Formal Verification of Module Interfaces against Real Time Specifications

Arindam Chakrabarti — Dept. of EECS, UC Berkeley, USA, arindam@cs.berkeley.edu

Pallab Dasgupta, P.P. Chakrabarti, Ansuman Banerjee — Dept. of Computer Science & Engineering, Indian Institute of Technology Kharagpur, INDIA, pallab,ppchak,ansuman@cse.iitkgp.ernet.in

ABSTRACT

One of the main concerns of the designer of a circuit module is to guarantee that the interface of the module conforms to specific protocols (such as PCI Bus, AMBA bus or Ethernet) by which it interacts with its environment. The computational complexity of verifying such open systems under all possible environments has been shown to be very hard (EXPTIME complete [10]). On the other hand, designers are typically required to guarantee correct behavior only for specific valid behaviors of the environment (such as a valid PCI Bus environment). Designers attempt to model these behaviors through an appropriate test bench for the module. In this paper we present a *module verifier tool* based on a proposed real time temporal logic called Open-RTCTL, which allows combined specification of the correctness properties and the input environments. The tool accepts the design in a subset of Verilog. By making the designer specify the environment constraints, we are able to verify a module in isolation, and thereby avoid the state explosion problem due to composition of modules. We present experimental results on modules from the Texas-97 Benchmark circuits [14] to demonstrate the space/time efficiency of the tool.

Categories and Subject Descriptors

B.7.2 [**Hardware**]: Integrated Circuits—*Verification*

General Terms

Verification

Keywords

Formal Verification, Temporal Logic

1. INTRODUCTION

Temporal logic model checking [4, 5] has emerged as one of the most powerful techniques for automated formal verification of hardware. In this approach, the design is modeled as a finite state non-deterministic transition system. The correctness property that needs to be verified on the design is specified in terms of a temporal logic formula. Computation Tree Logic (CTL) is one of the most popular temporal logics, and is the formal specification language behind many verification tools such as SMV [12], VIS [15] and FormalCheck [6].

The bulk of research in model checking has focussed on the verification of *closed systems*, that is, systems which are self contained and have no external inputs. The truth of properties specified in temporal logics such as CTL, LTL, and their real time extensions TCTL [1] and RTCTL [8] are not constrained by the inputs to the system. In many cases, state transitions enabled by different inputs are treated as non-deterministic transitions, and the semantics of these logics are interpreted over such non-deterministic FSMs.

In the other extreme, there has been research on the verification of *open systems* where a property is true if it holds regardless of the inputs to the system. Verification of open systems (referred to as *module checking*) has been shown to be much more complex than the verification of closed systems [10, 11]. For example, while CTL verification in *closed systems* is polynomial in the size of the system times the length of the formula, CTL verification in modules is EXPTIME-complete.

A natural way to verify a module (or component) lies in appropriate modeling of the system as well as the environment under which it is expected to work. Working under the assumption of a closed system or an adversarial open system are unsuitable in practice.

The *Module Verifier Tool* presented in this paper originates from the observation that the designer of a module is typically required to guarantee that the module works correctly under certain *valid* environments only. Typically, the designer attempts to model these environments through an appropriate test bench for the design. For example, the designer of a PCI compliant device may be required to validate the device under valid PCI Bus environments only.

In this paper, we present a real time logic called Open-RTCTL which allows the designer to express the correctness property and the environment under which the property is expected to hold, in a unified way. This also allows the designer to decompose the correctness specification based on different input scenarios.

The Module Verifier Tool accepts the design in a subset of Verilog and provides a succinct temporal framework for specifying both the environment constraints as well as the correctness properties. Designers may use this tool to eliminate as many errors as possible from the modules they design before integrating them with the other modules in the circuit. By making the designer specify the environment constraints, we are able to verify a module in isolation, and thereby avoid the state explosion problem due to composition of modules. The tool is also useful in the *assume-guarantee* style of reasoning [9].

We present experimental results on the Texas-97 Verification Benchmarks [14]. We discuss a scheme to incorporate automatic generation of Open-RTCTL properties from protocols specified as *timing diagrams.*

The paper is organized as follows. Section 2 describes the architecture of the tool. Section 3 presents the logic Open-RTCTL which is used to specify correctness properties, and a symbolic model checking algorithm for Open-RTCTL verification. Section 4 outlines the abstraction mechanism used in our tool. Section 5 presents experimental results. Section 6 discusses translation of timing diagrams to Open-RTCTL properties.

2. THE ARCHITECTURE OF THE TOOL

The architecture of the Module Verifier Tool is shown in Fig 1. The tool accepts the design in a subset of Verilog. In the current version of the tool, the correctness specification is input as a set of properties specified in a language called *Open-RTCTL* which we present in the next section. The correctness specification includes both timed and untimed properties.

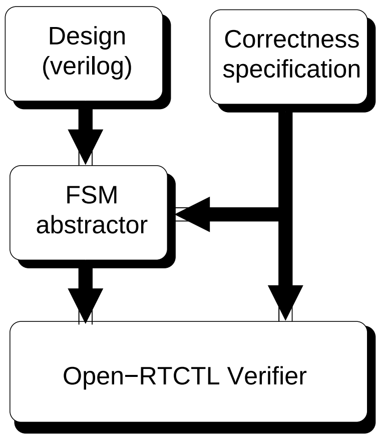

Figure 1: The Module Verifier Tool

From the given set of correctness properties, we extract the relevant state variables of the module which affect the interface protocol. Using a cone-of-influence reduction, we abstract a minimal FSM model of the given design (module). The Open-RTCTL verifier accepts the FSM model and the correctness properties as input and verifies the correctness of the design.

Timing diagrams are a popular form of representation of interface protocols. We present our approach to translate timing diagrams to Open-RTCTL properties. Since timing diagrams are commonly used in the industry, such an interface will alleviate the designer from having to form a complete set of Open-RTCTL properties which covers the correctness specification of the interface protocol.

3. CORRECTNESS SPECIFICATION

In this section, we outline the logic Open-RTCTL which forms the basis of the property specification syntax of the Module Verifier Tool. The untimed version of this logic (which does not express any quantitative timing properties) was presented in [7]. The logic presented here supports the *bounded until* operator, and can express timing properties.

Formally, we define a module as a tuple, $J = \langle \mathcal{AP}, S, \mathbb{I}, \tau, s_0, \mathcal{F}\rangle$, where:

- $\mathcal{AP}$ is a set of atomic propositions,
- S is a finite set of states,
- $\mathbb{I}$ is a finite set of inputs,
- $\tau : S \times 2^{\mathbb{I}} \rightarrow S$ is the next-state function. Given a state $s_i \in S$ and input vector η, $s_j = \tau(s_i, \eta)$ is the next state of s_i under input η,
- $s_0 \in S$ is the initial state,
- $\mathcal{F} : S \rightarrow 2^{\mathcal{AP}}$ is a labeling of states with atomic propositions true in that state.

A *path*, π, in the module is an infinite sequence of state-input pairs, $(s_0, \eta_0), (s_1, \eta_1), \ldots$, such that for all i, $s_i \in S$, $\eta_i \in 2^{\mathbb{I}}$ and $s_{i+1} = \tau(s_i, \eta_i)$. s_0 is called the starting state of π. Since the module has a finite set of states, one or more states will appear multiple number of times on a path. In other words, a path (as defined here) is an infinite *walk* over the state transition graph.

We further define a $\mathcal{I}$*-consistent* path as follows. A boolean constraint $\mathcal{I}$ over the inputs $\mathbb{I}$ is said to be satisfied by a input vector η iff $\mathcal{I}$ evaluates to true on the input η. Given a boolean constraint $\mathcal{I}$ over the inputs $\mathbb{I}$, a path $\pi = (s_0, \eta_0), (s_1, \eta_1), \ldots$ is $\mathcal{I}$*-consistent* if η_i satisfies $\mathcal{I}$ for all i. It may be noted that since the next state of a state is well defined for every input vector, there exists at least one $\mathcal{I}$-consistent path starting from each state whenever $\mathcal{I}$ is satisfiable. Since many different input vectors may satisfy $\mathcal{I}$, there may be multiple $\mathcal{I}$-consistent paths from a state.
The formal syntax of Open-RTCTL is as follows. p denotes an atomic proposition. a and b are positive integers.

- Each $p \in \mathcal{AP}$ is an Open-RTCTL formula,
- If f and g are Open-RTCTL formulas, then so are $\neg f$, $f \vee g$, $f \wedge g$, $EX^{\mathcal{I}} f$, $AX^{\mathcal{I}} f$, $E(f\ U^{\mathcal{I}}_{[a,b]}\ g)$, $A(f\ U^{\mathcal{I}}_{[a,b]}\ g)$, where $\mathcal{I}$ is a boolean formula over inputs in $\mathbb{I}$.

The syntax of Open-RTCTL is obtained by augmenting the X and U operators of RTCTL [8] with an input constraint, $\mathcal{I}$. We use $E(f\ U^{\mathcal{I}}\ g)$ as a shortform for $E(f\ U^{\mathcal{I}}_{[0,\infty]}\ g)$. We also use the usual short forms $EF^{\mathcal{I}} f$ for $E(\mathit{true}\ U^{\mathcal{I}}\ f)$, and $AG^{\mathcal{I}} f$ for $\neg EF^{\mathcal{I}} \neg f$.
The semantics of Open-RTCTL is defined over a module $J = \langle \mathcal{AP}, S, \mathbb{I}, \tau, s_0, \mathcal{F}\rangle$. We use the notation $s \models f$ to indicate that the Open-RTCTL formula f is true at state s of the module. Likewise, we use the notation $\pi \models \psi$ to indicate that a Open-RTCTL path formula, ψ (of the form $f U^{\mathcal{I}}_{[a,b]}\ g$) is true on the path π. The formal semantics of Open-RTCTL is as follows:

- $\forall s \in S, s \models \mathit{True}$ and $s \not\models \mathit{False}$
- $s \models p$ iff $p \in \mathcal{F}(s)$

- $s \models \neg f$ iff $s \not\models f$
- $s \models f \wedge g$ iff $s \models f$ and $s \models g$
- $s \models f \vee g$ iff $s \models f$ or $s \models g$
- $s \models EX^{\mathcal{I}} f$ iff $\exists s' = \tau(s, \eta)$ such that $s' \models f$ and the input vector η satisfies the constraint $\mathcal{I}$
- $s \models AX^{\mathcal{I}} f$ iff $s \models EX^{\mathcal{I}} f$ and $\forall s'$, such that $s' = \tau(s, \eta)$ for some η satisfying $\mathcal{I}$, we have $s' \models f$
- $\pi \models fU^{\mathcal{I}}_{[a,b]}\ g$ iff π is $\mathcal{I}$-consistent, and $\exists (s_i, \eta_i) \in \pi$ such that $a \le i \le b$ and $s_i \models g$, and for all (s_j, η_j) preceding (s_i, η_i) in π, $s_i \models f$
- $s \models E(fU^{\mathcal{I}}_{[a,b]}\ g)$ iff there exists a path π starting from s, such that $\pi \models fU^{\mathcal{I}}_{[a,b]}\ g$
- $s \models A(fU^{\mathcal{I}}_{[a,b]}\ g)$ iff $s \models E(fU^{\mathcal{I}}_{[a,b]}\ g)$ and for each $\mathcal{I}$-consistent path π starting from s, we have $\pi \models fU^{\mathcal{I}}_{[a,b]}\ g$

As mentioned earlier, an $\mathcal{I}$-consistent path always exists from a state s provided that $\mathcal{I}$ is satisfiable. In a Open-RTCTL formula, $\mathcal{I}$ is used to specify the possible input patterns under which we are to verify the module, and hence it is natural to assume that $\mathcal{I}$ is non-empty (satisfiable). Otherwise, the formulas are vacuously false at all states. It is for this reason that we have the requirement of $s \models EX^{\mathcal{I}} f$ in the semantics of $s \models AX^{\mathcal{I}} f$ and the requirement of $s \models E(fU^{\mathcal{I}}_{[a,b]}\ g)$ in the semantics of $s \models A(fU^{\mathcal{I}}_{[a,b]}\ g)$.

Though in the formal definition of a module, we have an explicit transition from a state s_i for every input vector, it is often natural to expect that the set of input vectors enabling a transition of the module is succinctly specified in the form of a boolean formula. It is convenient to construct BDDs for such succinct transition relations without actually creating the explicit FSM. The following example illustrates a typical succinct module where each transition is labeled by the boolean formula representing the set of input vectors for which the transition is taken. We also illustrate the syntax and semantics of Open-RTCTL through the following example.

Example 1. Figure 2 shows a simple module in succinct form with $S = \{s_1, s_2, s_3, s_4, s_5, s_6\}$, $\mathcal{AP} = \{f, g, h\}$ and $\mathbb{I} = \{i_1, i_2\}$. s_1 is the initial state. The atomic propositions true in a state is shown beside the state. The set of input vectors enabling a given transition is shown as a boolean formula beside the transition. For example, the transition from s_2 to s_1 is enabled by two input vectors, namely $(1, 0)$ and $(1, 1)$, which are represented by the boolean formula i_1.

Let us consider the following Open-RTCTL formulas:

$$\varphi_1 = E(f\ U^{\mathcal{I}}_{[0,2]}\ g)$$
$$\varphi_2 = A(f\ U^{\mathcal{I}}_{[0,2]}\ g)$$

The formula φ_1 asks whether *there exists* inputs satisfying the boolean constraint $\mathcal{I}$, which drives the system to a state satisfying g through states satisfying f. The formula φ_2 asks whether *all inputs* which satisfy $\mathcal{I}$ drive the system to a state satisfying g through states satisfying f.

Suppose $\mathcal{I} = i_1$. Let us examine the truth of the formulas at the state s_1. The constraint $\mathcal{I} = i_1$ enables the transitions from s_1 to s_3 and s_4. Since g is true at s_4, φ_1 is true at s_1.

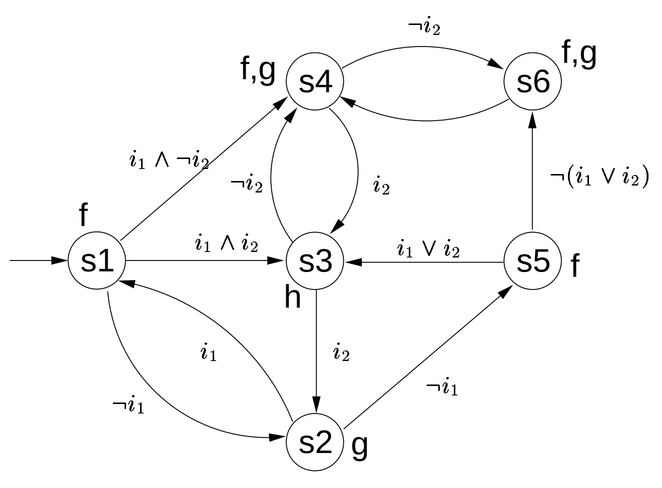

Figure 2: A simple module

On the other hand, since both g and f are false at s_3, φ_2 is false at s_1.

Suppose $\mathcal{I} = \neg i_1$. Now from s_1, only the transition to s_2 is enabled. Since g is true at s_2, both φ_1 and φ_2 are satisfied at s_1. Let us examine the truth of these formulas at s_5. The constraint $\mathcal{I} = \neg i_1$ enables the transition from s_5 to s_6. It also enables the transition from s_5 to s_3. This is because, the transition from s_5 to s_3 is enabled by a collection of three input vectors $\eta_0 = (0, 1)$, $\eta_1 = (1, 0)$ and $\eta_2 = (1, 1)$. Thus the transition shown in the diagram actually consists of three alternative transitions each enabled by a different input vector, but all leading to the same next state s_3. Since η_0 satisfies $\mathcal{I} = \neg i_1$, the transition from s_5 to s_3 is enabled. It is now easy to see that φ_1 is true at s_5, but φ_2 is false at s_5 since g is not true at s_3.

Let us now consider the following Open-RTCTL formulas, where $\mathcal{I} = i_1$:

$$\varphi_3 = E(f\ U^{\mathcal{I}}_{[3,5]}\ h)$$
$$\varphi_4 = E(f\ U^{\mathcal{I}}_{[5,5]}\ h)$$

φ_3 is true at s_1, due to the existence of the path $s_1, s_4, s_6, s_4, s_3, \ldots$, where the state s_3 satisfying h occurs at the 4^{th} cycle, and all previous states satisfy f. On the other hand, φ_4 is false at s_1, since there is no satisfying path from s_1 which reaches s_3 in the 5^{th} cycle.

The syntax of Open-RTCTL allows us to nest formulas with different input constraints. For example, consider the following query:

> *From the state s_1, by applying inputs satisfying the constraint $i_1 \wedge i_2$ are we always able to reach within 4 cycles any state satisfying g where a negative edge of the input i_1 produces a state satisfying f?*

The above query can be expressed in Open-RTCTL as a formula:

$$\varphi = A(true\ U^{\mathcal{I}_1}_{[0,4]}\ g \wedge AX^{\mathcal{I}_2}\ f)$$

where $\mathcal{I}_1 = i_1 \wedge i_2$ and $\mathcal{I}_2 = \neg i_1$. The subformula $g \wedge AX^{\mathcal{I}_2}\ f$ is true at the states s_2 and s_6. Now, all $\mathcal{I}_1$-consistent paths from s_1 have the prefix $s_1, s_3, s_2, \ldots$, which reaches s_2 in 2 cycles. Hence φ is true at s_1. □

3.1 Symbolic Open-RTCTL verification

In this section we outline a BDD-based symbolic model checking algorithm for Open-RTCTL verification. The next-state function τ is represented as a BDD for the relation, $N(V, I, V')$ where V denotes the present state, I denotes the input vector, and V' denotes the next state.

The verification procedure *Check* takes the Open-RTCTL formula to be checked as its argument and returns an OBDD that represents the states of the system which satisfy the formula. The procedure *Check* recursively handles formulas of the form $EX^{\mathcal{I}}$ f, $AX^{\mathcal{I}}$ f, E[f $U^{\mathcal{I}}_{[a,b]}$ g] and A[f $U^{\mathcal{I}}_{[a,b]}$ g] as follows:

Check($EX^{\mathcal{I}}$ f) = CheckEX($\mathcal{I}$, Check(f))
Check($AX^{\mathcal{I}}$ f) = CheckAX($\mathcal{I}$, Check(f))
Check(E[f $U^{\mathcal{I}}_{[a,b]}$ g]) = CheckEU(a, b, $\mathcal{I}$, Check(f), Check(g))
Check(A[f $U^{\mathcal{I}}_{[a,b]}$ g]) = CheckAU(a, b, $\mathcal{I}$, Check(f), Check(g))

Given the BDD $N(V, I, V')$ for the transition relation, the BDD $C(I)$ for the input constraint $\mathcal{I}$, and the BDDs $f(V')$ and $g(V')$ returned by Check(f) and Check(g) respectively, we have:

CheckEX($\mathcal{I}$, Check(f)) =
  $\exists V' \exists I\, [f(V') \wedge N(V, I, V') \wedge C(I)]$
CheckAX($\mathcal{I}$, Check(f)) =
  $\forall V' \exists I\, [f(V') \wedge N(V, I, V') \wedge C(I)]$
CheckEU(0, 0, $\mathcal{I}$, Check(f), Check(g)) = $g(V')$
CheckEU(0, b, $\mathcal{I}$, Check(f), Check(g)) =
  $g(V') \vee (f(V') \wedge CheckEX(\mathcal{I}, Z(V')))$
  where $Z(V')$ = CheckEU(0, b-1, $\mathcal{I}$, Check(f), Check(g))
CheckEU(a, b, $\mathcal{I}$, Check(f), Check(g)) =
  $g(V') \vee (f(V') \wedge CheckEX(\mathcal{I}, Z(V')))$
  where $Z(V')$ = CheckEU(a-1, b-1, $\mathcal{I}$, Check(f), Check(g))

We compute CheckAU() similarly, except that we use CheckAX() instead of CheckEX(). For the *unbounded until operators*, we use the usual fixpoint computations [3, 5]:

CheckEU(0, ∞, $\mathcal{I}$, Check(f), Check(g)) =
  lfp $Z(V')\, [g(V') \vee (f(V') \wedge CheckEX(\mathcal{I}, Z(V')))]$
CheckAU(0, ∞, $\mathcal{I}$, Check(f), Check(g)) =
  lfp $Z(V')\, [g(V') \vee (f(V') \wedge CheckAX(\mathcal{I}, Z(V')))]$

4. DESIGN EXTRACTION

Extraction of a FSM from a high level language such as Verilog is one of the most difficult tasks in formal verification. In this section we highlight some of the salient features of this part of our tool.

The tool supports a subset of Verilog HDL. It supports single bit register and wire data types with boolean operators defined on them. Also supported are `initial` and event-guarded `always` blocks, procedural assignments, if-then-else conditional statements, and continuous assignments.

Module instantiations and multi-bit data types are not supported in the current implementation. However, it is easy to translate a Verilog description into our supported subset. We have translated many of the modules from the Texas97 benchmark circuits (which are in Verilog) into our supported subset.

The tool performs a static analysis of the given design, during which it identifies the control and data dependencies between inputs, outputs, registers and wire variables. At this stage it detects whether race conditions exist in the circuit.

It is common practice to design modules which are *edge triggered*, that is, actions within the module are triggered by changes in the signal values. These are represented in Verilog by the `@posedge(signal)`, `@negedge(signal)`, and `@(signal)` constructs. In order to detect a change in a variable, the module must remember the value of the variable in the previous *fetch-update* cycle. In our tool, for a variable a which appears within a "@" construct, we create two variables, namely a and a' to represent respectively the current and previous value of the variable.

During the abstraction of a FSM from the given design, we use a cone-of-influence reduction based on the set of control variables that are of interest to us. Currently, this set of variables can be declared by the designer. Alternatively, the tool can extract the set of relevant variables from the set of Open-RTCTL properties that are to be verified.

5. RESULTS

We present experimental results of the tool on modules from the Texas-97 Verification Benchmarks [14]. These benchmarks consist of industrial grade circuit modules specified in Verilog. The verifier was run on a 333 MHz SUN sparc workstation with 128 MB RAM over SunOS 5.8. Our tool uses the CUDD BDD-package [13] for generating the BDDs.

Table 1 shows the results of our tool. The FSM reduction time includes the time required by the design analyzer to parse the circuit, the time required by the abstraction algorithm to prune the circuit, and the time required by CUDD to create the BDDs. The size of the reduced BDDs are shown in the third column. The verification time denotes the time required after extraction by the Open-RTCTL verifier to check the number of properties given in the fourth column. All times refer to CPU time.

It is important to note that since we are able to analyze one module at a time, the BDD sizes are within feasible limits. This supplements the efficiency of the Open-RTCTL verifier in the overall complexity of verification. As a result most of the modules were verified in less than a second, which appears promising considering the size of the original Texas-97 benchmark circuits.

6. TIMING DIAGRAMS

In industrial practice, it is common to represent the timing specification of an interface protocol in the form of a timing diagram. This has prompted researchers in the formal verification community to develop techniques for automated verification of circuits against their timing diagrams [2].

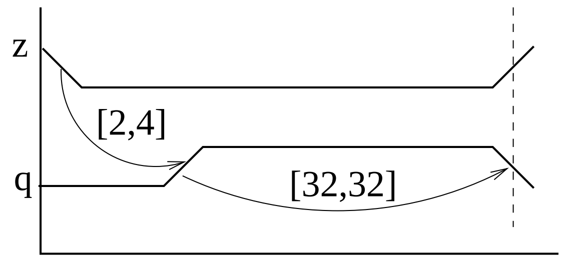

Figure 3: Sample Timing Diagram

We are integrating a technique for translating constraints specified using timing diagrams into Open-RTCTL proper-

Circuit Module	FSM Reduction & BDD Creation time (sec)	BDD size (no of nodes)	No. of Open-RTCTL Queries	Verification time (sec)
PCI Local Bus				
PCI Master	36.50	14786	3	2.30
PCI Target	28.62	34859	3	7.60
MSI Cache Coherence Protocol				
3-proc sys arbiter	0.11	249	3	0.12
2-proc sys arbiter	0.30	170	2	0.60
MPEG System Decoder				
timestamp	0.18	1140	2	0.98
prefixcode	0.05	173	2	0.14
parsepack	0.16	1159	2	0.72
packstart	0.10	324	2	0.37
PI Bus				
slave	1.80	4004	2	1.83

Table 1: Results of Open-RTCTL Verification

ties. Unlike previous work in verification of timing diagrams, we consider open systems. Therefore, some of the waveforms in a timing diagram are inputs to the given module, where as some others are expected outputs of the given module. Our task is to verify the expected output waveform when presented with the given input waveform.

As an example, consider the timing diagram shown in Fig 3. Suppose the given module M has input z and output q as shown in the figure. The protocol starts with `negedge(z)` which is expected to cause a `posedge(q)` between 2 to 4 cycles. Thereafter, exactly after 32 cycles both z and q toggle. Further, by virtue of the start of the protocol, we are given that initially z is high and q is low.

To verify whether a given implementation of M satisfies the timing diagram, we verify the following Open-RTCTL formula at states which serve as possible initial states of the protocol:

$$AX^{\neg z}\ A(\neg q\ U^{\neg z}_{[2,4]}\ (q \wedge A(q\ U^{\neg z}_{[32,32]}\ \neg q)))$$

Note that through this formula, we do not assert that z toggles exactly 32 cycles after `posedge(q)` occurs. For a compositional verification of the whole system, this has to be verified on the module which produces z as its output.

7. CONCLUSIONS

Designers like to eliminate as many errors as possible from the modules they design before integrating them with the other modules in the circuit. The Module Verifier Tool provides a simple way to analyze a single module at a time and allows the designer to express, in a unified way, the property to be verified and the inputs for which it is expected to hold. This can eliminate many bugs prior to simulation or compositional verification. Open-RTCTL is also useful as a specification language for assume guarantee styles of reasoning.

8. ACKNOWLEDGEMENTS

The authors acknowledge the support of Sun Microsystems, USA, for this work. Pallab Dasgupta further acknowledges the support of the Indian National Science Academy. P.P. Chakrabarti further acknowledges the support of Dept. of Sci & Tech, Govt. of India.

9. REFERENCES

[1] Alur, R., and Henzinger, T., Real time logics: Complexity and Expressiveness. *Information and Computation*, **104**, 1, 35-77, 1993.

[2] Amla, N., Emerson, E.A., Kurshan, R.P., and Namjoshi, K., RTDT: A Front-End for Efficient Model Checking of Synchronous Timing Diagrams, In *Proc. of CAV'2001*, 2001.

[3] Burch, J.R., Clarke, E.M., Long, D.E., McMillan, K.L., and Dill, D.L., Symbolic model checking for sequential circuit verification. *IEEE Trans. on CAD*, **13**, 4, 401-424, 1994.

[4] Clarke, E.M., Emerson, E.A., and Sistla, A.P., Automatic verification of finite-state concurrent systems using temporal logic specifications, *ACM Trans. on Prog. Lang. and Sys.*, 8(2):244-263, 1986.

[5] Clarke, E.M., Grumberg, O., and Peled, D.A., *Model Checking*, MIT Press, 2000.

[6] *Cadence Formalcheck Tool*, http://www.cadence.com/datasheets/formalcheck.html

[7] Dasgupta, P., Chakrabarti, P.P., and Chakrabarti, A., Open Computation Tree Logic for Formal Verification of Modules, In *Proc. of ASPDAC/VLSI Design Joint Conference*, Jan 2002, India.

[8] Emerson, E.A., Mok, A.K., Sistla, A.P. and Srinivasan, J., Quantitative temporal reasoning, In *Proc. of CAV'99*, France, 1990.

[9] Grumberg, O., and Long, D.E., Model Checking and Modular Verification, *ACM Trans. on Prog. Lang. and Sys.*, 16:843-872, 1994.

[10] Kupferman, O. and Vardi, M.Y., Module Checking. In *Proc. of CAV'96*, LNCS 1102, 75-86, 1996.

[11] Kupferman, O. and Vardi, M.Y., Module Checking revisited. In *Proc. of CAV'97*, LNCS 1254, 1997.

[12] *The SMV Model Checker*, http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/

[13] Somenzi, F., *CUDD: CU Decision Diagram Package, Release 2.3.0, User's Manual*, Dept. of Electrical and Computer Engg., Univ. of Colorado, Boulder, 1998.

[14] *Texas-97 Verification Benchmarks*, http://www-cad.eecs.berkeley.edu/Respep/Research/vis/texas-97/

[15] VIS: A system for verification and synthesis, The VIS Group, In *Proc. of CAV'96*, LNCS 1102, 1996.