View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Publications of the IAS Fellows

Open Computation Tree Logic for Formal Verification of Modules

Pallab Dasgupta* Arindam Chakrabarti*
P.P. Chakrabarti*
Department of Computer Science & Engineering,
Indian Institute of Technology, Kharagpur, INDIA 721302
pallab,ppchak@cse.iitkgp.ernet.in

Abstract

Modules of large VLSI circuits are often designed by different designers spread across the globe. One of the main challenges of the designer is to guarantee that the module he/she designs will work correctly in the global design, the details of which, is often unknown to him/her. Modules are open systems whose behavior is subject to the inputs it receives from its environment. It has been shown that verification of open systems (modules) is computationally very hard (EXPTIME complete [7]) when we consider all possible environments. On the other hand we show that integrating the specification of the properties to be verified with the specification of only the valid input patterns (under which the module is expected to function correctly) gives us a powerful syntax which can be verified in polynomial time. We call the proposed logic Open-CTL (CTL for open systems). The convenience of being able to specify the property and the environment in a unified way in Open-CTL is demonstrated through a study of the PCI Bus properties. We present a symbolic BDD-based verification scheme for checking Open-CTL formulas, and present experimental results on modules from the Texas-97 Verification Benchmark circuits [12].

1 Introduction

Components of a large VLSI circuit are often designed in different places by different designers. Due to the increasing complexity of the overall designs, it has become very important for the designer to guarantee that the component he/she designs will work in all valid environments where it is to be used. With the recent emphasis on component reusability, this has become even more significant, since the environment where it is reused may be entirely different from the one for which it was originally designed.

Temporal logic model checking [3, 4] 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 [10], VIS [13] and FormalCheck [5].

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. While this may be the case for the circuit as a whole, it is certainly not the case for the individual modules in the circuit which will typically receive inputs from the other modules. A module is an *open system* whose behavior is a function of the inputs it receives from its environment. Verification of open systems (referred to as *module checking*) has been shown to be much more complex than the verification of closed systems [7, 8]. 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.

In reality, the designer is typically required to guarantee that the module works correctly under certain *valid* environments only. These are the environments that the test bench attempts to model. For example, the designer of a PCI compliant device may be required to validate the device under valid PCI Bus environments.

In this paper, we present a temporal logic called *Open-CTL* for the verification of open systems (modules). The novelty of this logic is that it integrates the specification of the CTL property to be verified with the valid input patterns under which it is expected to hold. For example, while the CTL property EFq asks whether q holds sometime in the future, the Open-CTL property EF_Iq asks whether q holds

*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.

sometime in the future provided we continue to assert the input I. We show that Open-CTL is useful for expressing properties of modules, since it allows the specification of properties and input constraints in a unified way. We demonstrate the expressibility of Open-CTL through examples and a study of the PCI Bus protocol.

Open-CTL allows us to validate a module in absence of details about the other modules in the design. Since we do not have to consider all the modules together (to get a closed system) we avoid the state explosion problem which is the main bottleneck in model checking. Open-CTL is also useful in the *assume-guarantee* style of reasoning [6], where conditions guaranteed by the other modules in the design are used as assumptions about the environment of the module to be verified.

We establish that Open-CTL verification is also computationally attractive. Given a module which is explicitly specified as a FSM, Open-CTL verification works in time linear in the size of the module times the length of the Open-CTL formula. While this is good news, it is usually the case that module descriptions specified in high-level languages such as Verilog or SystemC are succinct in nature. Extracting the FSM explicitly from such specifications is by itself an expensive task.

On the other hand it is more convenient to translate the succinct description of a module to a equivalent BDD representation (which is also succinct). Our Open-CTL verification tool performs this task by accepting Verilog descriptions as input. We then present a symbolic BDD-based method for verifying Open-CTL formulae and present experimental results on the Texas-97 Verification Benchmarks [12].

The paper is organized as follows. The first two sections presents the foundations of Open-CTL. In Section 2, we present the syntax and semantics of Open-CTL, while in Section 3, we establish the complexity of Open-CTL verification on explicit modules. In Section 4 we consider succinct modules and illustrate the semantics of Open-CTL on succinct modules through examples. Section 5 presents a symbolic BDD-based verification method for Open-CTL verification in succinct modules. Section 6 demonstrates the expressibility of Open-CTL in modelling properties of PCI compliant master and slave devices. We present experimental results in Section 7.

2 Syntax and Semantics

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,
- $\tau : S \times 2^{\mathbb{I}} \to S$ 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,
- $\mathcal{F} : S \to 2^{\mathcal{AP}}$ 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 *I-consistent* path as follows. A boolean function I over the inputs $\mathbb{I}$ is said to be satisfied by a input vector η iff I evaluates to true on the input η. Given a boolean function I over the inputs $\mathbb{I}$, a path $\pi = (s_0, \eta_0), (s_1, \eta_1), \ldots$ is *I-consistent* if η_i satisfies 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 I-consistent path starting from each state whenever I is satisfiable. Since many different input vectors may satisfy I, there may be multiple I-consistent paths from a state.
The formal syntax of Open-CTL is as follows:

- Each $p \in \mathcal{AP}$ is an Open-CTL formula,
- If f and g are Open-CTL formulas, then so are $\neg f$, $f \vee g$, $f \wedge g$, $EX_I f$, $AX_I f$, $E(f\, U_I\, g)$, $A(f\, U_I\, g)$, where I is a boolean formula over inputs in $\mathbb{I}$.

The syntax is similar to CTL, except that the X and U operators are augmented with an input constraint, I. We use the usual short forms $EF_I f$ for $E(true\, U_I\, f)$, and $AG_I f$ for $\neg EF_I \neg f$.
The semantics of Open-CTL 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-CTL formula f is true at state s of the module. Likewise, we use the notation $\pi \models \psi$ to indicate that a Open-CTL path formula, ψ (of the form $f U_I\, g$) is true on the path π. The formal semantics of Open-CTL is as follows:

- $\forall s \in S, s \models True$ and $s \not\models False$
- $s \models p$ 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 $s \models g$
- $s \models EX_I f$ iff $\exists s' = \tau(s, \eta)$ such that $s' \models f$ and the input vector η satisfies the constraint I

- $s \models AX_I f$ iff $s \models EX_I f$ and $\forall s'$, such that $s' = \tau(s, \eta)$ for some η satisfying I, we have $s' \models f$

- $\pi \models f U_I\, g$ iff π is I-consistent, and $\exists (s_i, \eta_i) \in \pi$ such that $s_i \models g$, and for all (s_j, η_j) preceding (s_i, η_i) in π, $s_i \models f$

- $s \models E(f U_I\, g)$ iff there exists a path π starting from s, such that $\pi \models f U_I\, g$

- $s \models A(f U_I\ g)$ iff $s \models E(f U_I\ g)$ and for each I-consistent path π starting from s, we have $\pi \models f U_I\, g$

As mentioned earlier, an I-consistent path always exists from a state s provided that I is satisfiable. In a Open-CTL formula, 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 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 $s \models EX_I f$ in the semantics of $s \models AX_I f$ and the requirement of $s \models E(f U_I\, g)$ in the semantics of $s \models A(f U_I\, g)$.

3 Complexity of Open-CTL Verification

In this section we examine the complexity of Open-CTL verification in modules. It is easy to see that Open-CTL is strictly more expressive than CTL. This is because, Open-CTL reduces to CTL in the absence of input constraints, and whenever a input constraint is given, the Open-CTL formula cannot be expressed in CTL.

Theorem 1 *An Open-CTL formula, φ, can be verified at all states of a module, $J = \langle \mathcal{AP}, S, \mathbb{I}, \tau, s_0, \mathcal{F} \rangle$, in $O(|\varphi|.|I^*|.(|\tau| + |S|))$ time, where I^* is the length of the longest input constraint in φ.*

Proof: It is known that in the absence of input constraints, verifying a CTL formula ψ requires $O(|\psi|.(|\tau| + |S|))$ time [3]. Suppose φ is of the form $EX_I f$, $AX_I f$, $E(f\ U_I\ g)$ or $A(f\ U_I\ g)$. Given a transition (s_i, η, s_j) we can verify whether the input vector η satisfies I in $|I|$ time. Therefore, deleting all transitions not enabled by I can be done in $|I|.|\tau|$ time.

We now use induction on the length of the formula. Suppose the states of the transition systems are labelled by the subformulas of φ that are true in them. In order to verify φ, we first prune all transitions not enabled by I. We then perform model checking of the unconstrained formula on the pruned transition system using the labels of the subformulas, which requires $O(|I|.(|\tau| + |S|))$ time. Using the worst case size of the input constraint, and by induction on the length of φ, we obtain the complexity of verification as $O(|\varphi|.|I^*|.(|\tau| + |S|))$. $\square$.

4 Succinct Modules

In the definition of a module, we have a transition from a state s_i for every input vector. Theorem 1 gives us the complexity of Open-CTL verification, when these transitions are explicitly specified. However, designers often specify the conditions enabling a transition as a boolean formula f in the form of branching code with f as the branching condition, or in the form of synchronization formulas (say formulas using the @ construct in Verilog). It is therefore 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. We refer to such module specifications as succinct module.

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-CTL through the following example.

Example 1 Figure 1 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_1 to s_4 is enabled by two input vectors, namely $\eta_1 = (1, 0)$ and $\eta_2 = (1, 1)$, which are represented by the boolean formula i_1.

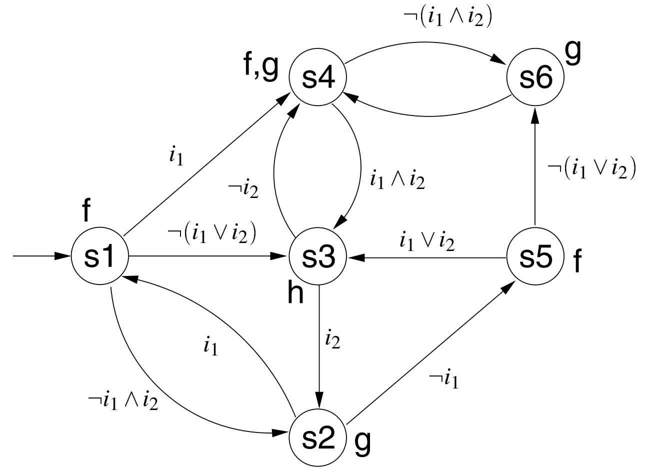

Figure 1. A simple module

Let us consider the following Open-CTL formulas:

$$\varphi_1 = E(f\ U_I\ g)$$
$$\varphi_2 = A(f\ U_I\ g)$$

where $I = i_1 \wedge \neg i_2$. Since g is true in the states $\{s_2, s_4, s_6\}$, both φ_1 and φ_2 are true in these states. At s_1, the input constraint I is satisfied only by the transition from s_1 to s_4, and therefore both φ_1 and φ_2 are true at s_1. On the other hand, at state s_5, the input

constraint I is satisfied only by the transition from s_5 to s_3, and hence both φ_1 and φ_2 are false at s_5.

Now, suppose in φ_1 and φ_2, we have $I = \neg i_1$. Under this constraint, the semantics of φ_1 seeks to determine whether by providing appropriate values to the other input i_2, the system can trace a path satisfying $(f\ U\ g)$. The semantics of φ_2 seeks to determine whether irrespective of the other input i_2, the system can trace a path satisfying $(f\ U\ g)$. At s_1, I is satisfied by the transitions to s_3 and s_2. Therefore, φ_1 is true at s_1, while φ_2 is false.

At s_5, I is satisfied by the transition to s_6 when we set $i_2 = 0$. More interestingly, it should be noted that the transition from s_5 to s_6 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 $I = \neg i_1$, the transition from s_5 to s_3 is enabled. Therefore, φ_1 is true at s_5 (by virtue of the path to s_6), but φ_2 is false (due to the path to s_3).

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

> *Does there exist a path where by giving input $\neg i_1$ we reach a state satisfying h from which there exists a path where by giving input i_2 we reach a state satisfying g?*

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

$$\varphi = E(true\ U_{\neg i_1}\ h \wedge E(true\ U_{i_2}\ g))$$

The subformula $h \wedge E(true\ U_{i_2}\ g)$ is true at state s_3. Therefore φ is true at s_1 (due to the path s_1, s_4, s_3). □

Given the boolean formula, b, representing the set of input vectors enabling a transition from s_i to s_j, determining whether the transition is enabled under input constraint I amounts to finding out whether $b \wedge I$ is satisfiable. In general this problem is NP-complete, and therefore Open-CTL verification in succinctly specified modules is also NP-complete.

Succinct representations of boolean functions are efficiently handled by *Binary Decision Diagrams* (BDDs). The BDD of a function actually represents the set of vectors satisfying the function, hence satisfiability is trivial once the BDD is constructed. Since BDDs can succinctly represent very large functions, BDD based symbolic CTL model checking algorithms have been preferred over explicit graph CTL model checking though the latter works in time polynomial in the size of the system times the length of the formula.

In the next section, we present a symbolic BDD-based verification algorithm for Open-CTL verification of succinctly specified modules which has the same complexity as symbolic model checking algorithms [2, 4] for CTL verification. Thereby we show that Open-CTL verification allows us to verify open systems in the same complexity as closed systems which is significant since CTL model checking in open systems is EXPTIME-complete.

5 Symbolic Open-CTL verification

We have developed a BBD-based symbolic module checking algorithm for Open-CTL verification. The algorithm works in the same style as the symbolic model checking algorithm for CTL verification, but our algorithm matches the input variables with the input constraint during fixpoint computation.

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. For better space utilization, $N(V,I,V')$ is stored as a partitioned transition relation.

The verification procedure *Check* takes the Open-CTL 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_I f, AX_I f, E[f U_I g] and A[f U_I g] as follows:

Check(EX_I f) = CheckEX(I, Check(f))
Check(AX_I f) = CheckAX(I, Check(f))
Check(E[f U_I g]) = CheckEU(I, Check(f), Check(g))
Check(A[f U_I g]) = CheckAU(I, Check(f), Check(g))

Given the BDD $f(V')$ returned by Check(f), the BDD $N(V,I,V')$ for the transition relation, and the BDD $C(I)$ for the input constraint I, we have:

CheckEX(I, Check(f)) =
$\exists V' \exists I [f(V') \wedge N(V,I,V') \wedge C(I)]$
CheckAX(I, Check(f)) =
$\forall V' \exists I [f(V') \wedge N(V,I,V') \wedge C(I)]$
CheckEU(I, Check(f), Check(g)) =
$\mathbf{lfp}\ Z(V') [g(V') \vee (f(V') \wedge CheckEX(I, Z(V')))]$
CheckAU(I, Check(f), Check(g)) =
$\mathbf{lfp}\ Z(V') [g(V') \vee (f(V') \wedge CheckAX(I, Z(V')))]$

The CheckEU and CheckAU procedures for computing the fixpoints are similar to CTL model checking procedures in the literature [2, 4]. The only difference is that we choose only the transitions enabled under the input constraint I.

6 Verifying the PCI-Bus Protocol

The PCI Local Bus, being a well-known and extensively used industry standard can be considered as a suitable example to demonstrate the applicability of the proposed verification approach to real problems encountered by system designers.

A PCI-compliant system consists of master, target and arbiter devices, each of which interact together according to a set of well-defined rules that constitute the protocol. A detailed description of the different signal pins and their semantics can be found in the PCI Special Interest Group documentation or in books such as [9] and [1].

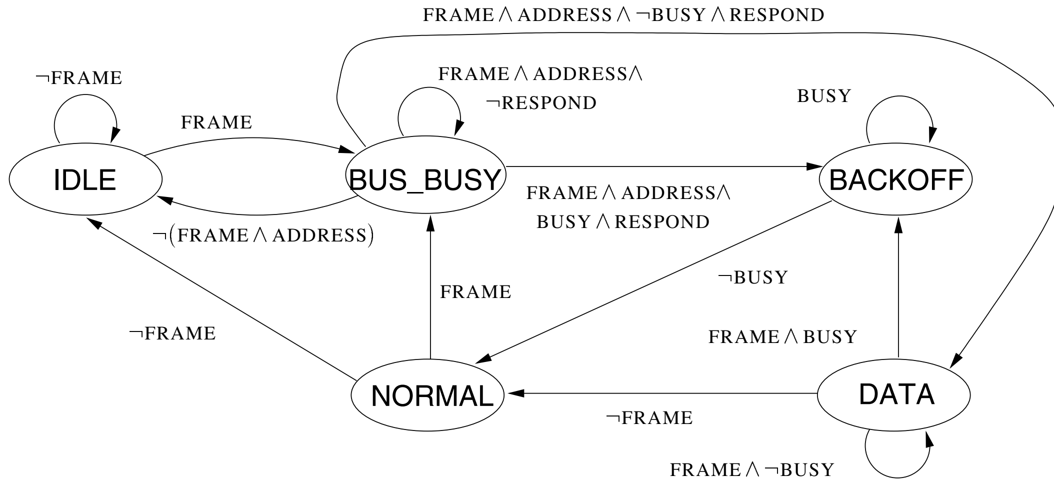

Figure 2. State transition diagram for a PCI slave

Figure 2 shows the state transition system for a PCI target (slave) device. For simplicity, we have removed the states corresponding to the data path configurations. They have been represented as part of the environment of the control module or as signals (like BUSY) derived as a logical function of such data path components.

Let us consider a PCI subsystem with a single target (slave) device. Our knowledge of this peculiarity of the system tells us that the environment signal ADDRESS for the target module can never be deasserted in the BUS_BUSY state. And as we know from the isolated descriptions of all PCI-compliant master devices, the signal FRAME can never be deasserted when the target device has just entered BUS_BUSY from IDLE. The CTL query

$$AG(\text{IDLE} \Rightarrow (AX(\text{BUS_BUSY} \Rightarrow AX(\text{BACKOFF} \vee \text{DATA} \vee \text{BUS_BUSY}))))$$

evaluates to false on the system in Figure 2 because of the *false path* in which a target machine enters state IDLE immediately after entering BUS_BUSY from IDLE. This path is considered by the verifier when evaluating the formula as $false$, not knowing the fact that such a path is not possible in any execution of this target machine in reality.

In Open-CTL the query can be enhanced to incorporate the available information. Thus the following Open-CTL query which actually expresses the designer's intent is true:

$$AG(\text{IDLE} \Rightarrow (AX(\text{BUS_BUSY} \Rightarrow AX_I(\text{DATA} \vee \text{BACKOFF} \vee \text{BUS_BUSY}))))$$

where $I = \text{FRAME} \wedge \text{ADDRESS}$.
Let us consider another example. The CTL formula:

$$AG(\text{IDLE} \Rightarrow AX(\text{BUS_BUSY} \Rightarrow AX(\text{BUS_BUSY} \Rightarrow AX(\text{BUS_BUSY} \Rightarrow AX(\text{IDLE})))))$$

comes out to be false as the verifier does not know that the PCI master would not continue to assert FRAME for more than 3 clock cycles if the target does not respond by asserting DEVSEL. Incoprporating this information we get the following Open-CTL query which evaluates to true as expected.

$$AG(\text{IDLE} \Rightarrow AX(\text{BUS_BUSY} \Rightarrow AX(\text{BUS_BUSY} \Rightarrow AX(\text{BUS_BUSY} \Rightarrow AX_I(\text{IDLE})))))$$

where $I = \neg\text{FRAME}$.

7 Results

We present experimental results of the Open-CTL verifier on modules from the Texas-97 Verification Benchmarks [12]. These benchmarks consist of industrial grade circuit modules specified in Verilog. The verifier was run on a 300 MHz SUN Enterprise 250 workstation with 128 MB RAM. We used a commercially available design analyzer to parse the Verilog code, and the CUDD BDD-package [11] for generating the BDDs.

Our verifier accepts modules written in a subset of synthesizable Verilog. The transition relation extracted from a module is stored in a BDD. In order to reduce the effective state space of the modules, we developed an abstraction algorithm, which removes portions of the circuit that do not affect the state bits concerning the properties to be verified. We observed that this simple algorithm removed major parts of the data path while retaining the control path. We decomposed some of the CTL properties given in the Texas-97 Verification Benchmarks into Open-CTL formulas on the individual modules.

Table 1 shows the results of abstraction. The second column specifies the module within the circuit specified in the

Circuit	Module	Pre-abstraction #States	Pre-abstraction BDD nodes	Post-abstraction #States	Post-abstraction BDD nodes	Our ref.
MSI Cache Coherence Protocol	3-proc sys arbiter	256	409	8	295	CCP-1
	2-proc sys arbiter	512	307	16	193	CCP-2
MPEG System Decoder	timestamp	10^{24}	60145	10^{4}	1287	MPEG-1
	prefixcode	32	140	32	140	MPEG-2
	parsepack	10^{21}	6662563	128	1133	MPEG-3
	packstart	32	398	32	398	MPEG-4
PI Bus (multimaster)	slave	10^{41}	-	32	4004	PI
PCI Local Bus	master	10^{62}	-	10^{12}	16857	PCI-M
	target	10^{23}	462804	10^{13}	24920	PCI-S

Table 1. Results of abstraction on Texas-97 Benchmarks

Circuit (our ref.)	Extraction time	#Queries	Verification time
CCP-1	40 ms	3	150 ms
CCP-2	80 ms	2	80 ms
MPEG-1	210 ms	3	1.5 sec
MPEG-2	30 ms	3	100 ms
MPEG-3	100 ms	4	610 ms
MPEG-4	80 ms	3	160 ms
PI	2 sec	2	540 ms
PCI-M	38.5 sec	4	6 sec
PCI-S	19.5 sec	4	5 sec

Table 2. Results of Open-CTL Verifier

first column. The last column in Table 1 shows the names that we use to refer to these modules. We show the approximate number of states and the number of BDD nodes before and after abstraction. In some cases, without abstraction CUDD failed to create the BDDs within our memory of 128 MB.

Table 2 shows the time required by our verifier. The extraction 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 verification time denotes the time required after extraction by the Open-CTL verifier to check the number of properties given in the third column. All times refer to CPU time.

As shown in Table 2, most of the modules were verified in less than a second. In the case of the PCI master and PCI slave modules, the verifier required nearly half a minute. The results clearly indicate the advantage of verifying modules one at a time, and Open-CTL appears to be very well suited for this style of verification.

References

[1] Anderson, D., and Shanley, T., *PCI System Architecture*, Addison-Wesley Longman, 1995.

[2] 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.

[3] 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 Systems*, 8(2):244-263, 1986.

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

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

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

[7] Kupferman, O. and Vardi, M.Y., Module Checking. In *Proc. 8th Int. Conf. on CAV*, LNCS 1102, 75-86, 1996.

[8] Kupferman, O. and Vardi, M.Y., Module Checking revisited. In *Proc. 9th Int. Conf. on CAV*, LNCS 1254, 36-47, 1997.

[9] Solari, E., and Willse, G., *PCI Hardware and Software Architecture and Design*, 4th Edition, Annabooks, 1998.

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

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

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

[13] VIS: A system for verification and synthesis, The VIS Group, In *Proc. of the 8th Int. Conf. on CAV*, LNCS 1102, 428-432, 1996.