SIMULATION-ORIENTED BEHAVIORAL VERIFICATION

by

Bassam Tabbara and Abdallah Tabbara

Memorandum No. UCB/ERL M99/38

13 July 1999

SIMULATION-ORIENTED BEHAVIORAL VERIFICATION

by

Bassam Tabbara and Abdallah Tabbara

Memorandum No. UCB/ERL M99/38

13 July 1999

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Simulation-Oriented Behavioral Verification

Bassam Tabbara*, Abdallah Tabbara
EECS Department, University of California at Berkeley
Berkeley, CA 94720
{tbassam, atabbara}@eecs.berkeley.edu

July 13, 1999

Abstract

Design validation currently consumes a significant percentage of the design team and takes months of simulation time. This validation strain is bound to increase as the complexity of designs increases; simulation alone cannot be expected to keep up with the verification problem. Purely formal techniques for verification have made considerable progress over the last decade but still fall short of providing the breadth and depth of verification required for most integrated circuits, and require considerable investment of time and energy on the designer's part.

We have developed a validation approach that lies between *simulation and formal verification, and uses a single specification model. The approach consists of a "golden"* deterministic *behavior description model, and a validation procedure to verify the equivalence between this golden model and any valid implementation of it. In this paper we present our approach where* Linear Temporal Logic (LTL) with bounded future-time operators *is used for behavioral property specification. Simulation monitoring and assertion checking is used in a state-of-the-art mixed Verilog/VHDL simulator (Mentor's ModelSim) in order to validate the design implementation against the desired properties.*

1 Introduction

The single biggest bottleneck in the Register Transfer (RT) level design today is ensuring that the design description, coded in a popular hardware description language (usually at the RT level), meets its specification. A battery of different techniques are currently applied in an attempt to ensure that this design interpretation is correct. The current challenge is that the amount of simulation (usually RTL) required to verify a circuit grows faster than linearly with the size of the circuit.

While formal verification can be a useful tool for early error detection at high abstraction design levels[1], extensive simulation is still the most common technique for RTL validation, the aim of this work is to bridge the gap between simulation and formal verification by incorporating formal property checks into simulation.

2 Related Work

There is a wide variety of approaches to the problem of design verification which seek to integrate formal techniques with simulation. These approaches include:

- Evaluating simulation coverage: Describing the coverage of a set of simulation vectors against a semi-formal model of the design being verified.
- Symbolic simulation: Improving the power of simulation techniques by using multi-valued simulation variables or formal analysis of the simulation output.

*SRC Graduate Fellow

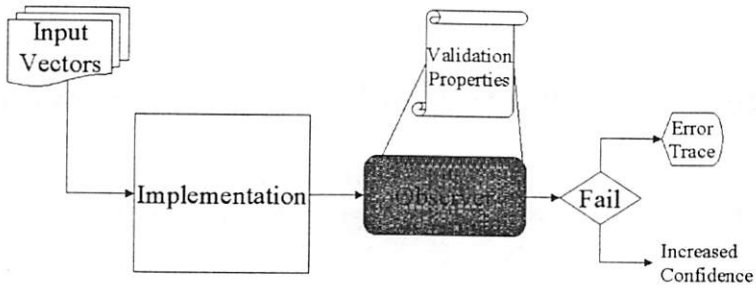

Figure 1: The Behavioral Verification Problem

- Building a transition system from the simulation semantics for model checking (such as in [3]).
- Formally directed simulation: Using a formal model of the design to be verified in order to direct simulation vectors.
- Assertion checking: Integration of formal assertions within a simulation framework [2].

In this effort, we will be concerned mainly with the last two points above: *Formally directed simulation* and *Assertion checking.*

3 Behavioral Verification

Figure 1 shows how we view the behavioral verification problem. In this effort we address and propose the following:

- A **property specification language** to describe the *observer* block whose job is to verify the conformance between the desired behavior and the implementation[5]. The language includes both *propositional*, and *temporal* logics to capture the static, and dynamic behavior respectively of the system,
- a **validation procedure** that operates on formulae in this language, and
- an **implementation** in a suitable simulation environment, that preserves the language semantics.

4 Observer Specification Language

Since we will be using simulation, and performing a trace check in order to observe the behavior of the system, our notion of conformance is that every *finite sequence* of observations that may result from executing the detailed implementation may also result from executing the more abstract "golden" specification model.

4.1 Specification Language Syntax

In general designers have a deterministic finite specification model in mind, non-determinism and liveness properties are used merely to simplify specification in the rather complex formal verification world. We believe that **Linear Temporal Logic (LTL) with bounded future-time operators** is quite suitable for simulation-oriented environments and is sufficient for most practical purposes that designers have in mind.

However, in order to make specification both intuitive for simulation, and applicable for later formal verification (if required), we decided to use CTL notation. In fact, checking a formula over a finite simulation trace is equivalent to model checking over an observation structure (also known as Kripke structure) where each state has one successor. In this context the semantics of the CTL operators **EG**, **EF**, **EX**, and **EU** coincide with the semantics of the corresponding linear time operators provided it is understood that all future-time operators

Operator	Meaning
AG	For all paths, at every point in time
AF	For all paths, at some point in time
AX	For all paths, at the next point in time
AU	has two operands *A[qUr]*.It means that for all paths, *q* is true until *r* is true
EG	For some path, at every point in time
EF	For some path, at some point in time
EX	For some path, at the next point in time
EU	has two operands *E[qUr]*. It means that for all paths, *q* is true until *r* is true

Table 1: Basic CTL Operators and their Meaning

are bounded by the time of the simulation completion. These operators will be the work-horse of the property specification mechanism.

4.1.1 Propositional Formulae

Propositional formulae are Boolean formulae and have the usual associated operators and semantics, but are evaluated *lazily* [2]. In other words, assignments are *evaluated only* to see if they hold (**true**) or not (**false**).

4.1.2 Temporal Formulae

CTL stands for Computation Tree Logic. The eight basic temporal logic operators describe computation *paths* in that tree. Table 1 lists the basic operators and their meaning (Adapted from [8]). A temporal logic operator consists of two parts: the *path quantifiers* and the *temporal operator*. There are two path quantifiers: **A** which means *on all paths*, and **E** which means *for some path*. There are four basic temporal operators: **G** which means *always*; **F** which means *eventually*; **X** which means *next*; and **U** which means *until*.

Since in simulation traces the branching operators have no meaning, we will negate the formulae that contain **A** thus changing those into the equivalent **E** formulae. In fact, path quantors have no significance in the context of LTL since we have a specific chosen path, therefore we go one step further and remove the **E** from formulae, and work with temporal quantifiers only.

4.1.3 Specification Language Semantics

An excellent discussion of the semantics of using traces in order to perform formal verification is presented in [2]. Our notions are quite similar to those presented by Canfield et. al., but we do not use a custom simulation environment; our semantics are integrated in a straightforward fashion into the *Discrete Event (DE)* domain semantics. In what follows, we discuss the semantics of the chosen specification language, when possible we use the same terminology used by Canfield et. al. in their work, and build on the concepts they introduced.

The semantics associated with our specification language are consistent with simulation concepts in a cycle-based simulator or an event-based simulator. Therefore our system can handle *any* HDL abstract or primitive data type[1]. *Signals* represent values on wires, and can be of any HDL type. *Variables* are temporary locations were computations are stored and can also be any valid HDL data object.

The values of signals and variables can be observed at discrete time points, each such interval is referred to as a *cycle*. The *state* of a module is a valuation of all its observable signals at the observation point at the end of a cycle. A simulation *trace* is a finite sequence consisting of the observed state of some module. A *golden behavior model* of some module is an unordered collection of formulae in the observer that conform to the syntax given earlier. Note that since we will be working from within the simulator we can observe **all** signals and variables no

[1]This will depend on the actual HDL, VHDL for example supports Abstract Data Types (ADTs) while Verilog does not

matter what their scope is[2]. Our role will therefore be that of a monitor only, we will not be tampering with any internal data structures (like the event or timing queue for example).

Rule violation is defined in terms of the first cycle of the trace, which corresponds to the usual way temporal logic is evaluated relative to an observation structure. Since simulation operates on finite traces, there are *three* possibilities for the truth value of a specification rule: **true**, **false**, and **undetermined**.

We briefly mention here that we could conceivably expand on this 3-valued logic introduced in[2], by adding a probabilistic aspect to the *undetermined* value thus giving the user a *degree of confidence* measure. This probabilistic concept is not new to the simulation domain at large, it is similar to concepts like probabilistic fault coverage, but to our knowledge has never been applied to account for the finiteness of simulation traces before. This feature is provided as a convenience to the user and involves only a minor syntactic change in the monitor procedures. To avoid distraction we will neglect this aspect in the sequel.

4.2 Validation Procedure

During a validation run, the observer will use its LTL specification to monitor *dynamically* the progress of the implementation under simulation. This approach should be contrasted to a *static* trace comparison of the implementation against the formal "golden" simulation trace, which is quite time and resource consuming[3] with little or no tangible result since it is hard to get a useful and reproducible error trace. Also, most trace comparison methods are not very complex and involve local comparisons between traces (i.e. they do not keep state), methods that do have an even larger overhead.

Because of the existence of temporal operators in the specification, the evaluation result cannot be known immediately in the same cycle since this result can depend on valuations at future cycles[2]. All future operators are however bounded with a upper limit, once that cycle is reached the `undetermined` valuation will be turned to `true` or `false` as the case may be.

The validation procedure we have presented can be qualified as *simulation of LTL formula* for each property to be validated and is polynomial in the size of the LTL formula. Evaluation optimizations that improve efficiency of this procedure include:

- A property determined to be `false` or `true` is dropped from the active list of properties to be checked.
- A "sensitivity list" is used for properties so that those whose symbols (variables and signals) have not changed from the previous cycle are not re-evaluated.

5 Implementation of the Simulation-Oriented Verification Approach

We have implemented our verification approach in a state-of-the-art mixed Verilog/VHDL simulator *Mentor's ModelSim*.

We provide the user with a *Formal Assertion Interface* where he/she can enter the formula in the notational subset of CTL we defined earlier (i.e. no path quantors, in effect it is LTL) provided in the form of a *Formal Assertion Library* (see the Appendix). The user can write assertions about **any** signal or variable in the design. The interface is integrated into the simulator so the user need only copy and paste signal and variable names and write any formula in LTL consisting of temporal and boolean propositions, and valuations for these signals and variables. In fact, we give the user a sample template of how to specify properties to be validated (see Section 6 for a sample template).

Once editing of this input file[4] is complete, the entered description is used to automatically generate the observer monitor that formally guides the simulation and checks the formal assertions at *each cycle*. A test bench is used in order to provide the simulation test vectors. The user can also choose to exercise the design using this

[2] This is a marked difference from other approaches that have to modify the given design description in order to make internal variables and signals visible to the outside observer

[3] The static trace comparison approach requires running the entire simulation while an error could have occurred early on in the run

[4] Actually it is a Tcl script but the user deals only with *macros* and cumbersome syntax is minimized

script (thus guiding the simulation) in a **formal** or **random** manner using the `force` macro (see [7] for flags and syntax).

We chose to use `Tcl` because it seems to be the most suited for the task at hand i.e. implementing a monitor script, and most simulators leave such a handle for users. Alternatives include writing the monitor processes in VHDL, or using the VHDL Foreign Language Interface (FLI) and then implementing the algorithms in C code for example. Both these latter approaches, however, suffer from a large overhead (code size, and execution time), and have serious limitations on the observations of signals and variables.

Note that the monitor overhead consists of evaluating the validity of the entered properties at each clock cycle[5]. The cost is therefore additive and *linear* in the number of properties. We do not interfere in any of the inner workings of the simulator and preserve its discrete event operation.

6 Hardware Verification Example: Cache Coherency Protocol

In this section we describe our validation experiments with a representative digital, synchronous, control-dominated design that has been used as a typical benchmark [6] in the literature on applied formal verification: a cache coherency protocol. The implementation example is taken from the VIS [9] release examples.

Protocols that maintain coherency for multiple processors are called *cache coherency protocols.* The protocol design described here is *directory based* that is the information about any one block of physical memory is kept in just one location. Information in the directory usually includes which caches have copies of the block, and whether it is dirty or not [4]. The design was implemented in Finite State Machine with Data path (FSMD) Register Transfer Level (RTL) VHDL.

6.1 List of Properties to be Validated

A list of properties to be checked is as follows (taken from VIS release [9]):

1. Liveness: If a processor requests a read then it will eventually be serviced.
2. Safety: If a cache has exclusive access to a block, then the other cache has no access to that block
3. Safety: If the block is in state SHARED in the cache controller, its corresponding bit is set in the Read List (Rlist) of the directory.
4. Safety: If the block is in state EXCLUSIVE in the cache controller, its corresponding bit is set in the Write List (Wlist) of the directory

6.2 Validation Using the Simulation-Oriented Approach

We validated the properties listed in Section 6.1 in the ModelSim VHDL simulator augmented with formal assertion. The following subsections provide more detail.

For the specific example at hand we wrote the following *observer script* using the macros provided in the formal assertion library (included in the Appendix).

```
# Cache Coherency Example (Property 4 shown only)
restart -force

# Formal Assertions Library
source ltl.tcl

# If the block is in state EXCLUSIVE in the cache controller,
# then its corresponding bit is set in the wlist of the directory
set cache2_blk_state_not_excl {[exa /test/coh/cache2/line__27/ \
```

[5] at the positive edge of the clk as currently implemented

Figure 2: The ModelSim with Formal Assertions Console

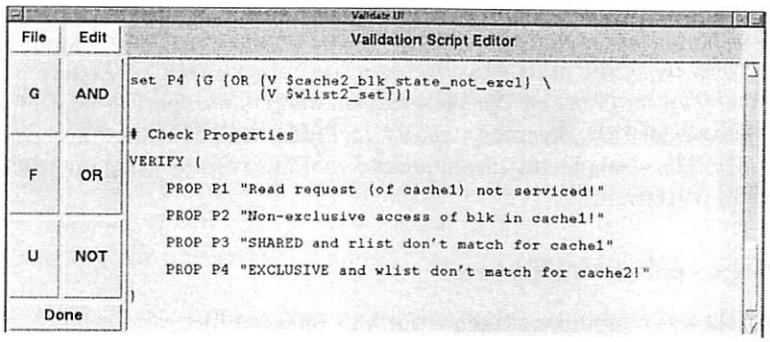

Figure 3: The Observer Script Editor

```
cache_blk_state] != "exclusive"}

set wlist2_set {[exa /test/coh/dir/line__35/ \
cache_wlist2([exa /test/coh/cache2/line__27/cache_blk_addr])] == "1"}

set P4 {G {OR {V $cache2_blk_state_not_excl} \
      {V $wlist2_set}}}

# Check Properties
VERIFY {
    PROP P4 "EXCLUSIVE and wlist don't match for cache2!"
}
```

6.2.1 Sample Simulation Run

In this section we attempt to demonstrate a simulation run of the above example with several screen captures.

Figure 2 shows the simulator console with the formal assertion interface, while Figure 3 displays the script editor.

In Figure 4 we show a simulation run at the time the simulator determines that property 4 (see Section 6.1) has been violated[6]. The relevant signal and variable waveforms are also displayed.

[6] This is consistent with the results of VIS

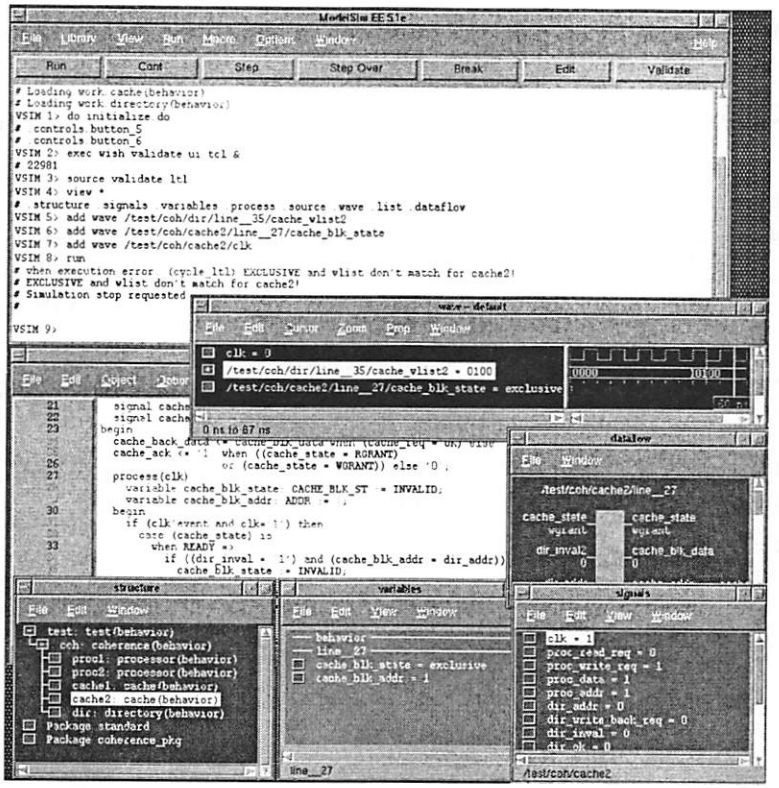

Figure 4: Screen Capture of a Simulation Run

7 Conclusions and Future Work

We believe that our work provides a tremendous added value for behavioral and RTL simulators in any shape or form, and promises to incorporate sound and appropriate formal approaches to the simulation world. By using a simulation-oriented approach, we need not explore the whole space as in formal verification to determine if a condition ever happens or not. The user may only be interested in a few specific guarantees and does not want to evaluate all possible scenarios as in the classical verification approach. The designer can therefore use our proposed approach to improve the coverage.

In this paper we have presented the validation procedure used to verify the equivalence of the "golden" simulation model and any valid implementation of it. We have developed a proof-of-concept simulation monitor, within a commercial mixed HDL simulator (Mentor's ModelSim). We use Tcl/Tk scripts in order to formally direct the simulation, monitor its progress, and perform the required assertion checking. We have also demonstrated our approach using a typical design benchmark from the verification domain: a cache coherence protocol. We leave evaluating the approach on industrial size examples for the future.

References

[1] Balarin F.; Chiodo M.; Giusto P.; Hsieh H.; Jurecska A.; Lavagno L.; Passerone C.; Sangiovanni-Vincentelli A. L.; Sentovich E.; Suzuki K.; and Tabbara B., "Hardware-Software Co-Design of Embedded Systems: The POLIS Approach", *Kluwer Academic Publishers*, MA, USA, May 1997.

[2] W. Canfield, E. A. Emerson, A. Saha "Checking Formal Specifications under Simulation" *ICCD*, 1997.

[3] E. Encrenaz "A Symbolic Relation for a Subset of VHDL'87 Descriptions and its Application to Symbolic Model Checking" *CHARME*, 1995.

[4] J. L. Hennessy, D. A. Patterson "Computer Architecture: A Quantitative Approach" *Morgan Kaufmann*. 1990.

[5] T. A. Henzinger, S. Qadeer, S. K. Rajamani "You Assume, We Guarantee: Methodology and Case Studies" *CAV*, 1998.

[6] Abhijit Jas, Alper Sen, Anand Ramachandran, Cagdas Akturan, ChiaBin Liu, Debaleena Das, I-Min Liu, Jayanta Bhadra, Justin R. Denison, Kaustubh Das, Malay K. Ganai, Padmini Gopalakrishnan, Parminder S. Chhabra, Praveen K. Jaini, Rajat Chaudhry, Ram Narayan, Ritu Chaba, Sriraman Padmanabhan, Srivatsan Srinivasan, Wasim U. Quddus, Zhao Zhe. "Examples of HW Verification using VIS", *Texas 97 Verification Benchmarks*, 1997.

[7] ModelSim EE/PLUS Reference Manual "Simulator Command Reference" *Reference Manual of ModelSim EE 5.1e*, 1998.

[8] T. Schlipf, T. Buechner, R. Fritz, M. Helms, J. Koehl "Formal Verification Made Easy" http://www.almaden.ibm.com/journal/rd/414/schlipf.html

[9] The VIS Group, "VIS: A system for Verification and Synthesis" *CAV*, 1996.

Appendix

Formal Assertion Library

```
proc V {expr} {
    return [expr $expr]
}

proc F {expr} {
    set sub [uplevel $expr]
    if {$sub == 1} {
return 1
    } else {
if {$sub == 0} {
    return [list F $expr]
} else {
    return [list OR $sub [list F $expr]]
}
    }
}

proc G {expr} {
    set sub [uplevel $expr]
    if {$sub == 0} {
return 0
    } else {
if {$sub == 1} {
    return [list G $expr]
} else {
    return [list AND $sub [list G $expr]]
}
    }
}

proc OR {lexpr rexpr} {
    set left [uplevel $lexpr]
    if {$left == 1} {
return 1
    } else {
set right [uplevel $rexpr]
if {$left == 0} {
    return $right
} else {
    if {$right == 1} {
return 1
    } elseif {$right == 0} {
return $left
    } else {
return [list OR $left $right]
    }
}
    }
}
```

```
proc AND {lexpr rexpr} {
    set left [uplevel $lexpr]
    if {$left == 0} {
return 0
    } else {
set right [uplevel $rexpr]
if {$left == 1} {
    return $right
} else {
    if {$right == 0} {
return 0
    } elseif {$right == 1} {
return $left
    } else {
return [list AND $left $right]
    }
}
    }
}

proc PROP {prop msg} {
    upvar $prop P
    if {$P != 1 && $P != 0} {
set P [uplevel $P]
if {$P == 0} {
    error $msg
}
    }
}

proc VERIFY {props} {
    when -label cycle_ltl {clk'event and clk = '1'} $props
}
```