Regular Temporal Logic Expressively Equivalent to Finite Automata and Its Application to Logic Design Verification

HIROMI HIRAISHI*, KIYOHARU HAMAGUCHI**, HIROSHI FUJII***
and SHUZO YAJIMA**

Due to the progress of VLSI technologies, logic circuits have become more complex. As a result, the possibilities of design errors have increased. Logic design verification, therefore, has become more important as a means of guaranteeing the correctness of logic degign. Logic design verification requires mathematical logic with enough expressive power to describe the behavior of logic circuits. While propositional logic is known to be equivalent to combinatorial logic circuits, a class of mathematical logic that corresponds to sequential machines and/or finite automata has not yet been clarified.

This paper introduces various types of Regular Temporal Logic (RTL), which is expressively equivalent to finite automata and can express the notion of time explicitly. A design verification algorithm for sequential machines based on a model-checking method of the RTL is also given. Although the complexity of the model-checking problem of the RTL is non-elementary, the proposed model-checking algorithm is efficient and still runs in a time proportional to the size of the structure models. An RTL model checker based on the proposed algorithm is implemented, and it is shown that it can determine whether the designs for medium-size sequential machines allow them to satisfy their specifications in a reasonable time and space.

1. Introduction

With the progress of VLSI technologies, the designs of logic circuits are becoming more and more complex. As a result, the possibility of design errors is increasing, and this greatly affects the development cost and time. It is therefore important to establish new logic design methodologies that make it possible to verify the correctness of logic design.

Conventional logic simulations do not meet this requirement because they cannot guarantee correctness of design except for the input patterns simulated. To guarantee the correctness of a design, we need to prove it formally by describing its specification/design and verifying its correctness according to some formal mathematical logic.

Although propositional logic is known to correspond to combinatorial logic circuits, we currently have no finite mathematical logics that correspond to sequential circuits and/or their mathematical models. The construction of such mathematical logic systems is strongly desired not only in the field of logic design verification and concurrent processes but also in the field of mathematical logic itself.

As a mathematic logic for logic design verification, temporal logic [10], which can express the notion of time explicitly, has been studied widely in the field of formal design verification of protocols and logic circuits. Some practical formal design verification systems have been developed on the basis of temporal logic [1, 2, 3, 4, 5, 11]. The temporal logic used in these systems, however, does not have enough expressive power to describe the specifications of any finite automata. Because of the weak expressive power of conventional temporal logic, there have been several attempts to extend its expressive power. Wolper et al. introduced temporal operators associated with right linear grammar and/or Büchi automata [12, 13] and achieved on expressive power equivalent to that of finite automata. In order to describe specifications in their logic, however, it is necessary to introduce temporal operators that correspond in a sence to the automaton to be designed. We therefore need an infinite number of temporal operators to describe the specifications of any finite automaton in general. Moszkowski proposed a more powerful temporal logic named *interval temporal logic* (ITL) [9], but its expressive power is too strong and its satisfiability problem is undecidable. It is therefore difficult to use ITL as a basis for logic design verification.

This is a translation of the paper that appeared originally in Japanese in Transactions of IPSJ, Vol. 31, No. 7 (1990), pp. 1134–1145.

*Department of Information and Communication Sciences, Faculty of Engineering, Kyoto Sangyo University.

**Department of Information Science, Faculty of Engineering, Kyoto University.

***Information and Communication Processing Research Laboratory, NTT.

Considering the above stated problems, we have proposed various classes of *regular temporal logic* (RTL) [6, 14, 17] that correspond to regular sets (regular sets can be regarded as input/output sequences (behavior sequences) of finite automata), ε-free reguale sets (a subclass of regular sets), and ∞ and/or ω regular sets, which include infinite sequences.

Regular temporal logic contains only three temporal operators, which express notions of time, namely, "*next time*," "*next interval*," and "*repeat*," in addition to the conventional propositional operators. Its expressive power is equivalent to that of finite automata. We can therefore describe the input/output specifications of any logic circuit in RTL.

In this paper, we first give a unified definition for various classes of temporal logics and discuss their expressive power. In order to verify that a given automaton or a sequential machine meets their specifications, we also propose an efficient model-checking algorithm of RTL. A formal logic design verification system based on the proposed model-checking algorithm has been implemented, and some verification examples show that it can verify the correctness of the design of sequential machines with several hundred states in a reasonable time and space.

This paper is organized as follows: Section 2 gives the basic notations and definitions of finite automata, infinite strings, and so on. Four classes of regular temporal logics are introduced. Section 3 discusses basic approaches to verification based on one class of RTL named ε-free RTL. In Section 4, a verification algorithm called model-checking algorithm is described and its computational complexity is discussed. Section 5 explains the verification system (model checker) based on the proposed algorithm. Examples of design verification are also given and show that the RTL model checker is useful from a practical point of view. Section 6 concludes this paper by summarizing the results.

2. Regular Temporal Logic

2.1 Basic Notations

This section gives the basic notation and definitions of basic terminologies.

A *finite word* over an alphabet Σ is a finite non-null sequence of symbols from Σ. An ω *wold* is a countably infinite sequence of symbols from Σ. An *empty word* is a null sequence that contains no symbols and is denoted by ε. $\Sigma^\dagger$ is a set of all finite words over Σ. Σ^ω is a set of all ω words over Σ. $\Sigma^* = \Sigma^\dagger \cup \{\varepsilon\}$ and $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$. An element in Σ^∞ is called a *word* over Σ.

For a word x over Σ, $|x|$ represents the length of x. $x(i)$ refers to the i-th symbol of x and x^i represents the suffix sequence of x starting at the i-th symbol of x. Note that $x^1 = x$, $|\varepsilon| = 0$, and $|x| = \omega$ if x is an ω word. We also define that $x(i) = \varepsilon$ and $x^i = \varepsilon$ for $i > |x|$.

Concatenation of two words x and y over Σ, denoted by xy, is defined as follows:

1. $\varepsilon x = x\varepsilon = x$.
2. If $x \in \Sigma^\dagger$ and $y \in \Sigma^\dagger \cup \Sigma^\omega$,
   - $xy(i) = x(i)$ for any integer i such that $1 \le i \le |x|$.
   - $xy(|x| + i) = y(i)$ for any integer i such that $1 \le i$.
3. If $x \in \Sigma^\omega$, $xy = x$.

A set of words over Σ is called a *language* over Σ. L_1L_2 denotes a concatenation of two languages L_1 and L_2 over Σ, and is defined as $L_1L_2 = \{xy \mid x \in L_1, y \in L_2\}$. Various types of closure of a language L over Σ are defined as follows: $L^\dagger = \bigcup_{i=1}^{\infty} L^i$, $L^* = \bigcup_{i=0}^{\infty} L^i$, $L^\omega = \{x_1x_2 \cdots \mid x_1, x_2, \cdots \in L \cap \Sigma^\dagger\}$, and $L^\infty = L^* \cup L^\omega$, where $L^0 = \{\varepsilon\}$ and $L^{i+1} = L^iL \, (i \ge 0)$.

Definition 1 (Regular Set). Let us consider the following production rules for a class of languages, $\mathscr{R}$, over Σ:

A1. An empty set $\emptyset$ is an element of $\mathscr{R}$.
A2. If $s \in \Sigma$, a set $\{s\}$ is an element of $\mathscr{R}$.
A3. In L_1, $L_2 \in \mathscr{R}$, $L_1 \cup L_2$ and L_1L_2 are elements of $\mathscr{R}$.
A4. If $L \in \mathscr{R}$, L^* is an element of $\mathscr{R}$.
A5. If $L \in \mathscr{R}$, L^ω is an element of $\mathscr{R}$.

If a language L over Σ is an element of the class of laguages $\mathscr{R}$ constructed by the finite application of the above rules A1-A5, L is called an ∞ *regular set*. In this case, if $L \subseteq \Sigma^\omega$, L is called an ω *regular set*; if $\varepsilon \notin L$, L is called an ε-*free* ∞ *regular set*. If L is an element of a class of languages obtained by the finite applications of the above rules A1-A4, L is simply called a *regular set*. In this case, if $\varepsilon \notin L$, L is called ε-*free regular set*.

2.2 Dfinition of Regular Temporal Logic

Regular Temporal Logic (RTL for short) is an extension of propositional logic with three temporal operators "$\bigcirc$", " : ", and "$\boxdot$" whose intuitive meanings are "*next time*," "*next interval*," and "*repeat*," respectively.

Definition 2(Syntax). Let AP be a set of atomic propositions. RTL formulas are defined inductively as follows:

B1. If $p \in AP$, then p is an RTL formula.
B2. If η is an RTL formula, then so are $(\neg\eta)$, $(\bigcirc\eta)$, and $(\boxdot\eta)$.
B3. If η and ξ are RTL formulas, then so are $(\eta \vee \xi)$ and $(\eta : \xi)$.

Let Σ be a set of states. The value of an RTL formula is defined according to a sequence of states in Σ. Let D be a semantic domain of RTL formulas. If $D = \Sigma^\dagger$, the RTL is called ε-*free finite RTL*. If $D = \Sigma^*$, it is called *finite RTL*. If $D = \Sigma^\infty - \{\varepsilon\}$, it is called ε-*free infinite RTL*. If $D = \Sigma^\infty$, it is called *infinite RTL*.

The ε-free finite RTL is exactly the same temporal logic as proposed in [6, 7]. The finite RTL is exactly the same temporal logic as proposed in [16, 17]. The ε-free infinite RTL is exactly the same temporal logic as proposed in [14, 15].

The formal semantics of RTL formulas are given in the following.

Definition 3 (Linear Model). A tuple $M=(\Sigma, I)$ is called a *linear model* of RTL, where Σ is a set of states and I is a function for interpreting atomic propositions. I labels each state with a set of atomic porpositions that are true in that state. For the ε-free finite/infinite RTL, $I: \Sigma \to 2^{AP}$; for the finite/infinite RTL, I: $\Sigma \cup \{\varepsilon\} \to 2^{AP}$.

Definition 4(Value of RTL Formulas). M, $\sigma \models \eta$ denotes that the RTL formula η is *true* for the state sequence σ over the linear model M. If there is no confusion, we sometimes omit M and just write $\sigma \models \eta$. Let p be an atomic proposition, η and ξ be RTL formulas, and D be a semantic domain of the RTL. We also assume that $\sigma \in D$. The relation $\models$ is defined inductively as follows:

C1. $\sigma \models p \Leftrightarrow p \in I(\sigma(1))$.

C2. $\sigma \models (\neg \eta) \Leftrightarrow \sigma \not\models \eta$.

C3. $\sigma \models (\eta \vee \xi) \Leftrightarrow \sigma \models \eta$ or $\sigma \models \xi$.

C4. $\sigma \models (\bigcirc \eta) \Leftrightarrow$
- In the case of ε-free RTL, $|\sigma| \geq 2$ and $\sigma^2 \models \eta$.
- Otherwise $\sigma^2 \models \eta$.

C5. $\sigma \models (\eta : \xi) \Leftrightarrow$
- If $|\sigma| \neq \omega$, $\sigma = \sigma_1 \sigma_2$ and there exist σ_1 and σ_2 in D such that $\sigma_1 \models \eta$ and $\sigma_2 \models \xi$.
- If $|\sigma| = \omega$, $\sigma \models \eta$ or there exist σ_1 and σ_2 in D such that $\sigma = \sigma_1 \sigma_2$, $|\sigma_1| \neq \omega$, $|\sigma_2| = \omega$, $\sigma_1 \models \eta$, and $\sigma_2 \models \xi$.

C6. $\sigma \models (\boxdot \eta) \Leftrightarrow$
- If $|\sigma| \neq \omega$, there exist $\sigma_i \in D$ $(1 \leq \forall i \leq m)$ such that $\sigma = \sigma_1 \sigma_2 \cdots \sigma_m$ and $\sigma_i \models \eta (1 \leq \forall i \leq m)$.
- If $|\sigma| = \omega$, there exist $\sigma_i \in D$ $(1 \leq \forall i \leq m)$ such that $\sigma = \sigma_1 \sigma_2 \cdots \sigma_m$, $\sigma_i \models \eta (1 \leq \forall i \leq m)$, $|\sigma_i| \neq \omega (1 \leq \forall_i < m)$, and $|\sigma_m| = \omega$; or there exist $\sigma_i \in D$ such that $\sigma = \sigma_1 \sigma_2 \cdots$, and $\sigma_i \models \eta$ and $|\sigma_i| \neq \omega$ for $\forall i \geq 1$.

Intuitively, "$\bigcirc \eta$" indicates that η holds for the sequence starting from the next state. "$\eta : \xi$" means that η holds for the former part of the sequence and ξ holds for the latter part of the sequence. "$\boxdot \eta$" indicates that η holds repeatedly.

We also use conventional abbreviations such as "$\wedge$" for *conjunction*, "$\Rightarrow$" for *implication*, "$\equiv$" for *equivalence*, "V_T" for *tautology*, and "V_F" for *invalid* formula. We also assume that unary operators have higher precedence than binary operators. When there is no ambiguity, we usually omit parentheses.

If M, $\sigma \models \eta$ for some linear model M and some state sequence $\sigma \in D$, η is said to be *safisfiable.*

2.3 Expressive Power of Regular Temporal Logic

Let η be an RTL formula. The set of sequences for which η becomes *true* is represented as $L\langle \Sigma, I \rangle(\eta) = \{\sigma \mid \sigma \in D, \sigma \models \eta\}$. When there is no ambiguity, we abbreviate $L\langle \Sigma, I \rangle(\eta)$ as simply $L(\eta)$. According to the definition of RTL (Definition 4), we have the following lemma:

Lemma 1. Let p be an atomic proposition and η and ξ be RTL formulas. For a linear model $M=(\Sigma, I)$ of RTL,

1. $L(p) = \{s\sigma \mid s \in \Sigma, p \in I(s), s\sigma \in D\} \cup \{\varepsilon \mid \varepsilon \in D, p \in I(\varepsilon)\}$
2. $L(\neg \eta) = D - L(\eta)$
3. $L(\eta \vee \xi) = L(\eta) \cup L(\xi)$
4. $L(\bigcirc \eta) = \Sigma L(\eta) \cup (\{\varepsilon\} \cap L(\eta))$
5. $L(\eta : \xi) = L(\eta) L(\xi)$
6. $L(\boxdot \eta) = L(\eta)^{\dagger} \cup (L(\eta)^{\omega} \cap D)$

Let $\mathscr{T}$ be one of the various classes of RTL (namely, ε-free finite RTL, finite RTL, ε-free infinite RTL, or infinite RTL). Let $\mathscr{R}$ be a class of languages. $\mathscr{T}$ is said to be *expressively equivalent* to $\mathscr{R}$ if and only if the following two conditions hold:

- For any RTL formula $\eta \in \mathscr{T}$, $L\langle \Sigma, I \rangle(\eta)$ is an element of $\mathscr{R}$.
- For any language $R \in \mathscr{R}$, there exist an RTL formula $\eta \in \mathscr{T}$ and an interpretation function I such that $L\langle \Sigma, I \rangle(\eta) = R$.

We have the following theorem about the expressive power of the four classes of RTL defined in the previous section.

Theorem 1 (Expressive Power of Regular Temporal Logic)

1. ε-free finite RTL is expressively equivalent to an ε-free regular set [6].
2. Finite RTL is expressively equivalent to a regular set [17].
3. ε-free infinite RTL is expressively equivalent to an ε-free ∞ regular set [14, 15].
4. Infinite RTL is expressively equivalent to an ∞ regular set.

(Proof) We give a proof only for the fourth case. From Lemma 1, it is clear that $L(\eta)$ is an ∞ regular set for any infinite RTL formula η. Conversely, let R be a ∞ regular set over Σ. Let $AP = \{p_s \mid s \in \Sigma\}$ and $I(p_s) = \{s\}$. Let $F(R)$ denote an infinite RTL formula η such that $L(\eta) = R$. $F(R)$ can be calculated inductively as follows:

1. $F(\emptyset) = V_F$
2. $F(\{s \in \Sigma\}) = p_s \wedge \neg p_0 \wedge \bigcirc p_0$
3. $F(R_1 \cup R_2) = F(R_1) \vee F(R_2)$
4. $F(R_1 R_2) = F(R_1) : F(R_2)$
5. $F(R^*) = p_0 \vee \boxdot F(R_1)$
6. $F(R^{\omega}) = \boxdot (F(R) \wedge Fin \wedge \neg p_0)$,

where p_0 is an atomic proposition such that $I(p_0) = \{\varepsilon\}$, $\neg p_0 \wedge \bigcirc p_0$ represents a set of all sequences whose length are exactly one, and $Fin \triangleq \neg (V_T : V_F)$ represents a set of all finite sequences that include an empty word ε.

A set of behavior sequences (input/output sequences) of a finite automaton is equivalent to an (ε-free) regular set when we are concerned only with finite behavior; it is equivalent to an ω regular set if we are concerned only with infinite behavior; it is equivalent to an (ε-free) ∞ regular set if we are concerned with both. In this sense, RTL is expressively equivalent to finite automata, and any behavior sequences of any finite automaton can be completely described in RTL.

3. Design Verifincation of Sequential Machines

3.1 Description of Specification and Design

In this section we discuss how to verify the design of sequential machines by using RTL. Let us consider verification of a deterministic Mealy-type or Moore-type sequential machine with n binary input signals $X=\{x_1, x_2, \cdots, x_n\}$ and m binary output signals $Z=\{z_1, z_2, \cdots, z_m\}$. Its behavior sequence (input-output sequence) is represented by a finite sequence ρ over $2^{X\cup Z}$ such that $\rho(k) \triangleq \{x_i | x_i=1$ at the k-th input$\} \cup \{z_j | z_j=1$ at the k-th output$\}$.

We assume that a design specification is given in terms of a set of possible behavior sequences of a machine to be designed. Let us consider a finite behavior sequence of any length. Let p_{x_i} and p_{z_j} be atomic propositions associated with input signal x_i and output signal z_j, respectively, such that p_{x_i} is *true* iff $x_i=1$ and p_{z_j} is *true* iff $z_j=1$. Then we can describe the set of all possible finite behavior sequences of the machine in ε-free RTL. Though we can express an infinite behavior sequence by using ε-free infinite RTL, we will discuss the verification problem for infinite behavior in a future paper. Hereafter, RTL refers to ε-free RTL.

For example, let us consider design verification of the T flipflop shown in Fig. 1. We assume that the output z is 0 in the initial state. The output z changes its value if and only if the previous input x is 1.

Let p_x be an atomic proposition whose value is *true* if and only if the input $x=1$, and let p_z be an atomic proposition whose value is *ture* if and only if the output $z=1$. Then the specification of the T flipflop can be written as follows:

$$\eta \triangleq p_x \equiv (p_z \equiv \bigcirc \neg p_z)$$
$$spec \triangleq \neg p_z \wedge \Box(\eta \vee LEN1),$$

where $LEN1 \triangleq \neg \bigcirc V_T$, $\Diamond\xi \triangleq \xi \vee (V_T : \xi)$, and $\Box\xi \triangleq \neg\Diamond\neg\xi$.

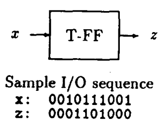


Fig. 1 T flipflop.

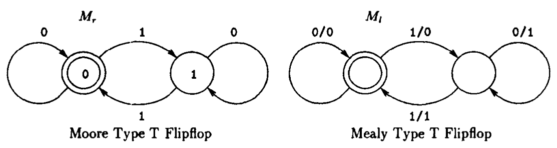


Fig. 2 Two designs of T flipflop.

η indicates that the value of z at the next time differs from that at the current time if and only if the current value of x is 1. $LEN1$, $\Diamond\xi$, and $\Box\xi$ mean that "the length of the sequence is 1," "ξ will become *true* at some time in the future or present," and "ξ is always *true* from now on." The specification *spec* of the T flipflop means that the initial value of z is 0 and η always holds from now on. Since this property of the T flipflop is meaningless for a time sequence whose length is 1, $LEN1$ is used in *spec* so that *spec* ignores a non-existing next state on the last occasion of a time period under consideration.

The design of this T flipflop is assumed to be given as either a Mealy-type or a Moore-type sequential machine, as shown in Fig. 2. The verification problem then becomes to check whether *spec* holds for all possible finite behavior sequences of the designed machine.

3.2 Structure Model

In order to treat possible behavior sequences more easily, we define a structure model.

Definition 5 (Structure Model). A *structure model* is a 4-tuple $K=(\Sigma, I, R, \Sigma_0)$, where

1. (Σ, I) is a linear model of RTL.
2. $R \subseteq \Sigma \times \Sigma$ is a binary relation over Σ and represents a transition relation between states.
3. $\Sigma_0 \subseteq \Sigma$ is a set of initial states.

A structure model is a kind of Kripke model [8] with a set of initial states. For a structure model $K=(\Sigma, I, R, \Sigma_0)$, a finite sequence of states $\pi = s_1 s_2 \cdots s_n \in \Sigma^+$ is called a *finite path* from s_1 if and only if $(s_i, s_{i+1}) \in R$ for any i such that $1 \leq i < n-1$.

The truth value of an RTL formula with respect to a structure model K is defined as follows.

Definition 6 (Truth Value with Respect to a Structure Model). If $\sigma \models \eta$ for a finite path σ from a state s of a structure model K, η is said to be *true with respect to* $\langle K, s\rangle$($\langle K, s\rangle$-true); otherwise, it is said to be *false with respect to* $\langle K, s\rangle$($\langle K, s\rangle$-false). Furthermore, if η is $\langle K, s_0\rangle$-true for some initial state $s_0 \in \Sigma_0$, η is said to be *true over* K(K-true); otherwise, η is said to be *false over* K(K-false).

Let $M=(X, Z, S, \delta, \lambda, s_0)$ be a deterministic sequential machine, where

1. X is a finite set of binary input signals.
2. Z is a finite set of binary output signals.
3. S is a finite set of states.
4. $s_0 \in S$ is the initial state.
5. $\delta : 2^X \times S \rightarrow S$ is the next state function.
6. λ is the output function such that
   - For Moore type; $\lambda : S \rightarrow 2^Z$ is a total function over S.
   - For Mealy type; $\lambda : 2^X \times S \rightarrow 2^Z$ and has the same domain as δ.

The structure model K corresponding to M is constructed as follows:

$$\Sigma \triangleq \{s_{i,j} | s_i \in S, j \in 2^X, \delta(j, s_i) \text{ is defined.}\}$$

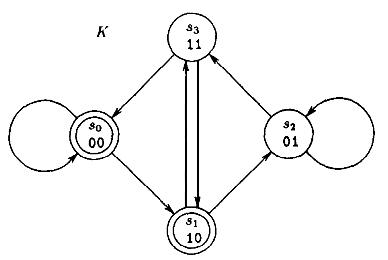


Fig. 3 Structure model of T flipflop.

$$I(s_{i,j}) \triangleq \begin{cases} \{p_x \mid x \in j\} \cup \{p_z \mid z \in \lambda(j, s_i)\} & \text{for Mealy type} \\ \{p_x \mid x \in j\} \cup \{p_z \mid z \in \lambda(s_i)\} & \text{for Moore type} \end{cases}$$

$$R \triangleq \{(s_{i,j}, s_{i',j'}) \mid s_{i,j}, s_{i'j'} \in \Sigma, \delta(j, s_i) = s_{i'}$$

$$\Sigma_0 \triangleq \{s_{0,j} \in \Sigma\}.$$

The structure model corresponding to the T flipflop is shown in Fig. 3. Let $|E|$ be the number of transition edges of a sequential machine M. Then the size of the structure model K corresponding to M becomes as follows:

$$O(|\Sigma|) = O(|E|) = O(|\Sigma| 2^{|X|})$$
$$O(|R|) = O(|E| 2^{|X|}) = O(|\Sigma| 2^{2|X|})$$
$$O(|\Sigma| + |R|) = O((|\Sigma| + |E|) 2^{|X|})$$

As can be easily seen from the way in which a structure model is constructed, there is a one-to-one correspondence between a set of possible behavior sequences of M and a set of paths from an initial state of the corresponding structure model K of M. The verification problem of a sequential machine M over its corresponding structure model K becomes one of checking whether "*spec* is *true* for every finite path from any initial state of K" or whether "$\neg$*spec* is K-false." This is done by a model-checking method described in the next section.

4. Model-Checking Approach

Model checking determines whether an RTL formula η is K-true or not. Before describing the model checking algorithm, we first define the *derivation* of the RTL formula.

4.1 Derivative of an ε-Free Finite Regular Temporal Logic Formula

The *derivative* of an RTL formula η by a state $s \in \Sigma$, denoted by η/s, gives a formula that should hold in the next state so that η may hold in the present state. It is formally defined inductively as follows:

$$p/s \triangleq \begin{cases} V_T \text{ if } p \in I(s) \\ V_F \text{ otherwise} \end{cases}$$

$$(\neg\eta)/s \triangleq \neg(\eta/s)$$

$$(\eta \vee \xi)/s \triangleq (\eta/s) \vee (\xi/s)$$

$$(\bigcirc\eta)/s \triangleq \eta$$

$$(\eta{:}\xi)/s \triangleq \begin{cases} \xi \vee ((\eta/s){:}\ \xi) \text{ if } s \vDash \eta \\ (\eta/s){:}\ \xi \quad \text{otherwise} \end{cases}$$

$$(\boxdot\eta)/s \triangleq \begin{cases} (\eta/s) \vee ((\eta/s){:}\boxdot\eta) \vee \boxdot\eta \text{ if } s \vDash \eta \\ (\eta/s) \vee ((\eta/s){:}\boxdot\eta) \quad \text{otherwise.} \end{cases}$$

Derivative of η by a sequence of states $\sigma = s_1 s_2 \cdots s_n \in \Sigma^\dagger$ is defined as

$$\eta/\sigma \triangleq ((\cdots((\eta/s_1)/s_2)\cdots)/s_n).$$

We also define that $\eta/\varepsilon \triangleq \eta$. Note that $V_T/s = V_T$ and $V_F/s = V_F$.

From the definition of derivatives, the next lemma clearly holds:

Lemma 2. Let σ be a finite sequence over Σ such that $|\sigma| \geq 2$. Then the necessary and sufficient condition for $\sigma \vDash \eta$ is $\sigma^2 \vDash \eta/\sigma(1)$.

Theorem 2. Let $K = (\Sigma, I, R, \Sigma_0)$ and η be a structure model and an RTL formula, respectively. The necessary and sufficient condition that η is (K, s)-true for a state $s \in \Sigma$ is that either $s \vDash \eta$ or there exists a state s' such that $(s, s') \in R$ and η/s is (K, s')-true.

(Proof) From Definition 6, η is $\langle K, s\rangle$-true if and only if $\sigma \vDash \eta$ for some finite path σ on K from s. Considering Lemma 2, this is equivalent to the condition that $s \vDash \eta$ or η/s is $\langle K, s'\rangle$-true for some state s' such as $(s, s') \in R$. (Q.E.D.)

In the calculation of derivatives and checking conditions stated in Theorem 2, we need to obtain the truth value of η for a sequence s whose length is 1. This can be obtained inductively as follows:

1. $s \vDash p \Leftrightarrow p \in I(s)$
2. $s \vDash \eta \vee \xi \Leftrightarrow s \vDash \eta$ or $s \vDash \xi$
3. $s \vDash \neg\eta \Leftrightarrow s \nvDash \eta$
4. $s \nvDash \bigcirc\eta$ always holds.
5. $s \nvDash \eta{:}\xi$ always holds.
6. $s \vDash \boxdot\eta \Leftrightarrow s \vDash \eta$

4.2 Model-Checking Algorithm

It is easy to construct a model-checking algorithm by a depth-first search for the necessary and sufficient condition stated in Theorem 2.

Algorithm 1 (Model-checking algorithm)

Input:	a structure model $K = (\Sigma, I, R, \Sigma_0)$ and an RTL formula η.
Output:	if η is K-true then 'T' else 'F'.
Method:	by *Verify*(K, η) in Fig. 4.

In Fig. 4, *Check* (K, s, η) is a procedure that returns 'T' if η is $\langle K, s\rangle$-ture, or 'F' otherwise. *Addlabel*(s, η, x) registers a label x to a tuple (s, η), which means that the value of η is x in state s. x='F' means that η is $\langle K, s\rangle$-false. x='C' means that the truth value of η in state s is now under investigation. These labels are used to prevent the procedure *Check* from being called more than once for the same pair of s and η. We need not use a label to show that η is $\langle K, s\rangle$-true, because the procedure *Verify* returns 'T' immediately after the procedure *Check* returns 'T'. *Label*(s, η) returns the label of η at s if it is already registered; otherwise it returns *null*.

```
procedure Verify($K, \eta$)
  begin
    for all $s \in \Sigma_0$
      begin
        if Label($s, \eta$) $\neq$ 'F' then
          if Check($K, s, \eta$) = 'T' then return 'T';
      end
    return 'F';
  end
end of procedure

procedure Check($K, s, \eta$)
  begin
    ($x, \xi$) := Derivation($s, \eta$);
    if $x$ = 'T' then return 'T';
    if $\xi = V_T$ then return 'T';
    if $\xi = V_F$ then
      begin
        Addlabel($s, \eta$,'F');
        return 'F';
      end
    Addlabel($s, \eta$,'C');
    for all $s'$ such that $(s, s') \in R$
      begin
        $x$ := Label($s', \xi$);
        if $x$ = NIL then
          if Check($K, s', \xi$) = 'T' then return 'T';
      end
    Addlabel($s, \eta$,'F');
    return 'F';
  end
end of procedure
```

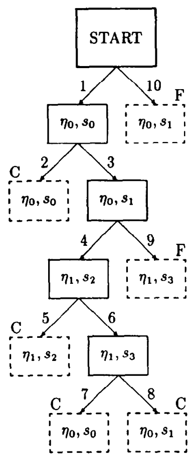
```
procedure Derivation($s, \eta$);
  begin
    switch($\eta$){
    case $\eta \in AP$:
      if $\eta \in I(s)$ then return ('T',$V_T$);
      else return ('F',$V_F$);
    case $\eta = \neg\eta_1$:
      ($x, \xi$) := Derivation($s, \eta_1$);
      if $x$ = 'T' then return ('F',$\neg\xi$);
      else return ('T',$\neg\xi$);
    case $\eta = \eta_1 \vee \eta_2$:
      ($x_1, \xi_1$) := Derivation($s, \eta_1$);
      ($x_2, \xi_2$) := Derivation($s, \eta_2$);
      if ($x_1$ = 'T' or $x_2$ = 'T') then $x$ := 'T';
      else $x$ := 'F';
      return ($x, \xi_1 \vee \xi_2$);
    case $\eta = \bigcirc\eta_1$:
      return ('F',$\eta_1$);
    case $\eta = \eta_1 : \eta_2$:
      ($x, \xi$) := Derivation($s, \eta_1$);
      $\xi := \xi : \eta_2$;
      if $x$ = 'T' then $\xi := \xi \vee \eta_2$;
      return ('F',$\xi$);
    case $\eta = \boxed{:}\eta_1$:
      ($x, \xi$) := Derivation($s, \eta_1$);
      $\xi := \xi \vee (\xi : \eta)$;
      if $x$ = 'T' then $\xi := \xi \vee \eta$;
      return ($x, \xi$);
    }
  end
end of procedure
```

Fig. 4 Model-checking algorithm.

Verify (K, η) calls *Check* (K, s, η) successively for each initial state $s \in \Sigma_0$ if η is not yet proved to be $\langle K, s \rangle$-false. If the procedure *Check* returns 'T' at least once, the procedure *Verify* also returns 'T'; otherwise it returns 'F'.

Check (K, s, η) first calls *Derivation* (K, s, η). *Derivation* (K, s, η) checks whether $s \models \eta$ as well as calculating η/s. It substitutes η/s into ξ and also substitutes either 'T' or 'F' into x according to whether $s \models \eta$ or $s \not\models \eta$. Then it returns 'T' when $\xi = \eta/s$ is V_T or x='T'. If $\xi = \eta/s$ is V_F, it labels the tuple (s, η) 'F', and returns 'F'. Otherwise, it labels the tuple (s, η) 'C'. Then it checks *Label* $(s', \eta/s)$, and if the tuple $(s', \eta/s)$ is not yet labeled, it calls *Check* $(K, s', \eta/s)$ recursively for each next state s' of s successively. It returns 'T' when one of the recursive calls to the procedure *Check* returns 'T'. If every recursive call to the procedure *Check* results in the return value 'F', it labels the tuple (s, η) 'F' and then returns 'F'.

Figure 5 shows an example of model checking of the T flipflop. In Fig. 5, a solid box indicates that *Check* (K, s, η) is called with the argument written inside the solid box. A dashed box indicates that the tuple of the state and the formula written inside the dashed box are already labeled either 'F' or 'C'. The numbers above the edges represent the order of depth-first search over a structure model. In this example, every call to the procedure *Check* ends with a return value 'F', and *Verify* $(K, \neg spec)$ returns 'F'. This means that the design shown in Fig. 2 satisfies the specification *spec*.



η	$\triangleq$	$p_x \equiv (p_z \equiv \bigcirc\neg p_z)$
η_0	$\triangleq$	$\neg spec = \neg(\neg p_z \wedge \Box(\eta \vee LEN1))$
η_1	$\triangleq$	$\neg(p_z \wedge \Box(\eta \vee LEN1))$
s_0	$\not\models$	η_0
s_1	$\not\models$	η_0
s_2	$\not\models$	η_1
s_3	$\not\models$	η_1
η_0/s_0	$=$	η_0
η_0/s_1	$=$	η_1
η_1/s_2	$=$	η_1
η_1/s_3	$=$	η_0

Fig. 5 Verification of T flipflop by the model-checking algorithm.

4.3 Correctness and Complexity

First we show that the number of different formulas obtained by derivations of an RTL formula η repeatedly is at most finite.

Let $D^*(\eta) \triangleq \{\eta/\sigma | \sigma \in \Sigma^*\}$. In the definition of $D^*(\eta)$ we regard two formulas as identical if they can be transformed into one another by using the relations $V_T \vee \xi = V_T$ and $V_F \vee \xi = \xi$, and by the commutative, associative, and idempotence laws of '$\vee$'.

Lemma 3. $D^*(\eta)$ is a finite set.

(Proof) It is shown inductively according to the rules for construction of RTL formulas:

1. $D^*(p)=\{p, V_T, V_F\}$.
2. $D^*(\neg\eta)=\{\neg\xi \mid \xi\in D^*(\eta)\}$.
3. $D^*(\eta_1\vee\eta_2)\subseteq\{\xi_1\vee\xi_2 \mid \xi_1\in D^*(\eta_1), \xi_2\in D^*(\eta_2)\}$.
4. $D^*(\bigcirc\eta)=\{\bigcirc\eta\}\cup D^*(\eta)$.
5. Let $E_1\triangleq\{\vee[\Theta] \mid \Theta\in 2^{D^*(\eta_2)}\}$ and $E_2\triangleq\{\xi_1{:}\eta_2 \mid \xi_1\in D^*(\eta_1)\}$. Then, $D^*(\eta_1{:}\eta_2)\subseteq\{\nu_1\vee\nu_2 \mid \nu_1\in E_1, \nu_2\in E_2\}$
6. Let $F_1\triangleq\{\vee[\Theta] \mid \Theta\in 2^{D^*(\eta)\cup\{\boxdot\eta\}}\}$ and $F_2\triangleq\{\xi_1{:}\boxdot\eta \mid \xi_1\in D^*(\eta)\}$. Then, $D^*(\boxdot\eta)\subseteq\{\nu_1\vee\nu_2 \mid \nu_1\in F_1, \nu_2\in F_2\}$

$\vee[\Theta]$ represents the disjunction of all RTL formulas in Θ. Every set represented by the right-hand sides of these expressions is clearly a finite set. (Q.E.D.)

[Proof of the correctness of Algorithm 1]. Its correctness is clear from the definition of K-true and Theorem 2 except for its termination and loop handling.

Termination: From Lemma 3, the number of different RTL formulas obtained by derivations during the execution of the algorithm is finite. In addition, the procedure *Check* is called only once for the same tuple of a state and an RTL formula. Therefore, Algorithm 1 always terminates.

Loop handling: When *Check* (K, s, η) detects a loop (that is, when $\xi=\eta/s$ and Label $(s', \xi)=$'C' for some s' such as $(s, s')\in R$), it just handles the loop in the same way as Label $(s', \xi)=$'F'. The correctness of this handling is shown as follows. Let σ be a sequence of states from s' to s along the loop. Since Label (s', ξ) $=$'C', $\xi/\sigma=\xi$ and $\sigma_p\not\models\xi$ for any prefix sequence σ_p of σ. Let us assume that $\tau\sigma_p\models\xi$ for some sequence $\tau\in\{\sigma\}^+$. Since $\xi/\sigma=\xi$, $\sigma_p\models\xi$ is shown by repeated application of Lemma 2, which is a contradiction. Therefore, ξ becomes *false* for any finite prefix sequence of the infinite sequence generated by the loop, and we can treat this case in the same way as that of *Label* $(s', \xi)=$'F'.

Next we evaluate the computational complexity of Algorithm 1. Let $\mathscr{N}\triangleq|\mathscr{D}^*(\eta)|$ and $\mathscr{L}$ represent the maximum number of operators contained in a formula in $D^*(\eta)$. $|\eta|$ represents the number of operators in a formula η.

Theorem 3. The time complexity of Algorithm 1 is $O((|\Sigma|(\mathscr{L}\log\mathscr{L}+\log\mathscr{N})+|R|\log\mathscr{N})\mathscr{L}\mathscr{N})$.

(Proof) First we evaluate the time complexity of *Derivation* (s, ξ). The procedure *Derivation* checks whether $s\models\xi$ holds at the same time as it calculates ξ/s. Whether or not $s\models\xi$ can be checked in a time proportional to $|\xi|$ by checking whether $s\models\xi'$ for each subformula ξ' of ξ in a bottom-up fashion. In the calculation of ξ/s, it is sometimes necessary to check whether $s\models\xi'$ for some subformulas ξ' of ξ, which should have been already obtained during the bottom-up calculation. Furthermore, in a derivative operation of an RTL formula, some subformula and/or its derivative may appear twice (in the case of a derivative of " : " or "$\boxdot$"). Even in such cases, the result can be obtained by adding at most three operators for each " : " or "$\boxdot$" if we adopt a graphic representation of formulas that share a common subformula. Basically, therefore, can be found ξ/s in a time proportional to $|\xi|$. In order to guarantee the termination of the algorithm, however, we need to simplify ξ/s by using the idempotence, commutative, and associative laws of '$\vee$'. This can be done by sorting every term in every disjunctive clause. Sorting of terms requires that formulas be compared a total of $O(|\xi/s|\log|\xi/s|)$ times and that each comparison can be done in a time proportional to $|\xi/s|$. Therefore, derivation requires $O(\mathscr{L}^2\log\mathscr{L})$ time, because ξ and ξ/s are in $D^*(\eta)$.

The procedures *Addlabel* and *Label* can be realized with $\log\mathscr{N}$ comparisons by sorting every formula in $D^*(\eta)$. Since each comparison requires a time proportional to $\mathscr{L}$, the time complexity of each of the procedures *Addlable* and *Label* is $O(\mathscr{L}\log\mathscr{N})$.

Furthermore, the procedure *Check* is called a total of $O(|\Sigma|)$ times for each formula in $D^*(\eta)$. As a result, *Derivation* and *Addlabel* will be called at most $|\Sigma|$ times in total. *Label* may be called $|R|$ times in total, because it will be called for the subsequent state of each current state. Therefore, the time complexity of Algorithm 1 is $O((|\Sigma|(\mathscr{L}\log\mathscr{L}+\log\mathscr{N})+|R|\log\mathscr{N})\mathscr{L}\mathscr{N})$.

We have the following theorem on the Deterministic Turing Machine (DTM) space complexity of the model checking problem:

Lemma 4. The DTM space complexity of the satisfiability problem of finite RTL is non-elementary [6].

Theorem 4. The DTM space complexity of the model-checking problem for a finite RTL formula is non-elementary.

(Proof) We show that the satisfiability problem of an RTL formula can be transformed in elementary time to a model-checking problem for an RTL formula. Let AP' be a set of atomic propositions appearing in a RTL formula η. Let Σ be a set consisting of $2^{|AP'|}$ states. Let I be a one-to-one mapping from Σ to $2^{AP'}$. We define R and Σ_0 as $R=\Sigma\times\Sigma$ and $\Sigma_0=\Sigma$. Then for a structure model $K_c=(\Sigma, I, R, \Sigma_0)$, η is K_c-true if and only if η is satisfiable. K_c can be constructed in $O(2^{2|\eta|})$ time. (Q.E.D.)

5. Implementation of a Model Checker and Verification Examples

Although the computational complexity of the model-checking problem is non-elementary for the length of a given RTL formula, as shown in Theorem 4, Theorem 3 states that the model-checking algorithm given in Section 4.2 can execute model checking in a time proportional to the size of a give structure model (that is, $|\Sigma|+|R|$) for a given fixed formula. In order to evaluate our model-checking algorithm, we implemented it on a SUN 3/60 workstation as an RTL model checker.

5.1. RTL Model Checker

In the RTL model checker, RTL formulas are stored basically as labeled directed acyclic binary graphs in the usual way. Labels are associated with nodes and represent either operators or atomic propositions. The graphs that represent a given formula and its subformulas are registered in an internal table. Among the subformulas generated by derivations, those that will/may appear again later are compared with the registered formulas. If the same formula has been already registered, the registered one is used and the nodes used in the generated formula are released to save space; otherwise, the generated formula is registered in the internal table.

In addition to the model checking of a given specification written in RTL, the RTL model checker can produce a sequence of states that contradicts the specification if it is not satisfied. Furthermore, it can show why the specification is not satisfied by representing a state and a subformula that make the specification *false*. This facility is very useful for analyzing design errors when a specification is not satisfied.

5.2 Varification of a Traffic Controller

As a test of its efficiency, the RTL model checker has been used for design verification of a traffic controller [1]. The traffic controller is stationed at the intersection of a two-way highway going north and south and a one-way road going east. It has three input signals (N, S, and E), three output signals (N_GO, S_GO, and E_GO), and five internal signals. N (north), S (south), and E (east) indicate that there is at least one car whose driver intends to cross the intersection to the north, south, and east, respectively. N_GO, S_GO, and E_GO indicate that the traffic light for the corresponding direction is *green*. The controller is designed as a Moore machine. One design, a 'bad design,' which has some design errors, has 43 states and its corresponding structure model has 344 states, while the other (a 'good design') has 31 states and its corresponding structure model has 248 states.

The full specification *spec* for the traffic controller is written in RTL, as shown in Fig. 6. *nocoli* states that the traffic lights for the east direction and the north-south direction never both become *green* at the same time. *ic* represents the input constraints that once N, S, and E have been asserted, they are never turned off until N_GO, S_GO, and E_GO are turned on, respectively. *asn*4, *ass*4, and *ase*4 represent the situation in which at least one driver intends to cross the intersection in the corresponding direction while there are no cars in directions orthogonal to it. *ngoby*4, *sgoby*4, and *egoby*4 state that the traffic lights will be *green* for the corresponding direction within four units of time including the present one. *spec* specifies that the traffic lights in mutually orthogonal directions never both become *green* at the same time, and that if a car arrives at the intersection and there are no cars in deirections orthogonal to it, then the traffic light for its direction will becomes *green* within four units of time including the present one so long as input constraints are satisfied.

$$
\begin{array}{rcl}
\mathit{len2} & \triangleq & \bigcirc(\mathit{LEN1}) \\
\mathit{len3} & \triangleq & \bigcirc(\mathit{len2}) \\
\mathit{lengt3} & \triangleq & \neg(\mathit{LEN1} \vee \mathit{len2} \vee \mathit{len3}) \\
\mathit{nocoli} & \triangleq & \Box(\neg(E\_GO \wedge (N\_GO \vee S\_GO))) \\
\mathit{icn} & \triangleq & \neg((\Box(N \wedge \neg N\_GO)) : \neg N) \\
\mathit{ics} & \triangleq & \neg((\Box(S \wedge \neg S\_GO)) : \neg S) \\
\mathit{ice} & \triangleq & \neg((\Box(E \wedge \neg E\_GO)) : \neg E) \\
\mathit{ic} & \triangleq & \Box(\mathit{icn} \wedge \mathit{ics} \wedge \mathit{ice}) \\
\mathit{asn4} & \triangleq & N \wedge \neg E \wedge \mathit{lengt3} \\
\mathit{ass4} & \triangleq & S \wedge \neg E \wedge \mathit{lengt3} \\
\mathit{ase4} & \triangleq & E \wedge \neg(N \vee E) \wedge \mathit{lengt3} \\
\mathit{ngoby4} & \triangleq & N\_GO \vee \bigcirc(N\_GO \vee \bigcirc(N\_GO \vee \bigcirc N\_GO)) \\
\mathit{sgoby4} & \triangleq & S\_GO \vee \bigcirc(S\_GO \vee \bigcirc(S\_GO \vee \bigcirc S\_GO)) \\
\mathit{egoby4} & \triangleq & E\_GO \vee \bigcirc(E\_GO \vee \bigcirc(E\_GO \vee \bigcirc E\_GO)) \\
\mathit{delay4} & \triangleq & \mathit{ic} \Rightarrow (\Box(\mathit{asn4} \Rightarrow \mathit{ngoby4}) \wedge \Box(\mathit{ass4} \Rightarrow \mathit{sgoby4}) \wedge \Box(\mathit{ase4} \Rightarrow \mathit{egoby4})) \\
\mathit{spec} & \triangleq & \mathit{nocoli} \wedge \mathit{delay4}
\end{array}
$$

Fig. 6 Specification of a traffic controller.

spec contains 89 operators, and the RTL model checker found that $\neg$*spec* is *true* for the bad design in 0.2 seconds when an additional 62 expression nodes are used. $\neg$*spec* becomes *false* for the good design in 1.1 seconds when an additional 322 expression nodes are used. The required time and space seem to be reasonable from a practical point of view.

5.3 Verification of a DMA Controller

As an example of verification of sequential machines with larger numbers of states, degign verification has been also done for a DMA controller [2]. The DMA controller is designed as a Moore machine with five input signals and 15 output signals. One design (a 'bad design'), which contains some design errors, has 392 states and its corresponding structure model has 12,544 states. The other (a 'good design') is a corrected version and has 272 states. Its corresponding structure model has 8704 states.

Figure 7 shows the assertions for the DMA controller. The assertion *as*1 states that if *MemReq* is always high (that is, *true*) then *ActivateComparator* is always low (that is, *false*), and if *MemReq* is always low then *MemGrant* is also always low. *as*2 states that it is always true that if *CpuReq* is high and *DmaReq* is low then *CpuReq* will eventually become low exactly two clocks after *MemFinished* is asserted. *as*3 states that if *TransferReq* becomes high then *DmaReq* will eventually be high the next time that *DeviceReady* is high. *as*4 states that if *DmaReq* is high and *ActivateComparator* and *ComparatorSet* will be high at some time, then either *DmaEnd* or *DmaCont* will be high the next time that DeviceReady is high. *as*5 states that *ActivateComparator* and *MemGrant* never become high at the same time. *as*6 states that *DmaType* never changes its value while *TransferReq* is high. *as*7 and *as*8 state that if *ComparatorSet* is high then the value of *DmaDone* never changes until *DmaEnd* or *DmaCont* becomes high. *as*9

$$
\begin{array}{rcl}
len2 & \triangleq & \bigcirc LEN1 \\
lengt2 & \triangleq & \neg(LEN1 \vee len2) \\
as1 & \triangleq & (\Box MemReq \Rightarrow \Box\neg ActivateComparator) \wedge (\Box\neg MemReq \Rightarrow \Box\neg MemGrant) \\
as2 & \triangleq & \Box((CpuReq \wedge \neg DmaReq) \Rightarrow \Diamond((MemFinished \wedge lengt2) \Rightarrow \bigcirc\bigcirc\neg CpuReq)) \\
as3 & \triangleq & \Box((\neg TransferReq \wedge \bigcirc TransferReq) \Rightarrow \bigcirc(\Diamond(DeviceReady \Rightarrow (LEN1 \vee \bigcirc DmaReq)))) \\
as4 & \triangleq & \Box(DmaReq \Rightarrow \Diamond(ActivateComparator \Rightarrow \Diamond(ComparatorSet \Rightarrow \\
 & & (LEN1 \vee \bigcirc(DmaEnd \vee DmaCont))))) \\
as5 & \triangleq & \neg\Diamond(ActivateComparator \wedge MemGrant) \\
as6 & \triangleq & \Box(\neg(TransferReq \wedge \bigcirc TransferReq \wedge (DmaType \oplus \bigcirc DmaType)) \vee LEN1) \\
as7 & \triangleq & \Box((DmaDone \wedge ComparatorSet) \Rightarrow (DmaEnd \vee \Box DmaDone \vee (\Box DmaDone : DmaEnd))) \\
as8 & \triangleq & \Box((\neg DmaDone \wedge ComparatorSet) \Rightarrow (DmaCont \vee \Box(\neg DmaDone) \vee \\
 & & (\Box\neg DmaDone : DmaCont))) \\
as9 & \triangleq & \Box(\neg(\neg ActivateComparator \wedge \bigcirc ActivateComparator \wedge ComparatorSet)) \\
asall & \triangleq & as1 \wedge as2 \wedge as3 \wedge as4 \wedge as5 \wedge as6 \wedge as7 \wedge as8 \wedge as9
\end{array}
$$

Fig. 7 Assertions for a DMA controller.

Table 1 Verification of the DMA controller.

		DMA Controller (5 input, 15 output)					
		Bad Design (392 states) Structure model 12544 states			Good Design (272 states) Structure model 8704 states		
Assertion	#Op.	Result	Time (sec)	#Node	Result	Time (sec)	#Node
*as*1	10	O.K.	0.8	4	O.K.	1.5	4
*as*2	13	O.K.	46.1	46	O.K.	29.5	46
*as*3	10	O.K.	16.0	13	O.K.	9.6	13
*as*4	9	O.K.	18.2	15	O.K.	11.4	15
*as*5	3	O.K.	14.0	1	O.K.	8.8	1
*as*6	8	O.K.	18.2	14	O.K.	11.9	14
*as*7	8	Fail	0.6	12	O.K.	9.9	12
*as*8	11	Fail	1.0	12	O.K.	9.9	12
*as*9	6	Fail	0.7	6	O.K.	10.3	6
asall	86	Fail	3.5	397	O.K.	70.6	915

states that *ComparatorSet* is never high just before *ActivateComparator* becomes high. *asall* is a logical conjunction of *as*1 to *as*9.

Table 1 shows the results of verification. The column '#Op.' shows the number of operators contained in assertions. The column '#Node' shows the number of expression nodes used in the verification process, excluding the nodes needed to store the given assertions themselves. Each assertion is checked independently. The RTL model checker finds that assertions *as*7, *as*8, *as*9, and *asall* are not satisfied by the bad design. The number of operators in each assertion varies from 3 to 86. The time needed to verify them varies from 0.6 to 70.6 seconds, which is acceptable from a practical point of view. In particular, design errors are detected much faster in general.

For the good design, it takes about 100 sec in total to check *as*1 to *as*9 independently, whereas it takes only 70 sec to verify *asall*. This is because that the checker handles *as*1 to *as*9 at the same time, and the results for a subformula of some assertion may be used in checking other assertions. On the other hand, the total number of nodes needed *as*1 to *as*9 is 123; whereas it is 915 for *asall*.

6. Conclusion

In investigating the finite and/or infinite behaviors of finite automata, we have shown four classes of regular temporal logic that are expressively equivalent to finite automata. We also discussed the verification problem for sequential machines by using regular temporal logic. As one approach to this problem, we described a model-checking method in which a given sequential machine is first converted to a corresponding structure model and the correctness of a given specification written in RTL is then checked. The computational complexity of the model-checking problem is non-elementary with respect to the length of a given formula, but we also proposed an efficient model-checking algorithm that runs in a time proportional to the size of a structure model. Furthermore, we implemented a model checker based on the proposed algorithm. Examples of verification of practically sized sequential machines show that the checker runs in reasonable time and space, and we believe that the model-checking approach based on RTL is very useful for formal logic design verification.

Acknowledgements

The authors would like to express their appreciation to Prof. E. M. Clarke of CMU who supported this research by providing various utilities, including design data for the traffic controller and the DMA controller. They also wish to thank Dr. N. Takagi, Dr. N. Ishiura, and members Prof. Yajima's Laboratory at Kyoto University for their valuable discussions.

References

1. BROWNE, M. C., CLARKE, E. M., DILL, D. L. and MISHRA, B. Automatic verification of sequential circuits using temporal logic. *IEEE Trans. Comput.*, C-35(12) (December 1986), 1035–1044.
2. CLARKE, E. M., BOSE, S., BROWNE, M. C. and GRUMBERG, O. The design and verification of finite state hardware controllers. Technical Report CMU-CS-87-145, Carnegie Mellon University (July 1987).
3. CLARKE, E. M. and EMERSON, E. A. Synthesis of synchronization skeletons for branching time temporal logic. In *Proc. Workshop on Logic of Programs*, Springer-Verlag (1981), 52–71.
4. CLARKE, E. M., EMERSON, E. A. and SISTLA, A. P. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. Technical Report CMU-CS-83-152, Carnegie Mellon University (1983).
5. FUJITA, M., TANAKA, H. and MOTOOKA, T. Verification with Prolog and temporal logic. In *Proc. 6th Int. Symp. Computer Hardware Description Language* (1983), 103–114.
6. HIRAISHI, H. Design verification of sequential machines based on a model checking algorithm of ε-free regular temporal logic. Technical Report CMU-CS-88-195, Carnegie Mellon University (1988).
7. HIRAISHI, H. Design verification of sequential machines based on ε-free regular temporal logic. In *Proc. 9th Int. Symp. Computer Hardware Description Language* (1989), 249–263.
8. HUGHES, G. E. and CRESSWELL, M. J. *An Introduction to Modal Logic*. Methouen, London (1977).
9. MOSZKOWSKI, B. Reasoning about digital circuits. Technical Report STAN-CS-83-790, Stanford Univ., (1983).
10. RESCHER, N. and URQUHART, A. *Temporal Logic*. Springer-Verlag, 1971.
11. UEHARA, T., SAITO, T., MARUYAMA, F. and KAWATO, N. DDL verifier and temporal logic. In *Proc. 6th Int. Symp. Computer Hardware Description Languages* (1983), 91–102.
12. WOLPER, P. Temporal logic can be more expressive. In *Proc. of 22nd Annual Symposium on Foundations of Computer Science* (1981), 340–348.
13. WOLPER, P., VARDI, M. Y. and SISTLA, A. P. Reasoning about infinite computation paths. In *Proc. 24th Symp. on the Foundations of Computer Science* (1983), 185–194.
14. HAMAGUCHI, K., HIRAISHI, H., YAJIMA, S. Temporal logic expressively equivalent to an ω regular set. Technical Report, COMP 88-8, *IEICE Japan* (May 1988).
15. HAMAGUCHI, K., HIRAISHI, H. and YAJIMA, S. Formal verification of sequential machines using an ω model-checking algorithm of ∞ regular temporal logic. Technical Report, COMP89-24, *IEICE Japan* (June 1989).
16. HIRAISHI, H., HAMAGUCHI, K., YAJIMA, S. Satisfiability algorithm for regular temporal logic. *Trans. IPS Japan*, **30**, 3 (March 1989), 366–374.
17. HIRAISHI, H., YAJIMA, S. RTL: Regular temporal logic expressively equivalent to a regular set. *Trans. IPS Japan*, **28**, 2 (Feb. 1987), 117–123.