

# To Model Check Or Not To Model Check

Nina Saxena  
C.E.R.C.  
Univ. of Texas  
Austin, TX 78712

Jason Baumgartner  
IBM  
11400 Burnet Rd.  
Austin, TX 78758

Avijit Saha  
IBM  
11400 Burnet Rd.  
Austin, TX 78758

Jacob Abraham  
C.E.R.C.  
Univ. of Texas  
Austin, TX 78712

## Abstract

*In the past, hardware design validation has relied primarily on simulation. New techniques such as model checking have been introduced, but no objective study investigating the advantages such techniques provide over simulation has been made. Simulation is model checking over a trace elicited by executing a test vector; model checking can be viewed as exhaustive simulation. Each has its own set of advantages and limitations.*

*A platform, “Sherlock”, was available wherein one could use properties or specifications expressed as CTL-like formulae interchangeably for checking simulation runs or for model checking. In this paper we describe and present results from an experimental study undertaken on a real implementation to better understand the efficacies of the two methods. We also present improved methods for accommodating liveness, fairness (of arbitration) and existence conditions in simulation and outline some techniques for writing implementation-independent properties for model checking.*

## 1. Introduction

Until recently simulation has been the industry standard approach for design verification, but formal techniques such as model checking are gradually finding their place in the verification process. As outlined in [2], we see strong similarities between the two approaches. In both cases we need specification models. For simulation they are the *checkers*, *behaviorals* or the *reference models* used to compute intermediate or final results and *irritators* or *test generators* that drive simulation vectors into the test model; in the case of model checking they are the *environments* and *properties* [5]. Checkers in simulation correspond to properties in model checking and irritators correspond to environments. In both cases the specifications are statements regarding the expected behavior of the design,

and significant effort is spent in their development. The properties should be true for the design regardless of the verification technique, but to date different languages and methods are used for the two — often with significant duplication of effort. This lack of an integrated approach to verification was also observed as a problem in [7].

With this in mind we have developed a platform [2], *Sherlock*, that allows one to write specifications in CTL-like notation. These can be translated to properties and environments for model checking or used as checkers or irritators in simulation. For model checking it is often the case that properties and environments must be overspecified or modified to contain the size of the BDDs, and some cannot be verified at all due to state space explosions. Furthermore, many complex designs are hierarchically partitioned and the partitions developed by different designers. Poor specification often yields bugs in the composed design due to incorrect assumptions about the interface driven by other designers’ logic. It is therefore desirable to check all properties and environments, written initially for model checking, in simulation as well.

This paper describes how properties and environments for model checking are converted to checks in simulation. The work was undertaken primarily to identify requirements for *Sherlock*. Our effort has helped us identify specification techniques that aid both approaches. Model checking validates conformance of a design to a property exhaustively, limited only by the correctness of assumptions of the corresponding environment and the fact that certain properties cannot be checked due to size limitations. Simulation, on the other hand, is random but not size- or environment-limited. It attempts to emulate “exhaustiveness” by coverage measurements [3] and coverage driven test generation [6]. Coverage models, test generation and checkers use specifications of some form and, like model checking, almost all verification effort is spent in this activity. Once we have the same specifica-

tion, from a practical bug discovery point of view, does it really matter which verification algorithm is used — model-checking or simulation? One of our goals was to better understand the efficacies of model checking versus simulation, and *Sherlock* was a suitable platform for evaluating this in an unbiased manner.

The rest of the paper is organized as follows. Section 2 describes the platform for executing properties in simulation. Properties to be model checked are expressed in Sugar [1] or CTL and the corresponding environments in environment description language (EDL)[5]. Both must be converted to a form that is accepted by our platform for simulation. This has led to interesting considerations on how to express and check properties such as arbitration fairness, liveness and existence conditions in simulation, and is discussed in Section 3. In Section 4 we discuss some specific properties of the design, which were tested. Care was taken to make sure that the set was representative of a general class of interesting design properties. We describe our experimental setup and present results. Discussions and conclusion follow in Section 5.

## 2. Sherlock Platform

*Sherlock* is a platform developed for providing a common verification interface for supporting both simulation and model checking. Underlying *Sherlock* is a trace checking program 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 them against a specification for the system. The following subsections describe *Sherlogic*, which is the specification language developed for *Sherlock*.

### 2.1. Language Features

*Sherlogic* is a linear, discrete-time, point-based, temporal logic with bounded and unbounded future-time operators. It is augmented with bounded past operators and such abstractions as auxiliary variables, data arrays, and first-in-first-out buffers (FIFOs or queues). It permits operations on formula variables which make it more powerful than a propositional logic.

*Sherlogic* permits two kinds of specifications: high-level abstract specifications and low-level specifications. While the former model the input-output behavior of a system, the latter verify that the internal operation of the system works as expected. Specifications are given as unordered sets of formulae, referred to as

*properties*. All values in the logic are represented internally as unsigned binary integers, irrespective of the notation used to express them. Parameterized property *templates* are used in the language for ease of specifying a system with many symmetric components.

### 2.2. Syntax and Semantics

*Sherlogic* was built using four main types of syntactical constructs: (i) Data objects, (ii) Statements, (iii) Boolean Formulae and (iv) Temporal Formulae.

*Data objects* are the building blocks of the logic, but they are not formulae in themselves since they have no truth value. They consist of *symbols* which represent elements of the design such as signal wires and data registers, *constants* that are unsigned integers of 32 bits, and *variables* that provide storage accessible from any property of the specification.

A *statement* is the smallest formula of the logic. It may be a comparison, assignment, an enqueue-dequeue statement, or an increment-decrement statement.

*Boolean formulae* are created using the usual boolean operators. *Temporal formulae* are constructed using temporal operators. These can be bounded or unbounded. Bounded operators contain a time bound expressed in terms of number of cycles counted from a certain event.

Variables may be classified as *read-before-write* or *write-before-read*. In the former case, if a property updates a variable, any property which reads the variable at the same cycle obtains the old value. In the latter case, it obtains the new value. All statements have a value of either *true* or *false* at the time they are evaluated; assignments and state-changing operations always evaluate to *true*.

## 3. Support for Formal Properties in Simulation

*Sherlogic* formulae support all CTL constructs that are used in model checking in addition to deterministic automata. However, the semantics and interpretation of some properties need to be different in simulation and may require special handling. In this section we outline some salient characteristics of environments and CTL properties typical of model checking. We describe techniques for supporting these in simulation.

### 3.1. Non-determinism

Model checkers typically support declarations of nondeterministic variables. These variables may either

comprise the environment (if they fanout to a model input) or be passive “satellite” variables. Specification is allowed upon both types. A satellite variable is derived from a passive automaton that records events in the design under test. In a typical case, model checking can be viewed as an aggregate of four different components — a non-deterministic language generator which is the environment, a deterministic language generator also a part of the environment, a deterministic language generator corresponding to the design and language acceptors which are the properties being verified. In order to use the properties and environments from model checking in simulation one must eliminate the non-deterministic automata. In addition, all language acceptor variables must be resolved to signals in the design.

When used in simulation, a model checking environment must remain a passive element. Hence the full nondeterminism of the environment must be resolved so as not to contradict the simulation model. For example, Figure 1 shows the dependence of an ar-

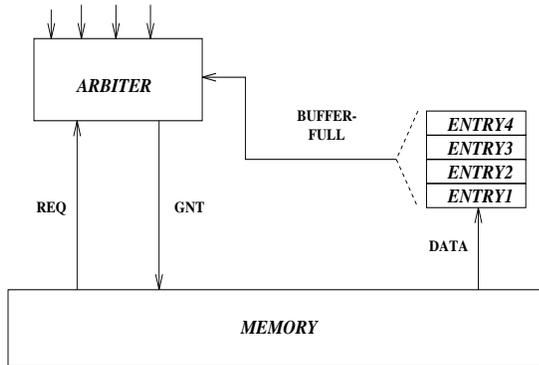


Figure 1. Arbiter Environment

biter on a data buffer. The buffer has four entries, and drives a buffer-full signal to an arbiter when three entries are reserved to cause the grant to memory to be pulled (to prevent buffer overflow). In model checking, we may wish to isolate the arbiter for verification (i.e., compositional verification [4]) in which case a non-deterministic environment variable would be needed to drive the buffer-full input to the model. In simulation, if we check a specification involving this environment variable, we must ensure that the values the variable takes do not contradict the current observable input

value of the simulation model (otherwise false negatives may occur). It is therefore required to build a wrapper around the simulation model. Such a wrapper contains constraints and definitions extracted from the environment and is used to prevent such contradictions.

### 3.2. Existence Conditions

Formulae with temporal operators are used to express and reason about events over time. In linear temporal logic such events are described in terms of a linearly ordered set of states. Examples of such operators are “ $G$ ” and “ $F$ ”. If  $p$  is a predicate, then “ $Gp$ ” means that “ $p$  is always true” and “ $Fp$ ” means “ $p$  is sometimes true”.

Model checking, however, is based on the notion of *branching time*. In this type of representation, the temporal order defines a tree that branches in future; each branch indicates a “possible” future. Hence, the notation for this requires extra operators such as “ $A$ ” and “ $E$ ”. If  $f$  is a formula, then “ $Af$ ” indicates that “ $f$  is necessarily true” (i.e., true along all paths) whereas “ $Ef$ ” indicates that “ $f$  is possibly true” (i.e., true along at least one path) [5].

Formulae that check for the possible existence of a predicate  $p$ , at any point in the future, are of the form “ $EFp$ ”. Verifying such formulae through simulation may seem futile since simulation, being a deterministic and finite run, does not allow the exhaustive search of predicate  $p$  in all possible states. This can be tackled by replacing properties containing existence conditions with equivalent properties expressed in terms of universal conditions, using the “ $A$ ” and “ $G$ ” operators. For example, if we want to verify that a certain data queue is empty at any point in the future, i.e., there exists a state in which all entries in the data queue are empty, we check for the reverse property in simulation. Essentially, it is verified that the data queue has at least one valid entry at all times in the simulation run. Whenever this is violated, the existence of an empty data queue state is verified. In effect, “ $EFp$ ” is a coverage measurement; the probability of occurrence of  $p$  increases marginally with the number of testcases run, as with any coverage metric.

This method can be extended to the recursive case where existence conditions are nested. In such cases, the entire formula consisting of existence conditions can be rewritten in terms of universal conditions; the reverse of the formula can then be verified through simulation.

### 3.3. Vacuous Passes

A property passes vacuously in model checking when the model composed with the formal environment obeys the specification trivially, primarily due to error in the logic, environment, or specification. For example, consider the property “ $AG(p \rightarrow AX(q))$ ”. If  $p$  never occurs, then the property is vacuously true: the subformula “ $AX(q)$ ” never affects the truth of the formula. Techniques for reporting such vacuous properties are discussed in [1].

### 3.4. Performance-Related Properties

Liveness properties specify conditions that some property must hold eventually (e.g., specified by the “ $F$ ” operator of CTL). For example, the specification: “ $AG(request \rightarrow AF(grant))$ ” means that along all paths, every request must be eventually followed by a grant. If a path (necessarily incorporating a finite cycle of states) exists such that a request is never followed by a grant, we have a violation of the liveness specification.

The most natural way of checking a “triggered” liveness condition in simulation is to track whether or not the triggering condition, i.e. the antecedent, occurred; if it did, then the “finally” property must subsequently hold during the simulation run. Otherwise an error is flagged upon simulation termination.

There are two distinct types of liveness bugs: deadlocks and livelocks. A deadlock situation occurs when the model under test enters a state which it cannot exit, such as a checkstop state. A livelock situation occurs when the model under test loops infinitely in a finite state transition cycle. Deadlock and livelock conditions constitute worst-case scenarios for performance problems in hardware.

Some hardware problems may have the appearance of a livelock condition. For instance, a design may break out of a livelock condition for a specific input sequence but continue in its state of pseudo-livelock otherwise. Deadlock and unbreakable livelocks need no special consideration when being checked in a simulation framework. Once entered, the state (or loop) can never be left, hence the “finally” property will still be unsatisfied at the end of the simulation run (whether the events to be simulated have all finished, or the testcase “timed-out” though incomplete) provided that the terminal condition is never true in the cycle. If it is, then model checking the specification will miss this loop as well.

Breakable loops may require additional consideration in simulation since an infinite sequence of inputs is

required to keep the model in its livelock loop, whereas simulation runs are finite. Consider the case that we have two masters arbitrating for a shared resource; the specification is that once a given master requests the resource, it will eventually be granted that resource. If asymmetry exists in the arbitration such that one master is always favored, this favored master can block the other master arbitrarily long through repeated requests. However, this favored master can only assert its request a finite number of times in a simulation run; once it stops requesting, the other master can access the resource unchallenged.

Often the actual liveness property to be checked is that the model under test is fair; i.e., it will supply grants equally to all masters; this will catch breakable and unbreakable loops. In this case, one may reword the specification to indicate that once a given master requests a resource, it will get that resource within “ $x$ ” resource grants given by the arbiter. For example, given our two-master scheme, if two resource grants occur after a given master asserts its request, and neither are to the given master, the given master was bypassed at least once. By increasing “ $x$ ”, we can check the unfairness of behavior to an arbitrarily large, yet finite, extent. Note that this arbitration fairness specification needs to be checked in conjunction with the original “ $AF$ ” property, since it will not flag an error if a loop is entered in which grant never occurs.

## 4. Experimental Setup and Results

All model checking as well as simulation experiments were done on RS6000 machines, with AIX 4.1.4 operating system. A common netlist was used for representing the model for both simulation and model checking. This netlist was obtained by synthesizing the HDL design.

The model under test was a four-port node controller which clustered four masters on one side to a system on the other. In the actual verification process for the chip, model checking was applied to the design over a period of about six months, whereas simulation proceeded for much longer. Since the logic continued to evolve for timing purposes after the model checking effort ended, six bugs known to have existed in previous versions of the model were seeded into the latest model. These bugs were then sought using both methodologies to ensure a one-to-one comparison of resources for model checking versus simulation.

Specifications and environments, first written in CTL for model checking, were then converted into their executable form written in *Sherlogic*, using *templates* and *properties* [2]. From the complete set of simulation

testcases used in the verification process, a small subset was chosen to represent a fair distribution of various categories, intended to stress the logic related to the selected bugs. The ratio of the testcases used for the purpose of this experiment, to the number originally used in simulation, is of the order 1:1000. The ones chosen for the experiment were not specifically targeted towards the seeded bugs. Simulation typically uses a large set of behavioral or specifications but we relied solely on the specifications that could also be validated via model checking. The next subsection describes the selected bugs and also discusses the criteria for their selection.

#### 4.1. Bug Descriptions

Verification tools vary in their performance depending on different factors. One of these factors is the type of logic being verified. Another factor is the bug itself. While model checking may work better in cases that do not cause a size problem, simulation may work better in cases where the number of possible next states, at each point in the automata, is limited. Therefore, it was essential to select bugs representing different types of logic and varying degrees of “hardness”, and to ensure the inclusion of bugs that were expected to stress both techniques.

The following bugs were selected for the experiment.

##### 1) Data buffer overflow.

This represents the data overflow problem as described in Section 3.1.

$$AG((Buffer\_Full \ \& \ New\_Transfer) \ \rightarrow \\ AX((A(!New\_Transfer \ U \ !Buffer\_Full))))$$

where *New\_Transfer* indicates that a new transfer has been initiated, which will fill one more slot in the buffer. *Buffer\_Full* indicates that only one more slot is available in the buffer, and two transfers under this condition would cause an overflow.

##### 2) Data grant before address grant.

This ensures that the master does not get a data grant before an address grant for a store request, which would result in disassociation of the data with the address.

$$AG(Store\_Request \ \rightarrow \\ A(!Data\_Grant \ U \ Address\_Grant))$$

##### 3) Low-priority port selection livelock.

This ensures that a given port will eventually get

access to the system for a low-priority transfer. The model biases towards high-priority transfers, even allowing the high-priority transfers to infinitely block low-priority transfers. In model checking, *Büchi* fairness “*AF(Grant\_Low\_Enabled)*” was necessary to restrict the reporting of failures due to these conditions. The property still failed due to a flawed round-robin scheme. As discussed in Section 3.5, traditional simulation would have missed this bug since it requires an infinite number of requests, from other ports, to fail as an “*AF*”. Hence, the problem was also cast as a safety property over automata in simulation; the automata counted the number of times a master was “skipped” sequentially by the arbitration logic. The pound sign in the formula below denotes the master number or id being checked by the formula.

$$AG(Port\_Request\_Low\_\# \ \rightarrow \\ AF(Port\_Grant\_Low\_\#))$$

##### 4) Data transfer livelock.

This is similar to **3** above; this bug is due to a flawed round-robin scheme.

##### 5) Data bus contention.

This verifies that a data-bus arbiter does not allow mutual access to a shared resource, in this case a shared bus.

$$AG(!(Master\_1\_Busy \ \& \ Master\_2\_Busy)).$$

##### 6) Unique prefetch addresses.

This validates that the prefetch logic does not fetch the same cache block twice due to any access patterns of the master. This bug would cause eventual deadlock if the master happened to fetch the replicated address.

$$AG(!(Buffer\_1\_Valid \ \& \ Buffer\_2\_Valid \ \& \\ (Buffer\_1\_Address \ = \ Buffer\_2\_Address)))$$

#### 4.2. Observations

Tables 1 to 3 list the results of our experiments. Table 1 shows results obtained on formally verifying the node controller design, for selected bugs, using model checking. Table 2 shows results obtained through simulation, for the same model and bugs. In both tables, the first column refers to the bug number, which corresponds to the order in which they are described in Section 4.1. The second and third columns give the CPU and user time, respectively, in verifying the model against the corresponding bug. The fourth column states the memory used and the last column states

whether the bug was “hit” by the verification technique (Y/N).

**Table 1. Model Checking Results**

BUG #	CPU TIME (sec)	USER TIME (sec)	MEM. USED (MB)	HIT
1	0.96	27	29	Y
2	1.69	281	116	Y
3	0.32	8	29	Y
4	1.67	16643	153	Y
5	0.4	11	28	Y
6	10.48	106131	911	N

**Table 2. Simulation Results**

BUG #	CPU TIME (sec)	USER TIME (sec)	MEM. USED (MB)	HIT
1	144899	265485	31	N
2	43684	79795	31	Y
3	420	840	31	Y
4	41607	75536	31	Y
5	0	0	0	Y
6	144899	265485	31	N

**Note: Bug 5 caught while testing for bug 3; hence the 0 time.**

Table 3 contains the results of checking the performance bug (bug 3) in both simulation and model checking frameworks (with the specification cast as a safety property over automata). The first column indicates the verification framework. The specification limits the number of times a given master’s request can be skipped in assigning grants. This number of skips is given in the second column. The third and fourth columns give CPU and user time required for the bug to first manifest (cumulative for all testcases run until bug manifestation in simulation); the fifth column is the memory used.

Due to the state space explosion induced by these automata, the specification was also cast as a temporal property with no automata. For a one-skip check,

**Table 3. Measuring “Performance”**

METHOD	# OF SKIPS (“x”)	CPU TIME (sec)	USER TIME (sec)	MEM. USED (MB)
M.CHK	1	0.56	18	27
M.CHK	2	0.61	22	28
M.CHK	5	0.59	30	29
M.CHK	10	0.51	40	30
M.CHK	20	0.91	56	32
M.CHK	50	1.37	153	39
SIM	1	420	840	31
SIM	2	0	0	0
SIM	5	5645	12498	31
SIM	10	16321	29349	31
SIM	20	0	0	0
SIM	50	0	0	0

**Note: Fields with 0 indicate bugs found by the testcase that caught the preceding one in the table.**

only 5.7 seconds (user time) were required; two skips required 50 seconds, three skips required 1108 seconds. This exponential time blowup is due to the more complex graph-traversal algorithms for unbounded temporal formulae. Note that the “AF” check was faster than even a two-skip check for both versions of the specification.

### 4.3. Discussions

The bugs selected for this experiment comprised a small subset of the total number of bugs known to exist in the node-controller design. Nine-tenths of the entire set of bugs in the design had been found through simulation and the remaining one-tenth through model-checking. The manpower employed on each effort was also roughly in the same ratio.

From this large set of bugs, our aim was to select some that were caught in model checking and not in simulation, and others that were caught in simulation and not model checking. However, for a fair sampling and unbiased comparison, we decided to choose bugs that exercised different parts of the logic.

In our experiment, the size of the design played an important role. The entire node controller was 48,853 latches. Consequently, the model checking effort required compositional verification [4]. The time re-

quired to develop adequate environments for each of the subunits ranged from one week for small blocks (e.g., a fifty-latch arbiter) to six weeks for large, complex blocks (e.g., a 2930-control-latch prefetch unit).

Due to the extremely large size of the model in bug 6, we were forced to greatly overspecify the formal environment. While several known bugs from the original set had been found in spite of this overspecified environment, there were slips that were caught in simulation.

The “data overflow” bug was caught by model checking but not by simulation. This could be a direct result of the fact that we ran only small number of random testcases and the fact that we did not attempt any coverage-driven test generation. From Tables 1 and 2, we can see that while the “low-priority port selection livelock” bug was easy to catch using either method, the “data transfer livelock” and “data grant before address grant” bugs took more time to show up, for both methods.

Bug 5, or the “data contention” bug was found in simulation in the same testcase as the one that found bug 3. It was seen that though random simulation took more CPU time than model checking, there were many testcases that uncovered multiple bugs. On the contrary, it is difficult to find a bug using model checking if the verification engineer is not specifically looking for it. This is due to the state explosion problem of model checking — all logic superfluous to a given specification is typically removed, hence a bug not in the target space is unlikely to be caught.

The CPU time as well as user time in simulation were found to be significantly higher than that of model checking. However, these do not reflect the time required by the verification engineer to generate environments for model checking or testcases for simulation; the former was a purely manual effort and required more time than the latter, making the overall time spent in model checking substantially higher than that in simulation on a per-bug basis. Furthermore, the amount of memory used for the full multi-chip simulation model was roughly equal to, and often drastically less than, the amount of memory needed to model check even a small portion of a single chip as indicated in Tables 1 and 2.

## 5. Conclusion

Model checking is deemed to be formal because it exhaustively proves the conformance of a design to a particular property. Although this is true for small designs, in reality this is far from being the case. Model checking *completely* verifies *parts* of a design, and sim-

ulation *partially* verifies a *complete* design. For real designs we feel that neither method can provide “proofs” and the sole objective is to detect bugs efficiently. The methods outlined in this paper for handling CTL formulae in simulation are not intended to constitute proofs of correctness but merely efficient schemes for bug discovery.

Our experiments validate the intuition that specification, and not so much the technique, is perhaps the most crucial component of a design validation process. Most of the bugs seeded in the design were caught in simulation. For model checking, specification is an enumeration of properties along with their corresponding environments. Deficiencies in model checking due to overspecification and erroneous environment assumptions can be partially compensated for by checking the unconstrained specifications in simulation. Generic specifications such as the ones introduced in this paper, for handling performance-related queries, are usually more abstract, less implementation specific and characterize the design space more “completely”. One disadvantage with such generic properties is that they tend to blow up in size. Therefore, it makes sense to formally check parts of the design using constrained versions of such generic properties and to check the unconstrained versions in simulation. Our attempt to develop a common set of specifications for both techniques have helped us develop better specification for the two approaches.

## References

- [1] I. Beer, S. Ben-David, C. Eisner, and A. Landver. Rule-base: an industry-oriented formal verification tool. *Design Automation Conference*, 1996.
- [2] W. Canfield, E. A. Emerson, and A. Saha. Checking formal specifications under simulation. *International Conference on Computer Design*, October 1997.
- [3] Y. V. Hoskote, D. Moundanos, and J. A. Abraham. Automatic extraction of the control flow machine and application to evaluating coverage of verification vectors. *International Conference on Computer Design*, 1995.
- [4] D. Long. *Model Checking, Abstraction, and Compositional Verification*. Ph.D. Thesis, CMU, July 1993.
- [5] K. L. McMillan. Symbolic model checking. *Kluwer Academic Publishers*, 1993.
- [6] B. O’Krafka, S. Mandyam, J. S. Kreulen, R. Raghavan, A. Saha, and N. Malik. Mptg: a portable test generator for cache coherent multiprocessors. *Proceedings of the 1995 IEEE Fourteenth Annual International Phoenix Conference on Computers and Communications*, 1995.
- [7] C.-J. Seger. An introduction to formal hardware verification. *University of British Columbia Tech. Report*, 1992.