Temporal Formula Specifications Using An Inductive Method

Chikatoshi Yamada [i, ii], Yasunori Nagata [ii], and Zensho Nakao [ii]
[i] 4558 Memu Fukagata-city, Hokkaido, 074-8585, JAPAN
Takushoku University Hokkaido College
email: cyamada@takushoku-hc.ac.jp
[ii] 1 Senbaru, Nishihara, Nakagami, Okinawa, 903-0213, JAPAN
Electrical and Electron Engineering Department, University of the Ryukyus
email: ngt@eee.u-ryukyu.ac.jp, nakao@augusta.eee.u-ryukyu.ac.jp

ABSTRACT

Design verification has played an important role in the design of large scale and complex systems. In this article, we focus on model checking methods. Behaviors of modeled systems are general specified by temporal formulas of computation tree logic, and users must know well about temporal specification because the specification might be complex. We propose a method temporal formulas are obtained inductively, and amounts of memory and time are reduced. We will show verification results using the proposed method.

***Index Terms*—Inductive Specification, Strong/Weak Temporal Order Relation, Computation Tree Logic, System Verification.**

I. INTRODUCTION

Recently design verification has played an important role in the design of large scale and complex systems. System verification ascertains whether designed systems can be executed or specified. Various formal methods for verification have been studied [1][2][3][4][5][6]. In this article, we focus on model checking methods. In model checking methods, a targeted circuit or system is modeled and unimportant or irrelevant to verification details are eliminated. Users can determine the structure of hierarchical levels (such as architecture, register-transfer or gate) in the modeling process. A circuit (or system) is modeled using a signal transition graph (STG) because we particularly aim to asynchronous handshake circuits in this research. Then behaviors of modeled systems are generally specified by temporal formulas of computation tree logic (CTL). Finally, checked circuits are verified whether the circuits can satisfy descriptions of specification. In specifying temporal formulas, however, users must know well about temporal specification because the specification might be complex. In this article, we extend our proposed method to specification method which can obtain temporal formulas inductively from modeling systems. Moreover, we also show that verification tasks can be executed efficiently by using the proposed method.

The rest of this article is organized as follows: In section II, Signal Transition Graph, Computation Tree Logic, Symbolic Model Checking and NuSMV are briefly explained, and in section III our proposed method is described by means of procedures of specification. Moreover, we demonstrate specification examples using the proposed method, and in section IV some asynchronous circuit bench marks are used for verification to compare by NuSMV tool. Finally, we summarize the discussion in section VI.

II. PRELIMINARIES

A. Signal Transition Graph

Signal Transition Graph (STG)[7][8][9][10] is a well-known method for specifying asynchronous sequential circuits. STG is a sub-class of Petri nets, which are free choice nets. The structure for STG is a triple $M = < S , R , S_0 >$, where S is a set of signal transitions in reachable states, R is a set of transition relations on S , and S_0 is a set of initial transitions, where S_0 is a subset of S. An signal transition graph is shown in fig.2.

B. Computation Tree Logic

The correctness property to be verified is specified in CTL. CTL is branching-time temporal logic, 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 start-

ing 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. CTL formulas and semantics are defined as follows.

1) CTL formulas

a) *Every atomic proposition is a CTL formula.*

b) *. If φ and ψ are CTL formulae, then so are $\neg\varphi$, $(\varphi \wedge \psi)$, $\mathbf{AX}\varphi$, $\mathbf{EX}\varphi$, $\mathbf{A}(\varphi \mathbf{U} \psi)$, $\mathbf{E}(\varphi \mathbf{U} \psi)$.*

The remaining operators are viewed as being derived from *1)* as following rules:

$$\varphi \vee \psi = \neg(\neg\varphi \wedge \neg\psi)$$
$$\mathbf{AF}\psi = \mathbf{A}(\text{true}\ \mathbf{U}\ \psi)$$
$$\mathbf{EF}\psi = \mathbf{E}(\text{true}\ \mathbf{U}\ \psi)$$
$$\mathbf{AG}\varphi = \neg\mathbf{E}(\text{true}\ \mathbf{U} \neg\varphi)$$
$$\mathbf{EG}\varphi = \neg\mathbf{A}(\text{true}\ \mathbf{U}\ \neg\varphi)$$

2) Semantics of CTL formulas

a) *$s \models \varphi \Leftrightarrow s \in L(p)$, where p is an atomic proposition.*

b) *$s \models \neg\varphi \Leftrightarrow s \not\models \varphi$*

c) *$s_0 \models \mathbf{AX}\varphi \Leftrightarrow$ For all paths $(s_0, s_1, \ldots)$, $s_0 \models \varphi$*

d) *$s_0 \models \mathbf{EX}\varphi \Leftrightarrow$ For a path $(s_0, s_1, \ldots)$, $s_1 \models \varphi$*

e) *$s_0 \models \mathbf{AG}\varphi \Leftrightarrow$ For all paths $(s_0, s_1, \ldots)$, for all i, $s_i \models \varphi$*

f) *$s_0 \models \mathbf{EG}\varphi \Leftrightarrow$ For some path $(s_0, s_1, \ldots)$, for all i, $s_i \models \varphi$*

g) *$s_0 \models \mathbf{A}(\varphi \mathbf{U} \psi) \Leftrightarrow$ For all paths $(s_0, s_1, \ldots)$, for some i, $s_i \models \psi$ and for all $j<i$, $s_j \models \varphi$*

h) *$s_0 \models \mathbf{E}(\varphi \mathbf{U} \psi) \Leftrightarrow$ For some path $(s_0, s_1, \ldots)$, for some i, $s_i \models \psi$ and for all $j<i$, $s_j \models \varphi$*

C. Symbolic Model Checking

Model Checking is a process of exploring a finite state space to determine whether or not a given property holds. The major problem of model checking is that generally making exhaustive exploration is feasible because the state space arising from practical problems are often extremely large.

A promising approach to this problem is the use of symbolic representations of the state space. In CTL symbolic model checking, Boolean functions represented by Ordered Binary Decision Diagrams (OBDDs) are used to represent the state space instead of explicit adjacency-lists. This can reduce the memory and required time dramatically, because OBDDs represent occurring Boolean functions very compactly in almost cases.

D. NuSMV

We execute verification using NuSMV[11] in this research. NuSMV is designed to be a well structured, open, flexible and documented platform for model checking. NuSMV is the result of the reengineering, reimplementation and extension of SMV[12]. NuSMV can process files written in the SMV language, and allows for the construction of the model with different modalities, reachability analysis, fair CTL model checking, computation of quantitative characteristics of the model, and generation of counterexamples. In addition, NuSMV features an enhanced partitioning method for synchronous models, and allows for disjunctive partitioning of asynchronous models, and for the verification of invariant properties in combination with reachability analysis. Furthermore, NuSMV supports Liner Temporal Logic (LTL) model checking. The algorithm is based on the combination of a tableau constructor for the LTL formula with standard CTL model checking.

III. PROPOSED METHOD

Behaviors of modeled systems are generally specified by temporal formulas of CTL. Finally checked circuits are verified whether the circuits can satisfy descriptions of specification or not. In specifying temporal formulas ,however, users must know well temporal specification because the specification might be complex. Here we show that temporal specifications can be obtained inductively. We especially take into account an asynchronous handshake circuit shown in fig.1. The procedure of specification is indicated as follows:

A. Procedure of specification

[1] Extracting all paths (fig.4).

(A) *a+ b+ d+ c+ a- b- d- c-*
(B) *a+ b+ d+ c+ b- a- d- c-*
(C) *a+ b+ d+ c+ b- d- a- c-*

[2] For each path, extracting input-output (IO) relations. The extraction can be repeated until detection of reverse value of a signal (such as from *a+* to *a-*). If the reverse value are detected, then the extraction can start for the next signal. Initial values of all signals are "zero", i.e. "-".values of all signals are "zero", i.e. "-".

(A) {(*a+* , *b+*) , (*a+* , *c+*) , (*d+* , *c+*) , (*d+* , *b-*) , (*a-* , *b-*) , (*a-* , *c-*) , (*d-* , *c-*) , (*d-* , *b+*)}

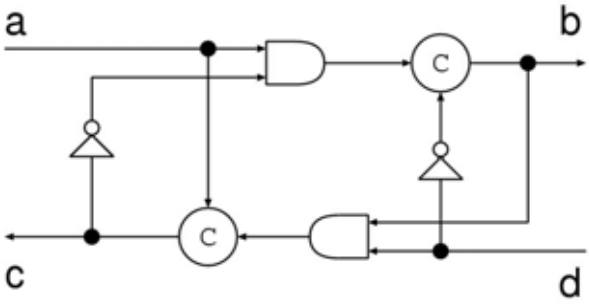

Figure 1: A checked handshake circuit.

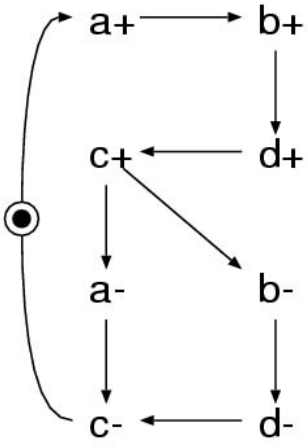

Figure 2: An signal transition graph for the circuit.

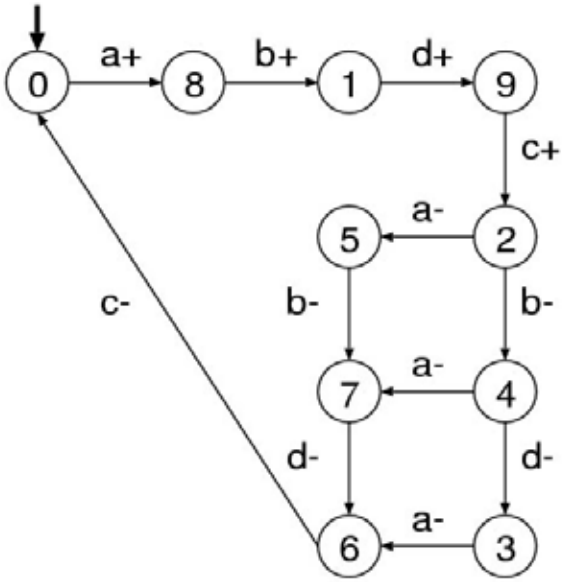

Figure 3: A state graph for the circuit.

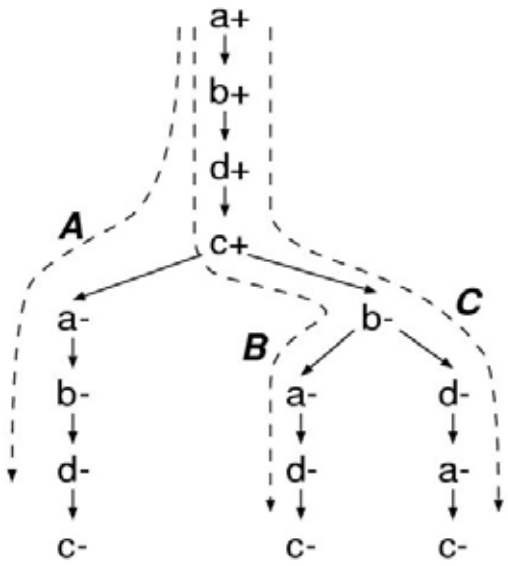

Figure 4: A branch expression for the state graph, where '+' means rising edge and '-' means falling edge respectively.

Here we compare input *a+* with *d+*. Successors (outputs) of *a+* are *b+* and *c+*, and successors of *d+* are *c+* and *b-*, respectively. However, successor b- of d+ expresses the next cycle of path (shown underlined). Thus signal event *a+* occurs earlier than *b+*. Such a relation is called as a *weak temporal order relation*.

(B) {(*a+* , *b+*) , (*a+* , *c+*) , (*d+* , *c+*) , (*d+* , *b-*) , (*a-* , *c-*) , (*d-* , *c-*) , (*d-* , *b+*)}
(C) {(*a+* , *b+*) , (*a+* , *c+*) , (*d+* , *c+*) , (*d+* , *b-*) , (*a-* , *c-*) , (*d-* , *c-*) , (*d-* , *b+*)}

For path **(B)** and **(C)**, we can see that these paths are equivalent. Thus there are equivalent paths in IO relation. Although there is an IO relation (*a-* , *b-*) in path **(A)**, there is not (*a-* , *b-*) in **(B)**. However, signal event *b-* in **(B)** can occur by (*d+* , *b-*). Elimination of (*a-* , *b-*) shows that *b-* occurs earlier than *a-*. Thus, if there are reverse IO relations between paths, such a relation is called as a *strong temporal order relation*.

[3] Let us introduce temporal operators to an IO relation. In path **(A)**, IO relation (*a+* , *b+*) shows that *b+* is an immediate successor of *a+*, specified as **AX**(*a+* , *b+*). Here temporal operator **AX** can be used because there is only a transition *b+* from *a+*. Moreover, IO relation (*a+* , *c+*) shows that *c+* is a successor of *a+*, not immediate, specified as **AF**(*a+* , *c+*). Similarly, temporal operators can be introduced to IO relations as follows:

(A) {**AX**(*a+* , *b+*) , **AF**(*a+* , *c+*) , **AX**(*d+* , *c+*) , **AF**(*d+* , *b-*) , **AX**(*a-* , *b-*) , **AF**(*a-* , *c-*) , **AX**(*d-* , *c-*) , **AF**(*d-* , *b+*)}

(B) {**AX**(*a+* , *b+*) , **AF**(*a+* , *c+*) , **AX**(*d+* , *c+*) , **AF**(*d+* , *b-*) , **AF**(*a-* , *c-*) , **AX**(*d-* , *c-*) , **AF**(*d-* , *b+*)}

(C) {**AX**(*a+* , *b+*) , **AF**(*a+* , *c+*) , **AX**(*d+* , *c+*) , **AF**(*d+* , *b-*) , **AF**(*a-* , *c-*) , **AX**(*d-* , *c-*) , **AF**(*d-* , *b+*)}

Although path **(B)** and **(C)** were distinguished in procedure 2, it can be done by introducing temporal operators.

[4] Specifying all paths using temporal formulas. In all paths, transitions which are the same temporal operators and IO relations can be extracted.

{**AX**(*a+* , *b+*) , **AF**(*a+* , *c+*) , **AX**(*d+* , *c+*) , **AF**(*d+* , *b-*) , **AF**(*d-* , *b+*)}

These relations can be satisfied *globally* on all paths. Thus temporal operator **AG** can be introduced as follows:

AG[**AX**(*a+* , *b+*) ∨ **AF**(*a+* , *c+*) ∨ **AX**(*d+* , *c+*) ∨ **AF**(*d+* , *b-*) ∨ **AF**(*d-* , *b+*)]

Since **AF** expresses “sometime in the future for all paths”, the *next* operator **AX** can be covered as **AX** ⊆ **AF**. Thus, for IO relations (*a-* , *c-*) and (*d-* , *c-*) on the path **(B)** and **(C)**, temporal formulas can be specified as follows:

AG[**AF**(*a-* , *c-*) ∨ **AF**(*d-* , *c-*)]

Therefore,

AG[**AX**(*a+* , *b+*) ∨ **AF**(*a+* , *c+*) ∨ **AX**(*d+* , *c+*) ∨ **AF**(*d+* , *b-*) ∨ **AF**(*a-* , *c-*) ∨ **AF**(*d-* , *c-*) ∨ **AF**(*d-* , *b+*)].

[5] Here, **AX**(*a+* , *c+*) and **AF**(*d+* , *c+*) can be combined as **AF**(*a+* ∧ *d+* , *c+*) because transition *c+* occurs by transitions *a+* and *d+* at the next. Thus the formulas can be specified as follows:

AG[**AX**(*a+* ∧ *d-* , *b+*) ∨ **AF**(*a+* ∧ *d+* , *c+*) ∨ **AF**(*d+* , *b-*) ∨ **AF**(*d+* , *b-*) ∨ **AF**(*a-* , *c-*) ∨ **AF**(*d-* , *c-*)]

This temporal specification can express liveness property. As mentioned above, we can obtain temporal formulas inductively.

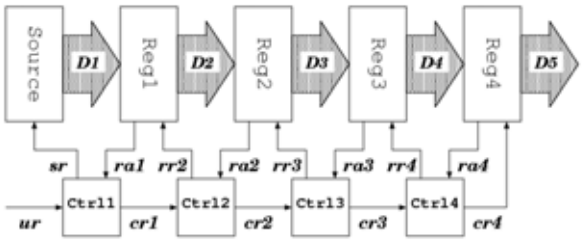

Figure 5: An asynchronous pipeline.

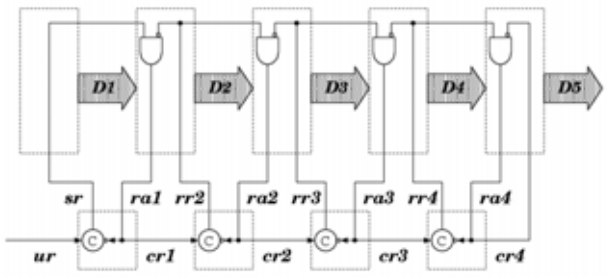

Figure 6: A low level construction of the pipeline.

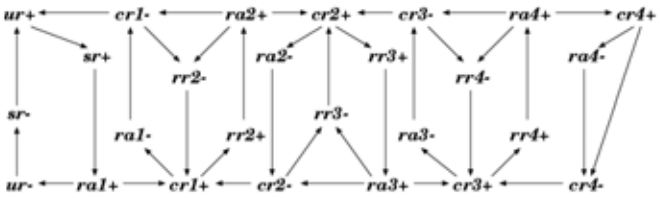

Figure 7: An STG of the pipeline.

IV. Specification Example

In this section, we demonstrate specification of an asynchronous pipeline shown in fig.5. An STG specification of the pipeline in fig.5 (middle) only shows behaviors of **ctrl** modules because arbitration modules are extremely important parts for asynchronous systems. First, temporal formulas are specified without our proposed method as follows:

A. Specification without the proposed method

[**ctrl1**] **AG**[**AX**(*ur+* , *sr+*) ∨ **AF**(*ur+* ∧ *sr+* , *ra1+*) ∨ **AF**(*ur+* ∧ *sr+* ∧ *ra1+* , *cr1+* ∧ *ur1-*) ∨ **AF**(*ra1+* , *sr-*) ∨ **AF**(*cr1+* , *ra1-*) ∨ **AX**(*ra1-* , *cr1-*) ∨ **AF**(*sr-* ∧ *cr1-* , *ur+*)]

[**ctrl2**] **AG**[**AX**(*cr1+* ∧ *cr2-* , *rr2+*) ∨ **AF**(*cr1+* ∧ *rr2+* , *ra2+*) ∨ **AX**(*cr1+* ∧ *rr2+* ∧ *ra2+* , *cr2+*) ∨ **AF**(*cr2+* , *ra2+*) ∨ **AF**(*rr2+* ∧ *ra2+* , *cr1-*) ∨ **AX**(*ra2-* ∧ *cr1-* , *rr2-*) ∨ **AF**(*cr2-* , *cr1+*)]

[**ctrl3**] **AG**[**AX**(*cr2+* ∧ *cr3-* , *rr3+*) ∨ **AF**(*cr2+* ∧ *rr3+* , *ra3+*) ∨ **AX**(*cr2+* ∧ *rr3+* ∧ *ra3+* , *cr3+*) ∨ **AF**(*rr3+* ∧ *ra3+* , *cr2-*) ∨ **AF**(*ra3+* ∧ *cr2-* , *rr3-*) ∨ **AF**(*cr3+* , *ra3-*) ∨ **AX**(*ra3-* , *cr3-*) ∨ **AF**(*rr3-* ∧ *cr3-* , *cr2+*)]

[**ctrl4**] **AG**[**AX**(*cr3+* ∧ *cr4-* , *rr4+*) ∨ **AF**(*cr3+* ∧ *rr4+* , *ra4+*) ∨ **AX**(*cr3+* ∧ *rr4+* ∧ *ra4+* , *cr4+*) ∨ **AF**(*rr4+* ∧ *ra4+* , *cr3-*) ∨ **AF**(*ra4+* ∧ *cr3-* , *rr4-*) ∨ **AF**(*cr4+* , *ra4-*) ∨ **AX**(*ra4-* , *cr4-*) ∨ **AF**(*rr4-* ∧ *cr4-* , *cr3+*)]

This specification is the result of all behaviors for the **ctrl** modules in fig.2 because temporal formulas are considered not only in input-output order relations but also in output-input order relations, such as relations between *sr+* and *ra1+*. Next, we indicate temporal formulas with our proposed method as follows:

B. Specification with the proposed method

[**ctrl1**] **AG**[**AX**(*ur+* ∧ *ra1-* , *sr+*) ∨ **AX**(*ur+* ∧ *ra1+* , *cr1+*) ∨ **AX**(*ur-* , *sr-*) ∨ **AX**(*ra1-* , *cr1-*)]

[**ctrl2**] **AG**[**AX**(*cr1+* ∧ *ra2-* , *rr2+*) ∨ **AX**(*cr1+* ∧ *ra2+* , *cr2+*) ∨ **AX**(*cr1-*, *rr2-*) ∨ **AX**(*ra2-* , *cr2-*)]

[**ctrl3**] **AG**[**AX**(*cr2+* ∧ *ra3-* , *rr3+*) ∨ **AX**(*cr2+* ∧ *ra3+* , *cr3+*) ∨ **AX**(*cr2-*, *rr3-*) ∨ **AX**(*ra3-* , *cr3-*)]

[**ctrl4**] **AG**[**AX**(*cr3+* ∧ *ra4-* , *rr4+*) ∨ **AX**(*cr3+* ∧ *ra4+* , *cr4+*) ∨ **AX**(*cr3-*, *rr4-*) ∨ **AX**(*ra4-* , *cr4-*)]

These temporal formulas considered only input-output order relations by our proposed method, and note that no **AF** temporal operator is used. Thus, the specifications are only used **AX** computation path.

V. Verification Results

We verify some asynchronous bench marks as shown in the table. All these circuits are performed on an Intel Pentium-III 800Mhz processor with 512Mb of RAM under Vine Linux 2.6. The table lists the results of such comparative circuits. For each circuit, we report the number of boolean variables necessary to represent the corresponding model ("**BDD** vars"), and memory and time required by the systems to analyze the model. In the table, some circuits can be found in the distribution of NuSMV[11].

Here, pipeline4 is an asynchronous pipeline shown in fig.5. The number # in pipeline # refers to the number of stages, for example, pipeline10 consists of 10 stages. C-element4 consists of 3 Muller C elements with 4 inputs and 1 output as shown in fig.3 (upper), and queue4 consists of 4 queue modules as shown in fig.3 (lower). For small circuits such as C-element4, queue4 and pipeline4, the memory is the same and time is not much difference 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 pipeline modules.

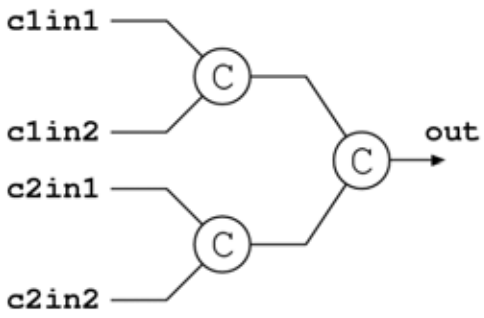

Figure 8: C-element4.

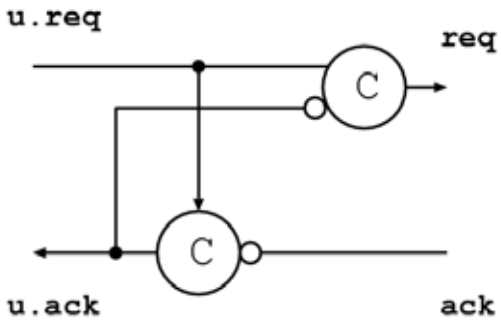

Figure 9: A queue module.

VI. CONCLUSION

Table: Verification results

Circuit Name	BDD Vars	With our method		Without our method	
		Memory (KB)	Time (secs)	Memory (KB)	Time (Secs)
C-element4	19	4355	0.11	4355	0.1
C-element16	217	5283	0.32	5283	0.39
queue4	55	4430	0.11	4430	0.12
Pipeline4	69	4457	0.14	4507	0.19
Pipeline10	141	5424	0.51	5551	0.82
Pipeline20	261	15207	34.45	15358	125.42
dme1	359	6397	0.59	6430	0.68
dme2	369	8408	1.29	8445	1.35
abp4	67	5184	0.57	5811	1.12
Pci	129	6634	0.98	6727	1.08

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 circuits (or systems), not considering output-input order relations. Furthermore, we defined 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 the verification tasks are reduced for memory and time with our proposed method. For the future works, we will consider only strong temporal order relations, and will check structures of complex systems with the relations.

REFERENCES

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

[2] M. Huth and M. Ryan: *Logic in Computer Science*, Cambridge University Press, 2001.

[3] M. Dwyer: *Model Checking Software*, Springer, 2001.

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

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

[6] B.Berard et al.: *Systems and Software Verification – Model Checking Techniques and Tools*, Springer, 2001.

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

[8] Eike Best, Raymond Devillers and Maciej Koutny: *Petri Net Algebra*, Springer, 2001.

[9] Chikatoshi Yamada, Yasunori Nagata and Zensho Nakao: "Efficient Verification of Asynchronous Circuits Exploiting Temporal Order-Based Reduced-STG," *Proc. of 2001 International Technical Conference on Circuit/Systems, Computers and Communications* Vol. II, ITC-CSCC2001, pp.965-968, July 2001.

[10] 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.

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

[12] *SMV*, http://www-2.cs.cmu.edu/~modelcheck/smv.html

[13] Hiromi Hiraishi: "Yet Another Image Computation for Symbolic Model Verifier SMV," *IEICE Trans.* D-I Vol.J82-D-I No.7 pp.791-798, 1999.

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

[15] C.Yamada, Y.Nagata, and M.Mukaidono: "A Specification of Asynchronous Systems Using Temporal Logic," *Notes on Multiple-Valued Logic in Japan*, Vol.23, No.15, 2000.

[16] Tatsuhiro Tsuchiya, Shin'ich Nagano, Rohayu Bt Paidi, and Tohru Kikuno: "Symbolic Model Checking for Self-Stabilizing Algorithms," *IEEE Trans. Parallel and Distributed Systems*, Vol.12, No.6, 2001.

[17] 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.

[18] 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.