities:

$$\begin{array}{rcl} \mathbf{X}\Phi_1 \vee \mathbf{X}\Phi_2 & \equiv & \mathbf{X}(\Phi_1 \vee \Phi_2) \\ \neg\mathbf{X}\Phi_1 & \equiv & \mathbf{X}\neg\Phi_1 \end{array} \tag{2}$$

Since $\wedge$ and $\Rightarrow$ are constructed from $\vee$ and $\neg$, $\mathbf{X}$ distributes over them in exactly the same way.

Theorem 1 Let $need\_to\_check(rule[i]) = true$ for all i, at every cycle. Then algorithm *check_rules* is correct; that is, given a formula $\Phi \in \mathcal{S}$ and a trace T, if Φ is either $true$ or $false$ of T (using the semantics given earlier), the algorithm will report that fact.

Proof Let Φ be a formula to be evaluated at cycle n. Our argument proceeds by induction on the structure of Φ. When dealing with temporal operators, we will restrict our attention to the unbounded *until*: proof of correctness of any other temporal operator is exactly similar.

If Φ is what we call a statement (a comparison or variable-changing operation), it evaluates immediately to either $true$ or $false$; if the evaluation routines adhere to the semantics of the logic, then *check_rules* gives the correct answer.

Now let $\Phi = \Phi_1\mathbf{U}\Phi_2$, where Φ_1 and Φ_2 are evaluated correctly by the algorithm. The result of applying the algorithm to Φ_i could be $true$ or $false$, or a formula Φ_i' whose truth next time is equivalent to the truth of $\Phi_i$this time. The operation of algorithm *evaluate_until* on Φ is summarized by the following table:

$eval(\Phi_1)$	$eval(\Phi_2)$	$eval(\Phi)$
true	true	true
true	false	Φ
true	Φ_2'	$\Phi_2' \vee \Phi$
false	true	true
false	false	false
false	Φ_2'	Φ_2'
Φ_1'	true	true
Φ_1'	false	$\Phi_1' \wedge \Phi$
Φ_1'	Φ_2'	$\Phi_2' \vee (\Phi_1' \wedge \Phi)$

Consider the most general case (represented by the last line in the table), where neither side of the formula evaluated to a truth value. The algorithm works correctly iff

$$\Phi \equiv \mathbf{X}[\Phi_2' \vee (\Phi_1' \wedge \Phi)]. \tag{3}$$

From the definition of *until* in (1), we have $\Phi \equiv \Phi_2 \vee (\Phi_1 \wedge \mathbf{X}\Phi)$, and our induction hypothesis is that $\Phi_1 \equiv \mathbf{X}\Phi_1'$ and $\Phi_2 \equiv \mathbf{X}\Phi_2'$. Substituting, we obtain: $\Phi \equiv \mathbf{X}\Phi_2' \vee (\mathbf{X}\Phi_1' \wedge \mathbf{X}\Phi)$; applying the identities in (2) gives us the desired result (equation (3)).

Similar arguments can be made for formulas of the forms $\Phi = \Phi_1 \vee \Phi_2$, and $\Phi = \neg\Phi_1$, as well as for the bounded temporal formulas; we omit them for the sake of brevity.

3.3 Optimizations

It is possible to strengthen the *need_to_check* predicate, so that not every rule need be checked at every cycle. Consider a formula Φ that contains no bounded temporal operators. Suppose that, at cycle $n-1$, $eval(\Phi)$ returned a value of Φ'. If, at cycle n, the values of all of the symbols and variables of Φ remain the same as at cycle $n-1$, then, at cycle n, $eval(\Phi')$ yields a result of Φ' again. We omit the proof, which is a simple matter of structural induction on Φ.

In fact the proof relies on the unchanged value of each subexpression of Φ, so that SHERLOCK can cut even more corners by constructing a list of symbols which are "crucial" to the truth value of Φ, and letting *need_to_check* return $true$ iff one of the crucial symbols has changed. For example, if $P = 1$ at cycle $n-1$, the formula

$$\Phi = \mathbf{G}((\text{“}P=0\text{”} \wedge \text{“}Q=0\text{”}) \Rightarrow \text{“}R=1\text{”})$$

evaluates to $true \wedge \mathbf{X}\Phi$. That is, $eval(\Phi)$ returns a value of Φ, and it will continue to return that value until P changes value, no matter what happens to Q and R. Thus, P is the only symbol crucial to Φ (until the next time Φ is evaluated). The set of these crucial symbols for Φ at some cycle coincides with the set of symbols visited during lazy evaluation. Thus,

```
check_rules()
{
  var result : expr;

  for i := 1 to num_rules
  {
   if (need_to_check(rule[i]))
   {
     result := eval(rule[i].to_check);
     if (result ∈ {True, False})
     {
       print "rule i is result";
       rule[i].to_check := nil;
     }
     else
     {
       rule[i].to_check := result;
     }
   }
  }
}
```

```
evaluate_F(E : expr) : expr
{
  var sub, result : expr;

  sub := evaluate(E.sub);
  if (sub = True)
  {
    return True;
  }
  else
  {
    if (sub = false)
    {
      return E;
    }
    else
    {
      result := or(sub, E);
      return result;
    }
  }
}
```

```
evaluate_or(E : expr) : expr
{
  var left, right, result : expr;

  left := evaluate(E.left);
  if (left = True)
  {
    return True;
  }
  else
  {
    right := evaluate(E.right);
    if (left = false)
    {
      return right;
    }
    else if (right.left = left)
    {
      return right;
    }
    else
    {
      result := new_node(OR, sub, E);
      return result;
    }
  }
}
```

Figure 1: Main checking loop, and sample evaluation routines.

compiling the list is a constant-time operation, and does not affect the running time of the algorithm.

4 Computational complexity

Definition: Let $\Phi \in \mathcal{S}$. The *length* of Φ, denoted $|\Phi|$, is the sum of the number of its statements and the number of its operators.

4.1 Tableaux vs. trace-checking

Formulas of propositional temporal logics can be represented as *Büchi automata* (one class of finite automata on infinite objects); the usual algorithm for testing satisfiability of a temporal logic formula relies on this fact. The *tableau construction* [3, 4] yields an automaton whose nodes and arcs are labeled with sets of the formula's subexpressions; for each infinite trace permitted by the formula, there is a path through the automaton which satisfies its acceptance condition. Given these characterizations of the state space and the language of the automaton, it is not surprising that its size is exponential in the length of the formula, as is the time required to construct it.

The traces we check can be thought of as describing paths in the automaton of the formula; the next-time formulas constructed by our algorithm correspond to states of the tableau. Our algorithm is somewhat naïve, in the sense that for complicated nestings of temporal and boolean operators, a pathological trace can fool it into constructing unnecessarily redundant next-time formulas. The redundancy results in a worst-case time complexity that is exponential in the size of the formula, though there are classes of interesting formulas for which the complexity is only linear in the length of the formula.

We mention two possible solutions to the redundancy problem. One is to represent boolean expressions as ordered binary decision diagrams (OBDDs), applying the OBDD reduction steps at the end of each cycle. In Section 4.2, another solution is presented: a completely different algorithm which can efficiently handle formulas of any complexity, but which is less suitable in practice, since it requires that a history of symbol values be stored (and referenced) as the simulation proceeds. So far, the naïve algorithm works well enough that the extra overhead required by the other solutions does not seem warranted.

4.2 An alternative algorithm

The problem of checking a formula Φ over a trace T can also be posed in terms of model checking. Recall that a trace T is is a finite sequence $t_0, \ldots, t_{n-1}$ such that t_i is the observed state of M at cycle i. Let $S = \{(t_i, i) : 0 \leq i < n\}$, let $R = \{((t_{i-1}, i{-}1), (t_i, i)) : 0 < i < n\}$, and let L be a mapping from S to the set of atomic formulas of Φ which labels (t_i, i) with the set of atomic formulas true at t_i. Then $M =$

(S, R, L) is a Kripke structure, each of whose states has exactly one successor (except $(t_{n-1}, n-1)$, which has no successors).

In such a structure, the semantics of the CTL operators **EU**, **EG**, and **EF** coincide with the semantics of the corresponding linear-time operators. Thus, a method for checking whether T violates Φ is to convert each temporal operator in Φ to its CTL equivalent, and apply the classical CTL model-checking algorithm. Let us call this trace-checking procedure "*MC_trace*"; for a formula of arbitrary length, its complexity is $O(|\Phi| \times \log|\Phi| \times |T|)$ (though for formulas with fewer than a given number of subformulas, the complexity is $O(|\Phi| \times |T|)$).

5 Conclusions

For the foreseeable future, industrial hardware design is likely to continue to use both cycle simulation and formal verification in the design validation process. We have described a platform, "SHERLOCK", which allows both techniques to be applied to a single design specification. Toward that end, we have defined a bounded-operator temporal logic, augmented with data structures and formula variables, and we have designed and implemented an algorithm which checks specifications of the logic in a cycle simulation environment. For most useful specifications, the running time of the algorithm is linear in the number of cycles simulated and the formula length. Future work includes refining abstract formulas of the logic into sets of propositional formulas, and satisfiability testing of abstract formulas.

Acknowledgments

The graphical interface for SHERLOCK was conceived and implemented by Ahran Dunsmoor; as the project progressed, the interface was maintained and extended by Samuel Guyer. Discussions with Richard Trefler and Kedar Namjoshi contributed greatly to the analysis of the trace-checking algorithm.

References

[1] R. Alur and T.A. Henzinger. Logics and models of real-time: A survey. In *Real Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 74–106. Springer-Verlag, 1991.

[2] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs,* Yorktown Heights, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[3] E. Allen Emerson and A. Prasad Sistla. Deciding full branching time logic. *Information and Control*, 61(3):175–201, June 1984.

[4] Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli. A decision algorithm for full propositional temporal logic. In *Proc. 5th Inter. Conf. on Computer Aided Verification* Elounda, Greece, pages 97–109, 1993.

[5] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, March 1994.

[6] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*, volume 1. Springer-Verlag, Berlin/New York, 1992.