Temporal Formula Specifications of Asynchronous Control Module in Model Checking

CHIKATOSHI YAMADA
Takushoku Univ. Hokkaido Jr. College
4558 Memu, Fukagawa, Hokkaido 074-8585
JAPAN

YASUNORI NAGATA
Univ. of the Ryukyus
Dept. of Electrical and Electronics Eng.
1 Senbaru, Nishihara, Okinawa 903-0213
JAPAN

Abstract: System verification plays an important role in large scale and complex systems. However, it is very difficult for designers other than the specialist who is well versed in Temporal Logic to specify behaviors of the system. This article considers the case where designers of systems can specify temporal formulas easily in system verification. We propose a method by which temporal formulas can be obtained inductively for specifications in system verification. System designers can easily derive complex temporal formulas by using the specification method.

Key–Words: Formal verification, Model checking, Temporal formula specifications, Asynchronous control module

1 Introduction

Today, hardware and software systems are widely used in applied field where no failure is permitted: electronic commerce, telephone switched network, and medical equipment, etc. Industrial designs are becoming more and more complex as technology advances and demand for higher performance increases. The validity of a design accompanies checking whether the physical design satisfies its specification. In traditional design flow, validation is accomplished through simulation and testing. Some errors inside a design may exhibit nondeterministic behaviors, and therefore, will not be reliably repeatable. This makes testing and debugging using simulation difficult. Also, exhaustive testing for nontrivial designs is generally infeasible, therefore, testing provides at best only a probabilistic assurance. Formal verification, in contrast to testing, uses rigorous mathematical reasoning to show that a design meets all or parts of its specification. For that reason, formal verification is on the critical path for today's IC designers, no matter what type of systems they are building[1].

Formal verification has problems of its own class too. The major problem with automatic formal verification is that a large amount of memory and time is often required, because the underlying algorithm in these methods usually involves systematic examination of all reachable states of the system to be verified. As the number of reachable states increases rapidly with the size of the system, the basic algorithm by itself becomes impractical: the number of states for the system is often too large to check exhaustively within the limited time and memory that is available. This phenomenon is known as the state space explosion problem[2, 3]. In design of complex and large scale systems, system verification has played an important role. System verification ascertains whether designed systems can be executed or specified. Various formal methods for verification have been studied[1, 2, 4]. However, specification methods for the verification have not been studied so far very much.

This article is to focus on specification process of model checking in system verification shown in **Fig.1**, and to propose a new method which can obtain temporal formulas inductively from modeling systems. System designers can easily derive complex temporal formulas by using the method.

2 Model Checking

The model checking[1, 2, 4] is easy to describe. Given a Kripke structure $M = (S , R , L)$ that represents a finite-state concurrent system and a temporal logic formula f expressing some desired specification, find the set of all states in S that satisfy f:

$$\{ s \in S \mid M , s \models f \}.$$

An important issue in specifications completeness. Model checking provides means for checking that a model of the design satisfies a given specification, but it is impossible to determine whether the

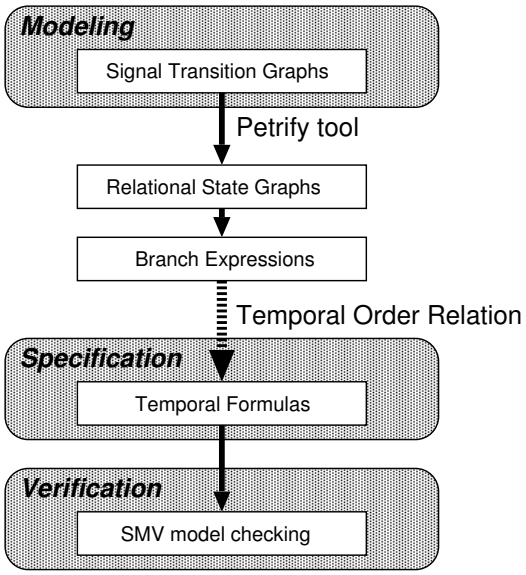

Figure 1: The framework of proposed method.

given specification covers all the properties that the system should satisfy.

- *Reachability property* states that some particular situation *can be reached*.
- *Safety property* expresses that, under certain conditions, nothing bad *will happen*.
- *Liveness property* express that, under certain conditions, something good *will eventually happen*.
- *Fairness property* express that, under certain conditions, something will (or will not) occur infinitely often.

In this article, behaviors of a system are specified by temporal formulas in Computation Tree Logic.

2.1 Signal Transition Graph

In order to describe highly concurrent systems, graph-based specification methods[5] have been widely used. An Signal Transition Graph (STG)[6], a labeled interpreted Petri Net[7], has been considered as a well-suited specification method to describe asynchronous circuits.

Definition 1 (Petri Net (PN)). A *Petri Net* is a bipartite directed graph consisting of 4-tuple $\sum = (P, T, F, m_0)$, where

1. P is a finite set of places.
2. T is a finite set of transitions, satisfying $P \cap T = \phi$ and $P \cup T = \phi$.
3. F is a flow relation $F \subseteq (P \times T) \cup (T \times P)$, specifies binary relation between transitions and places.
4. m_0 is the initial marking of the PN.

When transitions are interpreted as rising and falling transitions of signals of a control circuit, an STG is one interpretation of a PN.

Definition 2 (Signal Transition Graph (STG)). Let J be a set of signals of a network, A *Signal Transition Graph* defined on J is a Petri Net $\sum_J = \langle P, T, F, M_0 \rangle$ with $T : J \rightarrow \{ + , - \}$.

Each transition of the STG is interpreted as a rising transition or a falling transition of a signal.

Consider an arbiter module shown in **Fig.2**. An STG for the arbiter module is shown in **Fig.3**, where '+' mean a rising edge and '-' means a falling edge of a certain signal, respectively. This example uses two signals **u0** and **u1**. Black circle on a transition edge indicates a token. A transition is enabled when all input places have at least one token. When an enabled transition fires, it removes one token from each input place and adds one token to each output place.

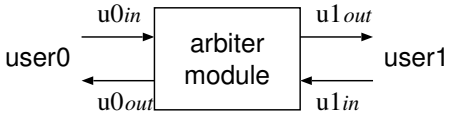

Figure 2: An arbiter handshake module.

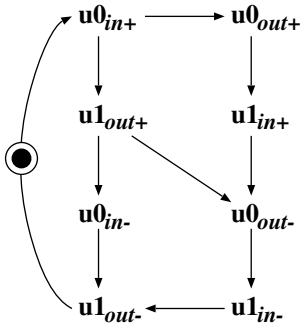

Figure 3: A signal transition graph for **Fig.2**

2.2 Temporal Logic

Temporal logic[1, 2, 4, 8] is a formalism for describing sequences of transitions between states in a reac-

tive system. In the temporal logics that we will consider, time is not mentioned explicitly; instead, a formula might specify that *eventually* some designated state is reached, or that an error state is *never* entered. Properties like *eventually* or *never* are specified using special *temporal operators*. These operators can also be combined with boolean connectives or nested arbitrarily. Temporal logics differ in the operators that they provide and the semantics of those operators. Its operators mimic linguistic constructions (the adverbs "always" , "until" , the tenses of verbs, etc.) with the result that natural language statements and their temporal logic formalization are fairly close. Finally, temporal logic comes with a formal semantics, an indispensable specification language tool.

2.2.1 Computation Tree Logic(CTL)

Here we give descriptions of CTL[1, 2, 4, 8]. CTL is a sort of temporal logic, which has the following formulas:

- **G**q : means that q always holds for all successor states on a certain path.
- **F**q : represents that q must be sometimes true for only one successor state of the path, and is similar to the formula which expresses future in linear temporal logic.
- p**U**q : is that p must be true on the path states, beginning at the current state, until q becomes true.
- **A**q : is that q is true on all possible paths, from the current state.
- **E**q : is that q is true at least on one path.

The correctness of properties to be verified is usually specified in CTL. CTL, branching-time temporal logic, is extending propositional logic with temporal operators that express how propositions change their truth values over time. Here we use temporal operators: Operators **G**, **F**, and **X** meaning *globally*, *sometime in the future*, and *next time*, respectively. In CTL, these operators must be preceded by a path quantifier which is either **A** (for all computation paths) or **E** (for some computation path). We consider operators **AG**, **AF**, and **AX**: The formula **AG** p holds in state s if p holds in all states along all computation paths starting from s, while the formula **AF** p holds in state s if p holds in some state along all computation paths starting from s. The formula **AX** p holds in state s if p holds in all the states that can be reached from s in exactly one step.

Many of the methods to avoid the state explosion problem rely on compositional reasoning or abstraction. The logic that is typically used in these cases is more restricted and allows only universal path quantifiers.

3 Proposed Method

3.1 Strong/Weak Temporal Order Relation

In verifying behaviors of a system, checking all signal events is inefficient. Reducing signal events to be checked is necessary for specifying behaviors of the system. Here, We consider a system which has 3-inputs (a , b , c) and 2-outputs (x , y). Suppose that behaviors of the system occur as $a \rightarrow x \rightarrow b \rightarrow c \rightarrow y \rightarrow a$, repeatedly. All relations of the signal events can be indicated as follows:

$$\{(a\ ,\ x)\ ,\ (a\ ,\ y)\ ,\ (x\ ,\ b)\ ,\ (b\ ,\ c)\ ,\ (b\ ,\ y)\ ,\ (c\ ,\ y)\},$$

where (a , x) indicates that output x occur after input a . Although output y is not an immediate successor of input a , (a, y) can be considered because output y must occur after input a in the future. Definitions of *strong/weak temporal order relations* are as follows:

Definition 3 (strong temporal order relation). *A strong temporal order relation is any inverse input-output relation of event sequences.*

Here, we focus on relation (x , b). We notice that (x , b) indicates an inverse relation of input and output events. However, it is not necessary that input b must occur after output y in many cases excepting systems of 1-input and 1-output. Thus such an inverse input-output relation can be reduced by a *strong temporal order relation*.

Definition 4 (weak temporal order relation). *A weak temporal order relation is any relation of input signal events.*

Further, we focus on relation (b , c). We notice that the relation only indicates inputs. Output y is a successor of inputs b and c by relations (b , y) and (c , y). On the other hand, output y can occur by rendezvous of inputs b and c. Output y can occur independently of relation (b , c). Therefore, such a relation can be reduced by a *weak temporal order relation*.

Thus, behaviors of the system can be specified by introducing strong/weak temporal order relations as follows:

$\{ (a , x) , (a , y) , (b , y) , (c , y) \}$

Its specification shows that output x can occur after input a and output y can occur by rendezvous inputs a, b, and c.

3.2 Procedure of Specification

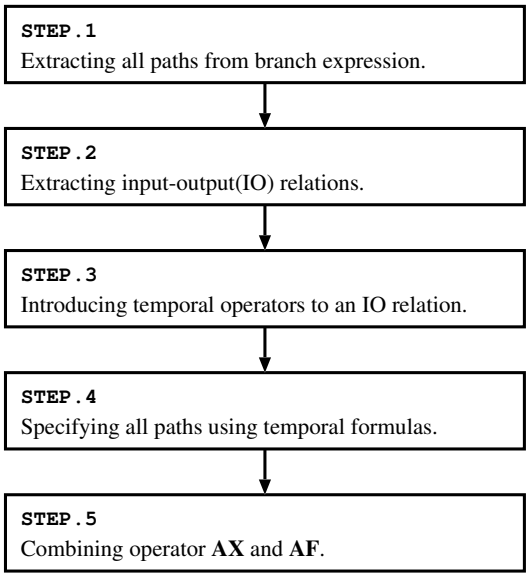

Figure 4: Procedure of Specification.

In this section, we describe the procedure of the proposed specification method shown in **Fig.4**. This procedure corresponds to the part in the wavy arrow line in **Fig.1**. The procedure is composed of five steps shown in **Fig.4**. Here, we explain the procedure as follows:

[STEP.1]
In this step, event sequences are extracted from branch expression.

[STEP.2]
In this step, checked signal events can be reduced by introducing *strong/weak temporal order relations*.

[STEP.3]
In each path, if IO relation shows that there is immediate successor, specified as **AX** operator, otherwise specified as **AF** operator.

[STEP.4]
In all paths, relations of the same temporal operator and the same IO can be extracted. Otherwise only the same IO relation can be extracted. Since **AF** expresses "*sometime in the future for all paths*," the *next* operator **AX** can be covered as $\mathbf{AX} \subseteq \mathbf{AF}$. Thus, the extracted same IO relation can be gathered by **AF**.

[STEP.5]
In all paths, relations of the same output can be combined.

Temporal formulas can be derived by repeating the above-mentioned steps.

4 Specification Examples

We show specifications for the DME cells as is shown in **Fig.5**. Temporal formulas are specified without our proposed method as follows:

[Specification without the proposed method]

[DME1]
$\mathbf{AG}\ [\ \mathbf{AF}\ (u1.req_+ \wedge d1.ack_+ \wedge d4.req_-\ , u1.ack_-)$
$\vee\ \mathbf{AF}\ (u1.req_+ \wedge d4.req_-\ , d1.req_+)$
$\vee\ \mathbf{AX}\ (u1.req_+\ , d4.ack_-)$
$\vee\ \mathbf{AF}\ (u1.req_- \wedge d1.ack_- \wedge d4.req_+\ , u1.ack_+)$
$\vee\ \mathbf{AF}\ (u1.req_- \wedge d4.req_+\ , d1.req_-)$
$\vee\ \mathbf{AX}\ (u1.req_-\ , d4.ack_+)$

[DME2]
$\mathbf{AG}\ [\ \mathbf{AF}\ (u2.req_+ \wedge d2.ack_+ \wedge d1.req_-\ , u2.ack_-)$
$\vee\ \mathbf{AF}\ (u2.req_+ \wedge d1.req_-\ , d2.req_+)$
$\vee\ \mathbf{AX}\ (u2.req_+\ , d1.ack_-)$
$\vee\ \mathbf{AF}\ (u2.req_- \wedge d2.ack_- \wedge d1.req_+\ , u2.ack_+)$
$\vee\ \mathbf{AF}\ (u2.req_- \wedge d1.req_+\ , d2.req_-)$
$\vee\ \mathbf{AX}\ (u2.req_-\ , d1.ack_+)$

[DME3]
$\mathbf{AG}\ [\ \mathbf{AF}\ (u3.req_+ \wedge d3.ack_+ \wedge d2.req_-\ , u3.ack_-)$
$\vee\ \mathbf{AF}\ (u3.req_+ \wedge d2.req_-\ , d3.req_+)$
$\vee\ \mathbf{AX}\ (u3.req_+\ , d2.ack_-)$
$\vee\ \mathbf{AF}\ (u3.req_- \wedge d3.ack_- \wedge d1.req_+\ , u3.ack_+)$
$\vee\ \mathbf{AF}\ (u3.req_- \wedge d2.req_+\ , d3.req_-)$
$\vee\ \mathbf{AX}\ (u3.req_-\ , d2.ack_+)$

[DME4]
$\mathbf{AG}\ [\ \mathbf{AF}\ (u4.req_+ \wedge d4.ack_+ \wedge d3.req_-\ , u4.ack_-)$
$\vee\ \mathbf{AF}\ (u4.req_+ \wedge d3.req_-\ , d4.req_+)$
$\vee\ \mathbf{AX}\ (u4.req_+\ , d3.ack_-)$
$\vee\ \mathbf{AF}\ (u4.req_- \wedge d4.ack_- \wedge d3.req_+\ , u4.ack_+)$
$\vee\ \mathbf{AF}\ (u4.req_- \wedge d3.req_+\ , d4.req_-)$
$\vee\ \mathbf{AX}\ (u4.req_-\ , d3.ack_+)$

Moreover, we indicate temporal formulas with our proposed method as follows:

[Specification with the proposed method]

[DME1]
AG [**AF** ($u1.req_+ \wedge d1.ack_+ \wedge d4.req_-$, $u1.ack_+$)
$\vee$ **AX** ($u1.req_+$, $d1.req_+$) $\vee$ **AX** ($u1.req_+$, $d4.ack_-$)
$\vee$ **AF** ($d1.ack_- \wedge u1.req_- \wedge d4.req_+$, $d4.ack_+$)]

[DME2]
AG [**AF** ($u2.req_+ \wedge d2.ack_+ \wedge d1.req_-$, $u2.ack_+$)
$\vee$ **AX** ($u2.req_+$, $d2.req_+$) $\vee$ **AX** ($u2.req_+$, $d1.ack_-$)
$\vee$ **AF** ($d2.ack_- \wedge u2.req_- \wedge d1.req_+$, $d1.ack_+$)]

[DME3]
AG [**AF** ($u3.req_+ \wedge d3.ack_+ \wedge d2.req_-$, $u3.ack_+$)
$\vee$ **AX** ($u3.req_+$, $d3.req_+$) $\vee$ **AX** ($u3.req_+$, $d2.ack_-$)
$\vee$ **AF** ($d3.ack_- \wedge u3.req_- \wedge d2.req_+$, $d2.ack_+$)]

[DME4]
AG [**AF** ($u4.req_+ \wedge d4.ack_+ \wedge d3.req_-$, $u4.ack_+$)
$\vee$ **AX** ($u4.req_+$, $d4.req_+$) $\vee$ **AX** ($u4.req_+$, $d3.ack_-$)
$\vee$ **AF** ($d4.ack_- \wedge u4.req_- \wedge d3.req_+$, $d3.ack_+$)]

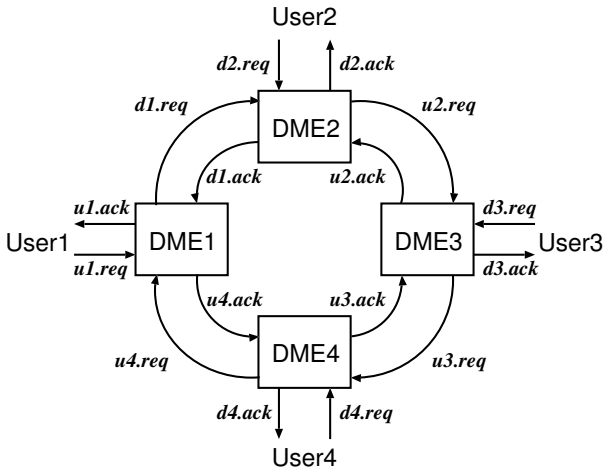

Figure 5: Chained DME modules[3].

5 Verification Results

We show some asynchronous bench marks in the table. All these circuit verifications are performed on an 3.2GHz Pentium Xeon processor under Linux with 1GB of available RAM. In this article, all circuits are verified by Cadence SMV[10].

For each circuit, we report the number of boolean variables necessary to represent the corresponding model, OBDD nodes, and time required by the systems to analyze the model. Some circuits in the table can be found in the distribution of SMV[11, 12]. For small circuits such as C-element4, p-queue and pipeline4, time is not much different between the two methods. On the other hand, as the circuits become larger, the effect begins to appear in the results: It is remarkable especially for control modules. Moreover, performance results of distributed mutual exclusion(DME)[4] circuit are shown in **Fig.6**, where the number of cells refer to the number of DME modules shown in **Fig.5**. Executions of verification without the method resulted in "disable" at the DME of 8-cells.

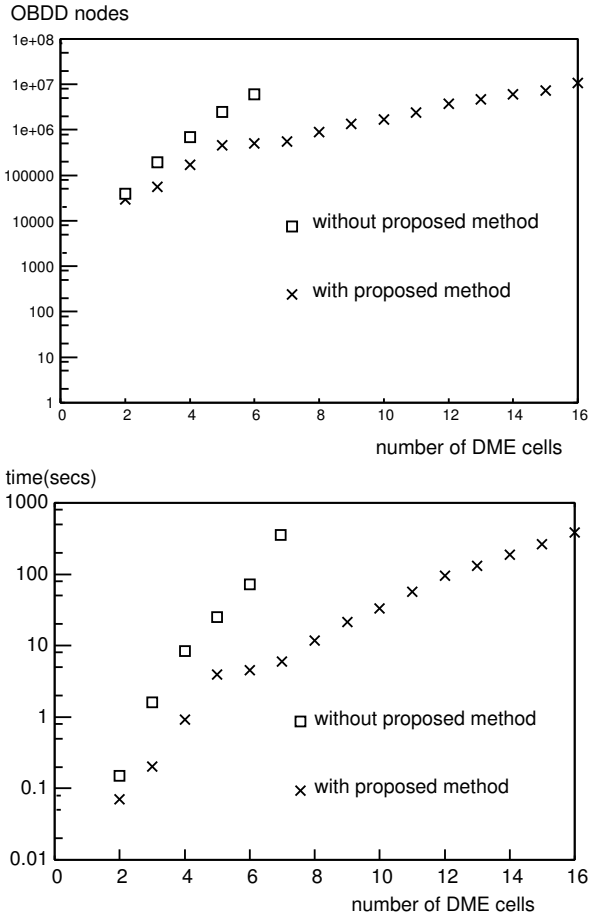

Figure 6: Verification performance of the DMEs: OBDD nodes(upper), execution time(lower).

6 Conclusion

We proposed a method by which temporal formulas can be obtained inductively for specifications in model checking. Users must generally know well temporal specification because the specification might be complex. Our proposed method can gain temporal formula specifications inductively. We aimed at input-output order relations for systems, not considering output-input order relations. Furthermore, we de-

Table 1: Verification results

	with proposed method				without proposed method	
Circuit name	OBDD nodes	Reduce(%)	Time(secs)	Reduce(%)	OBDD nodes	Time(secs)
C-element4	1988	-1.5%	0.06	0.0%	2018	0.06
C-element16	242244	-0.1%	0.97	-1.0%	242462	0.98
p-queue	139530	-5.8%	0.82	-3.5%	148160	0.85
pipeline2	679	-23.5%	0.02	-60.0%	888	0.05
pipeline4	3272	-22.9%	0.06	-66.8%	4244	0.09
pipeline8	144431	-13.8%	0.79	-66.8%	167469	2.38
abp4	75661	-21.5%	0.43	-21.8%	96384	0.55
pci3p	447889	-21.5%	1.19	-66.5%	570388	3.55
pci	193576	-44.2%	385.57	-34.6%	346758	589.75

fined strong/weak temporal order relations in the procedure of specification. Weak temporal order relations include orders of inputs implicitly. Strong temporal order relations express inverse input-output order relations. We showed that the verification tasks are reduced for OBDD nodes, and execution time with our proposed inductive specification method. System designers can easily lead complex temporal formulas by using the method.

In this research work, the scale of circuits and systems used for the verification didn't reach at practicable levels. Then, it is assumed to be a research work in the future to embed the specification into commercial CAD tools, and will check structures of complex systems with the relations under an embedded system.

References:

[1] E.M. Clarke, O. Grumberg, and D. A. Peled: *Model Checking*, MIT Press, 2001.

[2] T. Kropf: *Introduction to Formal Hardware Verification*, Springer, 1999.

[3] Randal E. Bryant: "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. On Computers*, Vol.C-35, No.8, 1986.

[4] Kenneth L. McMillan: *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[5] Tam-Anh Chu: "Synthesis of self-timed VLSI circuits from graph-theoric specifications," *In Proc. International Conf. Computer Design (ICCD)*, pp.220-223, IEEE Computer Society Press, 1987.

[6] Sung-Tae Jung and Chris J. Myers: "Direct Synthesis of Timed Circuits From Free-Choice STGs," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, Vol.21, No.3, pp.275-290, March 2002.

[7] Alex Yakovlev, Luis Gomes and Luciano Lavagno: *Hardware Design and Petri Nets*, Kluwer Academic Publishers, 2000.

[8] Dov M. Gabbay, Mark A. Reynolds, and Marcelo Finger: *Temporal Logic Mathematical Foundations and Computational Aspects*, Volume 2, Oxford Science Publications, 2000.

[9] D. L. Dill, S. Nowick, and R. F. Sproull: "Specification and automatic verification of self-timed queues," *In Formal verification of hardware design*, IEEE Computer Society Press, 1990.

[10] Cadence SMV, `http://www.kenmcmil.com/kenmcmil/smv.html/`.

[11] NuMSV, `http://nusmv.irst.itc.it/index.html`.

[12] SMV, `http://www.cs.cmu.edu/~modelcheck/smv.html`.

[13] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A.Yakovlev: "Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *IEICE Transactions on Information and Systems*, Vol.E80-D, No.3, pp.315-325, 1997.