Automatic Generation of Run-Time Test Oracles for Distributed Real-Time Systems *

Xin Wang, Ji Wang, and Zhi-Chang Qi

National Laboratory for Parallel and Distributed Processing
300 Lichen Rd., Changsha, 410073 China
xinwang76@yahoo.com.cn, ji.wang@263.net

Abstract. Distributed real-time systems are of one important type of real-time systems. They are usually characterized by both reactive and real-time factors and it has long been recognized that how to automatically check such systems' correctness at run time is still an unaddressed problem. As one of the main solutions, test oracle is a method usually used to check whether the system under test has behaved correctly on a particular execution. Test oracle is not only the indispensable stage of software testing, but also the weak link of the software testing research. In this paper, real-time specifications are adopted to describe the properties of distributed real-time systems and a real-time specification-based method for automatic run-time test oracles generating is proposed. The method proposed here is based on tableau construction theory of real-time model checking, automatically generates timed automata as test oracles, which can automatically check system behaviors' correctness from real-time specifications written in $MITL_{[0,d]}$.

1 Introduction

With the development of the network, distributed computing has become the mainstream of the computing technology undoubtedly. As a special kind of real-time systems, Distributed Real-Time Systems (DRTS) built on network environment have been applied widely in industry, military and commercial high-tech areas, especially in power engineering, aviation, real-time control systems, flexible manufacturing system, vision systems, etc [1]. Most of DRTS require high safety and strict time constraints, though the complexity of DRTS spans the gamut from very simple control of laboratory experiment, to very complicated projects such as the fighter avionic. So they are new challenge to the software testing methods during the software development.

Test oracle is a method for checking whether the system under test has behaved correctly on a particular execution [2]. It is the indispensable stage of software testing and also the weak link of the software testing research. The

* This work is supported by the National Natural Science Foundation of China under Grant No. 60233020 and No. 90104007; the National High Technology Development 863 Program of China under Grant No. 2001AA113202.

D. de Frutos-Escrig and M. Núñez (Eds.): FORTE 2004, LNCS 3235, pp. 199–212, 2004.

correctness of DRTS depends not only on the logical result of the computation, but also on the time when the results are produced [3]. Using run-time test oracles can not only check whether the system run is correct, but also improve the efficiency of software testing, set free testers from heavy work of checking system results.

DRTS are usually command-control systems, so their primary characteristics are event-triggered, complex event sequences, and real-time, precise time constraints. Temporal logic is the most important formal specification that describes distributed event-triggered, real-time systems' properties, and is used widely during software development. Using test oracles generated from temporal logic can reduce the costs of rewriting specifications greatly. The properties about time constraints of DRTS that described by real-time temporal logic are called real-time specifications. Test oracles generated from real-time specifications can automatically check if the run sequences of tested systems satisfy their specifications, if not, they can report corresponding error information.

The method proposed here is based on tableau construction theory of real-time model checking [4], automatically generates timed automata as test oracles, which can automatically check system behaviors' correctness, from real-time specifications written in $MITL_{[0,d]}$. The remainder of this paper is organized as follows. Section 2 describes relevant methods of automatically generating test oracles for reactive, real-time systems and their relative merits. Section 3 introduces the logic and timed automata that we use. Section 4 gives two approaches to acquire the traces of distributed real-time systems as input of test oracles. The work presented in core section 5 represents the method of automatically generating test oracles and case study. Section 6 concludes this paper and points out future work.

2 Related Work

Based on the generic tableau algorithm that generates specification automata for model checking, Dillon and Yu have proposed an automata-based method that can translate a temporal logic formula into a finite state machine as a test oracle [5, 6]. Once an execution sequence of a program is put into the finite state machine, the finite state machine can check if this execution sequence satisfies its specification. Proposition temporal logic can be applied only to describe the properties of reactive systems, but not real-time systems, because it doesn't support time quantifier. Therefore, this method can only be applied to reactive systems.

Method proposed by Geilen [7] also comes from the idea of model checking, which is very similar with the algorithm proposed by Kupferman and Vardi. This method has on-the-fly feature and has the same applicability as Dillon and Yu's.

Temporal assertions are proposed by Doron Drusinsky of Time-Rover Press [8]. The main idea is that temporal logic formulas are translated into some special kind of assertions (i.e. temporal assertions) as test oracles. Assertions

are inserted into a tested program manually in order that they can automatically check the program's correctness at run time. Assertion preprocessor must insert sub-assertions (assertions get by decomposing temporal assertions) into all related positions and maintain the relationship among them at run time. The costs of assertion maintenance will seriously influence the run of real-time systems and lend to the violation of time constraints, thus this method is more perfect for reactive systems or real-time simulators that amplify the absolute time than real-time systems.

John Håkansson has given an on-line test oracle generation method that is the only one that can check the correctness of real-time systems at run time [9]. Based on the rewriting rules of safety properties of real-time systems, this method discretizes the continuous time and automatically generates test oracles to monitor systems' behaviors externally. The test oracles fetch system data through sampling corresponding signals externally and compute the state-of-the-art pattern of systems at every end of the cycles, and in this way this method can support relatively strict time constraints. Because time is discretized, some system behaviors may be lost or some wrong behaviors may not be checked out if the time constraints in specifications are not integral multiple of read cycle.

3 Preliminaries

This section introduces the propaedeutics about logic and automata that we will use when automatically generating test oracles from real-time specifications.

3.1 Timed State Sequences

Let time domain be the non-negative real number set $R_{\geq 0}$. An interval I is a convex subset of $R_{\geq 0}$, which has the form $[a, b)$ where $a, b \in R_{\geq 0}$ and $a \leq b$. For a finite interval I, let $l(I)$ and $r(I)$ denote the left and right end of I respectively, and $|I|$ denote the length of I. Two intervals I, I' are adjacent if and only if $r(I) = l(I)$. We use $t + I$ to denote the interval $\{t + t' | t' \in I\}$. Let P be a finite proposition set and state s be a subset of P. If $s \subseteq P$ and $p \in s$ is a proposition in s, s is called p-state, denoting as $s \models p$.

Definition 1. *[10] A state sequence $\bar{s} = s_0 s_1 s_2 \ldots$ is an infinite sequence of states $s_i \subseteq P$. An interval sequence $\bar{I} = I_0 I_1 I_2 \ldots$ is an infinite sequence of timed interval such that*

[Initiality] I_0 is left-closed and $l(I_0) = 0$;
[Adjacency] for all $i \geq 0$, the intervals I_i and I_{i+1} are adjacent;
[progress] every time $t \in R_{\geq 0}$ belongs to some interval I_i.

A timed state sequence $\tau = (\bar{s}, \bar{I})$ is a pair that consists of a state sequence $\bar{s}$ and an interval sequence $\bar{I}$.

3.2 Real-Time Logic $MITL_{[0,d]}$

$MITL$ is a kind of linear temporal logic that is interpreted over timed state sequence [10]. In this paper, we only consider the real-time specifications written in $MITL_{[0,d]}(d{\in}R_{\geq 0})$, a restrict version of $MITL$.

Definition 2. *The formulas of $MITL_{[0,d]}$ can be inductively defined as follows:* $\phi ::= true|\ \ p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 U_{[0,d]}\phi_2$

The semantics of $MITL_{[0,d]}$ is presented in [10].

We also use dual operators "$\vee$" and "V" to define $\phi_1 \vee \phi_2 \triangleq \neg(\neg\phi_1 \wedge \neg\phi_2)$ and $\phi_1 V_{[0,d]}\phi_2 \triangleq \neg(\neg\phi_1 U_{[0,d]}\neg\phi_2)$. Similar with the "Always ('$\Box$')" and "Sometime ('$\Diamond$')" operators in LTL(Linear Temporal Logic), we use operators "Always in the interval [0,d] ('$\Box_{[0,d]}$')" and "Sometime in the interval [0,d] ('$\Diamond_{[0,d]}$')" to define $\Box_{[0,d]}\phi \triangleq falseV_{[0,d]}\phi$ and $\Diamond_{[0,d]}\phi \triangleq trueU_{[0,d]}\phi$.

Definition 3. *[10] Let ϕ be a formula of $MITL_{[0,d]}$. We call interval sequence $\bar{I}$ is ϕ-fine if for every sub-formula ψ of ϕ, every $k{\geq}0$ and every $t_1, t_2 \in \bar{I}(k)$, $\tau^{t_1} \models \psi$ if and only if $\tau^{t_2} \models \psi$. We call a timed state sequence $\tau = (\bar{s}, \bar{I})$ is ϕ-fine if the $\bar{I}$ in $(\bar{s}, \bar{I})$ is ϕ-fine.*

In [11], Lemma 4.11 is shown that the intervals of any timed state sequence can always be refined to be fine for any $MITL$ formula. It holds for the subset of $MITL$, $MITL_{[0,d]}$ too.

The tableau construction theory of real-time model checking requires that the truth value of the formulas interpreted over the time state sequence $(\bar{s}, \bar{I})$ can't change during a single interval of $\bar{I}$, i.e. the timed state sequence $(\bar{s}, \bar{I})$ must be ϕ-fine.

3.3 Timed Automata

The test oracles studied here are a variant of timed automata originally proposed by Alur and Dill [11] to serve as our test oracles. Timed automata use clocks whose values are positive real number to record the points when real-time specifications become true and when states change. Give a clock set C, clock interpretation function $v{\in}CInt(C)$ and clock setting function $CS \in Cset(C)$ are partial mappings from C to $R_{\geq 0}$. For some $d \in R_{\geq 0}$ and every $x \in dom(v)$, $v{+}d$ denotes the clock interpretation that assigns $v(x) + d$ to any clock x in the domain of v, and $CS(v)$ denotes that $CS(x)$(if defined)or $v(x)$ is assigned to $CS(v)(x)$. For a subset γ of C, we use $CS[\gamma := n]$ to denote the clock setting that maps all clocks in γ to n and keeps other clocks unchanged. Clock condition set $CCond(C)$ over clock set C is $\{x := t, x \geq d, 0 \leq t < d, x \leq t < d + x, y := t - x \mid x, y, t \in C\}$.

Definition 4. *Let P be a priority function. For a set $I = \{c_1, c_2, \ldots c_n\}$ ($i{\in}N$, $c_i{\in}CCond(C)$) and the natural number set N, we define:*

- *sorting function $o : I{\mapsto} < c_{i_1}, c_{i_2}, \ldots c_{i_n} >$ for $i_r{\in}\{1, 2, \ldots n\}$ such that for every $1{\leq}i_r{\leq}i_s{\leq}n$, $P(c_{i_r}){<}P(c_{i_s})$ holds or their is no comparability between c_{i_r} and c_{i_s};*

- *o's inverse function* $o^{-1} : \{c_1, c_2, \ldots c_n\} \mapsto I$ *such that* $o^{-1} \circ o(I) = I$.

Definition 5. *Let* P *be a set of proposition constraints. A finite-input timed automaton* $< S, S_0, C, Q, CC, OCC, Tran, s_e >$ *over* P *is defined as follow:*

- S *is a finite set of states;*
- C *is a finite set of clocks;*
- S_0 *is a finite set of initial extended states,* $(s_0, v_0) \in S_0 \subseteq S \times CInt(C)$*;*
- $Q : S \rightarrow 2^P$ *is a state labelling function which maps a state to a proposition constraint subset, i.e.* $Q(s) \subseteq 2^P$*;*
- $CC : S \rightarrow 2^{CCond(C)}$ *is also a state labelling function which maps a state to a clock condition subset;*
- $OCC : S \rightarrow o(2^{CCond(C)})$ *is an another kind of state labelling which maps a state to an ordered clock condition subset;*
- $Tran \subseteq S \times CSet(C) \times S$ *is a set of transitions, each of which labels with a clock setting function;*
- $S_e \in S$ *is a set of finite states.*

Definition 6. *A run of a finite-input timed automaton* $< S, S_0, C, Q, CC, OCC, Tran, S_e >$ *is a finite sequence* $\overset{v_0}{\underset{\gamma_0}{\rightarrow}}(s_0, t_0)\overset{v_1}{\underset{\gamma_1}{\rightarrow}}(s_1, t_1)\overset{v_2}{\underset{\gamma_2}{\rightarrow}}\cdots\overset{v_r}{\underset{\gamma_r}{\rightarrow}}(s_r, t_r)\overset{v_{r+1}}{\underset{\gamma_{r+1}}{\rightarrow}}\cdots\overset{v_n}{\underset{\gamma_n}{\rightarrow}}(s_n, t_n)$ *of states* $s_i \in S$ $(0 \leq i \leq n)$*, clock* $t \in C$*, clock sets* $\gamma_i \subseteq C$ *and a sequence of clock interpretation function* $\bar{v} = (v_0, v_1, \ldots, v_{n-1})$ *satisfying the following constraints:*

- $(s_0, v_0) \in S_0$*,* $s_n \in S_e$*;*
- *For every* $0 \leq k < n$*, there exists some* (s_k, CS_{k+1}, s_{k+1}) *such that for every* $c \in C$*,*

$$v_{k+1}(c) = \begin{cases} t_{k+1} & c = t \\ CS_{k+1}[\gamma_{k+1} := 0](c) & c \neq t \text{ and } CS(c) \text{ is defined} \\ v_k(c) & \text{otherwise} \end{cases};$$

- *For every* $0 \leq k < n$*, every* $c \in C$*,* $v_k(c)$ *satisfies* $CC_k(s_k)$ *and* $OCC_k(s_k)$*.*

Definition 7. *Let* $\tau = (\bar{s}, \bar{I})$ *be a timed state sequence and be* ϕ*-fine,* A *is a finite-input timed automaton* $< S, S_0, C, Q, CC, OCC, Tran, S_e >$*. We say that* $\tau = (\bar{s}, \bar{I})$ *can be accepted by* A*, if:*

- *for* $\varepsilon \rightarrow 0+$ *there is some* $r \in N$ *such that* $\overset{v_0}{\underset{C}{\rightarrow}}(s_0, r(I_0) - \varepsilon)\overset{v_1}{\underset{\gamma_1}{\rightarrow}}(s_1, r(I_1) - \varepsilon)\overset{v_2}{\underset{\gamma_2}{\rightarrow}}\cdots\overset{v_r}{\underset{\gamma_r}{\rightarrow}}(s_r, r(I_r) - \varepsilon)$ *is a run of* A*;*
- *for every* $k \in Z^+$*, we have* $\bar{s}_k \subseteq Q(u(k))$ *if* $u(k)$ *is a state of* A *which corresponds to the* k*-th position of the above run.*

In fact, the acceptable input of a timed automaton is not timed sequences, but the sequences of state-time pairs like $(s_0 s_1 \ldots s_n, t_0 t_1 \ldots t_n)$. Thereby, we must translate the timed state sequences into the above format. With this end of view, we introduce $\varepsilon(\varepsilon \rightarrow 0+)$ and use $r(I_i) - \varepsilon$ to replace the right end of

the interval I_i. In the field of real number, it is not holds to treat $r(I_i) - \varepsilon$ as the right end of I_i, because no matter how close ε approximates to zero, $\varepsilon/2 < \varepsilon$ always holds. But from the fact that the run of computers is step by step, i.e. the time is discrete, the interval $(r(I_i) - \varepsilon, r(I_i))$ doesn't exist if the value of ε is small enough, such as let ε equal to or smaller than the minimal click of the clock in a real-time system.

4 Traces Acquisition from DRTS

The behaviors that we want to acquire from DRTS are decided by the atomic formulas of real-time specifications. There are only two methods that can acquire the behaviors of DRTS and their occurring time; one is acquiring traces from outside of DRTS, another is acquiring traces from inside.

If all system behaviors involved by real-time specifications can be detected from outside of the DRTS, the first method can be used. Some kind of program module serves as monitor, detecting observable signals from outside of systems periodically and sending them to test oracles. The point when the output signals change is the left end of interval, and the point when the output signals change next time is the right end of interval; the interval is left-closed and right-open.

If real-time specifications involve internal states of systems, the second method must be used, i.e. some kind of assertions are inserted into proper positions (such as where the related states maybe change) of DRTS; assertions will send the information on which states change and when they change as soon as the true value of assertions change to test oracles. The point when the output signals change is the left end of interval, and the point when the output signals change next time is the right end of interval; the interval is left-closed and right-open.

Example: We consider the Carrier Sense, Multiple Access with Collision Detection protocol, or CSMA/CD for short [12, 13] which is widely used on LANs in the MAC sublayer. One safety property of CSMA/CD can be described as $\Box_{[0,\infty]}((Trans1 \wedge Trans2) \Rightarrow \diamond_{[0,\sigma]} coll)$ written in MITL that means whenever both senders begin transmitting, a collision is inevitably detected within σ. The value of varies with the network on which the protocol runs. For instance, for a 10 Mbps Ethernet with a typical worst case round trip propagation delay of $51.2\mu s$, we set σ to be $25.6\mu s$. We can get three system events from the above property: $Trans1$, $Trans2$ and $coll$.

Fig.1 is the oscillogram of $Trans1$, $Trans2$ and $coll$. While the three traces come to test oracles, one part of test oracles must arrange them into one trace according to the time they happen and assure the trace is ϕ-fine. Suppose that the vertical dash lines denote the right end of the refined intervals, we can get following timed state sequence from Fig. 1:

$(\{\neg Trans1, \neg Trans2, \neg coll\}[0, t_1))\ (\{\neg Trans1, \neg Trans2, \neg coll\}[t_1, t_2))$
$(\{Trans1, \neg Trans2, \neg coll\}[t_2, t_3))\ (\{Trans1, Trans2, \neg coll\}[t_3, t_4))$

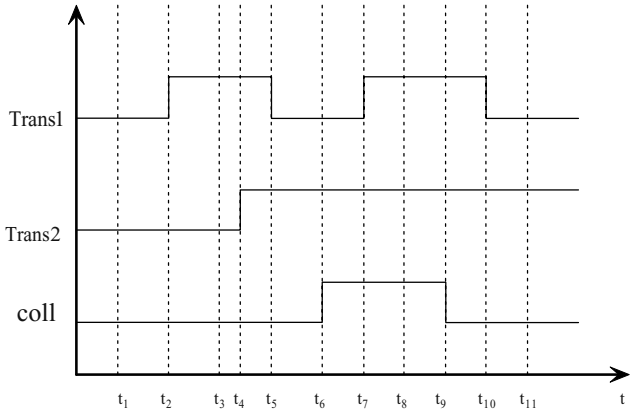

Fig. 1. The oscillogram of events $Trans1$, $Trans2$ and *coll*

$(\{Trans1, Trans2, \neg coll\}[t_4, t_5))$ $(\{\neg Trans1, Trans2, \neg coll\}[t_5, t_6))$
$(\{\neg Trans1, Trans2, coll\}[t_6, t_7))$ $(\{Trans1, Trans2, coll\}[t_7, t_8))$
$(\{Trans1, Trans2, coll\}[t_8, t_9))$ $(\{Trans1, Trans2, \neg coll\}[t_9, t_{10}))$
$(\{\neg Trans1, Trans2, \neg coll\}[t_{10}, t_{11}))\ldots$

The refinement procedure is done from the inner sub-formulas to outer sub-formulas, which give in [10]. In out example, as long as we get an event, we should refine the interval of *coll* first, then refine the interval of $Trans1 \wedge Trans2$ and the refined interval of *coll*. In this way, one interval may be cut into several smaller intervals which act as the actual input of the test oracles.

In this example, there must be three assertions corresponding to the three events. The assertions should be put on the neck of the statements that can cause the event status to change, such as *readin*, *assignment*, *output* statement and etc. In each of the assertions, there must be two variables to record the value before the current point and at the current point respectively. When the values of the two variables don't equal, the assertions should output the current values of the events and the time acquired from the system clock or specific external clock.

5 Automatic Generation of Run-Time Test Oracles

Generating run-time test oracles automatically is to construct timed automata based on real-time specifications written in $MITL_{[0,d]}$. Differing from specification automata in real-time model checking, the automata constructed for software testing need only accept finite state sequences. We say that a timed state sequence satisfies its specifications if this timed state sequence can reach the final states of automata constructed based on its specifications, i.e. it can

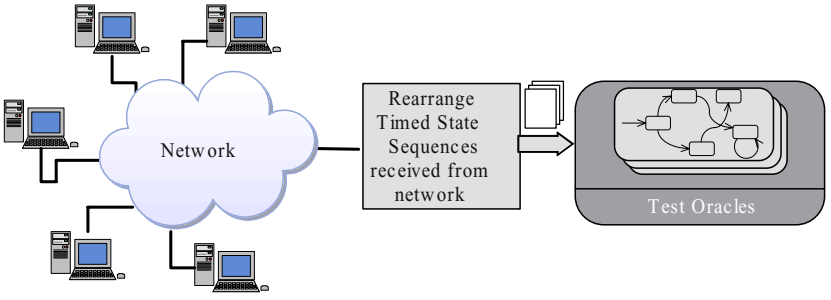

Fig. 2. The role of test oracle in software testing

pass through test oracles. The role of test oracles generated by our method in software testing is represented in Fig.2.

5.1 Rewrite Rules

The procedure that generates automata automatically from logic formulas often has to use rewrite rules. Based on rewrite rules, a logic formula can be equivalently decomposed tow parts: constraints that can be computed in current state and constraints that will be computed in subsequent states. Rewrite rules usually use "◯" operator to denote the constraints that will be computed in the subsequent states. In order to express the constraints that will be computed after some time point in $MITL_{[0,d]}$, we define similarly operator "◯" and extend the syntax of $MITL_{[0,d]}$.

Extended $MITL_{[0,d]}$ In order to rewrite the formulas of $MITL_{[0,d]}$, we use "◯" operator, clocks, clock conditions and clock interpretation function to extend the syntax and semantics of basic $MITL_{[0,d]}$.

Definition 8. *Based on the formulas of basic $MITL_{[0,d]}$, the formulas of $EMITL_{[0,d]}$ can be defined inductively as follows, where ϕ, ϕ_1 and ϕ_2 are formulas of $MITL_{[0,d]}$, x and t are clocks, $d \in R_{\geq 0}$:*
$\varphi ::= \phi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid CS.\varphi \mid x \geq d \mid 0 \leq t \leq d \mid x \leq t \leq x + d \mid x := t \mid y := t - x \mid \phi_1 U_{[0,d]_x} \phi_2 \mid \phi_1 V_{[0,d]_x} \phi_2 \mid \phi_1 V_{[x,d]} \phi_2 \mid \phi_1 V_{[y,d]_x} \phi_2 \mid \bigcirc \varphi$

Definition 9. *The satisfiability relationship $\tau \models_v \phi$ denotes that the timed state sequence τ satisfies ϕ in context of the clock interpretation v. The semantics of $EMITL_{[0,d]}$ is extended as follows:*

- $\tau \models_v \phi$ *iff* $\tau \models \phi$;
- $\tau \models_v \varphi_1 \vee \varphi_2$ *iff* $\tau \models_v \varphi_1$ *or* $\tau \models_v \varphi_2$;
- $\tau \models_v \varphi_1 \wedge \varphi_2$ *iff* $\tau \models_v \varphi_1$ *and* $\tau \models_v \varphi_2$;
- $\tau \models_v CS.\varphi$ *iff* $\tau \models_{CS(v)} \varphi$;
- $\tau \models_v x \geq d$ *iff* $v(x) \geq d$;

Table 1. Rules for decomposing a basic $MITL_{[0,d]}$ formula ϕ

$\phi =$	$\Phi \cup \{\phi\}$reduces to
$\phi_1 \Xi \phi_2$	$\Phi \cup \{\phi\}$
$\phi_1 \Delta \phi_2$	$\Phi \cup \{\phi_1, \phi_2\}$

- $\tau \models_v 0 \leq t \leq d$ *iff* $0 \leq v(t) \leq d$;
- $\tau \models_v x \leq t \leq x + d$ *iff* $v(x) \leq v(t) \leq v(x) + d$;
- $\tau \models_v x := t$ *iff* $v(x) := v(t)$;
- $\tau \models_v y := t - x$ *iff* $v(y) := v(t) - v(x)$;
- $\tau \models_v \phi_1 U_{[0,d]_x} \phi_2$ *iff there is some* $d_1 \in [v(x), v(x) + d]$, *such that* $\tau^{d_1} \models_{v+d_1} \phi_2$ *and for every* $d_2 \in [v(x), d_1]$, $\tau^{d_2} \models_{v+d_2} \phi_1$;
- $\tau \models_v \phi_1 V_{[x,d]} \phi_2$ *iff for every* $d_1 \in [v(x), d]$, $\tau^{d_1} \models_{v+d_1} \phi_2$, *or there is some* $d_2 \in [v(x), d_1]$, *such that* $\tau^{d_2} \models_{v+d_2} \phi_1$;
- $\tau \models_v \phi_1 V_{[0,d]_x} \phi_2$ *iff for every* $d_1 \in [v(x), v(x) + d]$, $\tau^{d_1} \models_{v+d_1} \phi_2$, *or there is some* $d_2 \in [v(x), d_1]$, *such that* $\tau^{d_2} \models_{v+d_2} \phi_1$;
- $\tau \models_v \phi_1 V_{[y,d]_x} \phi_2$ *iff for every* $d_1 \in [v(x) + v(y), v(x) + d]$, $\tau^{d_1} \models_{v+d_1} \phi_2$, *or there is some* $d_2 \in [v(x) + v(y), d_1]$, *such that* $\tau^{d_2} \models_{v+d_2} \phi_1$;
- $\tau \models_v \bigcirc\varphi$ *iff* $(\bar{s}^1, \bar{I}^1) \models_{v+|I_0|} \varphi$.

For a basic formula of $MITL_{[0,d]}$, we use operator"$\vee$"and "$V_{[0,d]}$" to push all negations inward until they reach atomic propositions. When mentioning the basic formulas of $MITL_{[0,d]}$ in rest of this paper, we mean the formulas of $MITL_{[0,d]}$ whose negations have been pushed inward.

Then we must define the clocks for the temporal operators in basic $MITL_{[0,d]}$ formulas. The decomposition rules is presented in Table 1 for a basic $MITL_{[0,d]}$ formula ϕ, $\Xi \in \{U_{[0,d]}, V_{[0,d]}\}$ and $\Delta \in \{\wedge, \vee\}$.

Definition 10. *Let ϕ be a basic $MITL_{[0,d]}$ formula. The set of the clocks of the temporal operators in ϕ can be defined recursively using following steps:*

1. *Let $\Phi_0 = \{\phi\}$. As long as one of the rule in Table 1 can be applied to any terms in Φ_n, then apply one to a term to obtain Φ_{n+1}. When no more rules can be applied, we get a set of sub-formulas of ϕ. In this set, each element either is a proposition, or has a temporal operator as outermost operator.*
2. *For each element whose outermost operator is a temporal operator, we define a clock x for the outermost temporal operator and an null ancestor set $AncSet_x$ for every clock.*
3. *For each element φ in Φ_{n+1}, let $\Phi_0 = \{\varphi_1, \varphi_2\}$ and repeat step 1 if its form is $\varphi_1 \Xi \varphi_2$. As a result, we will get a set similar to the one in step 1. For each element which has an outermost temporal operator, the clock of its outermost temporal operator is y $|_x$ if the clock of the operator Ξ is x, i.e. a clock y whose value is relative to clock x, the ancestor set of the clock $AncSet_y$ equals $\{x\} \cup AncSet_x$.*
4. *Repeat step 3 until all elements of all sets generated by these steps are propositions.*

Rewrite rules The aim of using rewrite rules is to transform a basic $MITL_{[0,d]}$ formula ϕ into normal form: $\phi = \bigvee_i CS_i.(cc_i \wedge \phi_i \wedge \bigcirc\varphi_i)$, where CS_i denotes that current clocks are set by clock setting function, cc_i are conjunctions and "ordered conjunctions" of clock conditions, ϕ_i are conjunctions of atomic propositions, and φ_i are subsequent formulas, i.e. conjunctions of $EMITL_{[0,d]}$.

While constructing timed automaton, we use triple $< CS, Now, Next >$ to denote $\phi = \bigvee_i CS_i.(cc_i \wedge \phi_i \wedge \bigcirc\varphi_i)$, where CS denotes the labels of transitions, Now includes clock conditions and propositional ϕ_i that must be satisfied in current state, and $Next$ includes subsequent formulas that will be satisfied. The data structure of Now is the same as the labels of the states of the timed automata, i.e. $Now =< D, E >$, where D is a set of clock conditions and propositions, E is an order set of clock conditions. Thereby, we must define the priorities for the clock conditions.

Definition 11. *A priority function* $P : CCond(C) \times CCond(C) \rightarrow \{>\}$ *is defined as follows:*

- *if* $x \in C$ *and* $x := t, x \geq d \in CCond(C)$, $P(x := t) > P(x \geq d)$;
- *if* $x, y \in C$ *and* $x := t, y := t - x \in CCond(C)$, $P(y := t - x) > P(x := t)$;

Definition 12. *Based on the sorting function* o*, its inverse function* o^{-1} *from Definition 4, we define operations* $\oplus : Now \times Now \mapsto Now$ *and* $\ominus : Now \times Now \mapsto Now$ *as follows, where* $Prop \in 2^P$ *is a proposition constraint subset,* $CCS, \{e_1, e_2, \ldots, e_n\} \subseteq CCond(C)$*:*

- $Now \oplus < Prop \cup CCS, o(\{e_1, e_2, \ldots, e_n\}) >=$
  $< Now.D \cup Prop \cup CCS, o(o^{-1}(Now.E) \cup \{e_1, e_2, \ldots, e_n\}) >$;
- $Now \ominus < Prop \cup CCS, o(\{e_1, e_2, \ldots, e_n\}) >=$
  $< Now.D \setminus (Prop \cup CCS), o(o^{-1}(Now.E) \setminus \{e_1, e_2, \ldots, e_n\}) >$.

Definition 13. *For the clock* x *of a temporal operator, we define an assignment set* $AssSet_x = \{y := t, y := t - z \mid y, z \in AncSet_x \text{ and } z \in AncSet_y\}$.

5.2 Algorithm of Constructing Timed Automaton

Definition 14. *If* Ψ *is a set of* $MITL_{[0,d]}$ *formulas, the normal form* $NF(\Psi)$ *of* Ψ *is computed with the following procedure. Let* $P_0 = \{< \varnothing, < \{\phi\}, \varnothing >, \varnothing >\}$. *As long as one of the rewrite rules in Table 2* [1] *can be applied to any of the terms in* $Now.D$ *of* P_n*, then apply one to a term to obtain* P_{n+1}*. The normal form* $NF(\Psi)$ *is obtained from* P_n *when no more rules can be applied.*

Lemma 1. *The equivalences between the basic* $MITL_{[0,d]}$ *formulas and their rewritten forms hold and the value of clocks is the time when the truth values of the sub-formulas within their domains are true until now.*

[1] The clocks mentioned in this table is the clocks binding with its temporal operators defined in Definition 10.

Table 2. Rewrite rules

$\phi =$	$Conditions$	$\Phi \cup \{< CS, Now\oplus < \{\phi\}, \varnothing >, Next >\}$reduces to
$true$		$\Phi \cup \{< CS, Now, Next >\}$
$false$		Φ
$\phi_1 \vee \phi_2$		$\Phi \cup \{< CS, Now\oplus < \{\phi_1\}, \varnothing >, Next >,$ $< CS, Now\oplus < \{\phi_2\}, \varnothing >, Next >\}$
$\phi_1 \wedge \phi_2$		$\Phi \cup \{< CS, Now\oplus < \{\phi_1, \phi_2\}, \varnothing >, Next >\}$
$\phi_1 V_{[0,d]} \phi_2$		$\Phi \cup \{< CS[x := 0], Now\oplus < \{\phi_1 V_{[x,d]} \phi_2\} >\}, \varnothing >, Next >$
$\phi_1 V_{[x,d]} \phi_2$		$\Phi \cup \{< CS, Now\oplus < \{\phi_1, \phi_2\}, < x := t >>, Next >,$ $< CS, Now\oplus < \{\phi_2\}, < x := t >>, Next \cup \{\phi_1 V_{[x,d]} \phi_2\} >,$ $< CS, Now\oplus < \{\phi_2\}, < x := t, x \geq d >>, Next >\}$
$\phi_1 V_{[0,d]_x} \phi_2$		$\Phi \cup \{< CS[y := 0], Now\oplus < \{\phi_1 V_{[y,d]_x} \phi_2\}, \varnothing >, Next >>\}$
$\phi_1 V_{[y,d]_x} \phi_2$		$\Phi \cup \{< CS, Now\oplus < \{\phi_1, \phi_2\}, < y := t - x >>, Next >,$ $< CS, Now\ominus < \varnothing, o(AssSet_y) > \oplus < \{\phi_2\},$ $< y := t - x >>, Next \cup \{\phi_1 V_{[y,d]_x} \phi\} >, < CS, Now\oplus$ $< \{\phi_2\}, < y := t - x, y \geq d >>, Next >\}$
$\phi_1 U_{[0,d]} \phi_2$	$\phi_2 \in Now.D$	$\Phi \cup \{< CS, Now, Next >\}$
	$\phi_2 \notin Now.D$	$\Phi \cup \{< CS, Now\oplus < \{\phi_2\}, < x := t >>, Next >,$ $< CS, Now\oplus < \{\phi_1, 0 \leq t < d\}, < x := t >>,$ $Next \cup \{\phi_1 U_{[0,d]} \phi_2\} >\}$
$\phi_1 U_{[0,d]_x} \phi_2$	$\phi_2 \in Now.D$	$\Phi \cup \{< CS, Now, Next >\}$
	$\phi_2 \notin Now.D$	$\Phi \cup \{< CS, Now\oplus < \{\phi_2\}, < y := t - x >>, Next >,$ $< CS, Now\ominus < \varnothing, o(AssSet_y) > \oplus < \{\phi_1, x \leq t < x + d\},$ $< y := t - x >>, Next \cup \{\phi_1 U_{[0,d]} \phi_2\} >\}$

Definition 15. *Let ϕ be a basic $MITL_{[0,d]}$ formula, P be the set of propositions that occur in ϕ. The tableau automaton A_ϕ is the finite-input timed automaton $< S, S_0, C, Q, CC, OCC, Tran, S_e >$ over 2^P, where*

- *C is the set of all clocks computed by Definition 10;*
- *S, S_0, $Tran$ and S_e can be computed by the procedure depicted in Fig. 3.*
- $Q(s) = \{\phi \in 2^P \mid \forall_{p \in P} p \in Now.D \Rightarrow p \in \phi, \neg p \in Now.D \Rightarrow p \notin \phi\}$
- $CC(s) = \{\eta \in CCond(C) \mid \eta \in Now.D\}$
- $OCC(s) = o\{\eta \in CCond(C) \mid \eta \in Now.E\}$

The algorithm presented above is correct and it can be stated by the following theorem.

Theorem 1. *Let ϕ be an $MITL_{[0,d]}$ formula and timed automaton A_ϕ be the corresponding tableau automaton, then for every timed state sequence τ, A_ϕ accepts τ iff $\tau \models \phi$.*

Case Study We consider the CSMA/CD protocol in section 4 again. While testing, we need presuppose the max testing time, such as $10^{10}\mu s$. So above property can be rewritten as $\Box_{[0,10^{10}]}((Trans1 \wedge Trans2) \rightarrow \Diamond_{[0,60]} coll)$ in $MITL_{[0,d]}$. Using our method, we can get the tableau automaton for the above property, which consists of 18 states and 70 transitions. But the state and transition amount of the resulting automaton can be decreased to 12 and 35 ulteriorly, if we treat the sub-formula $Trans1 \wedge Trans2$ from the example of CSMA/CD protocol as an atomic formula. And what we must modify is the component of rearrangement. For clarity we only draw the smaller automaton as Fig. 4, in which the round-corner rectangles denote states, the arrowed lines denote transitions. Every state

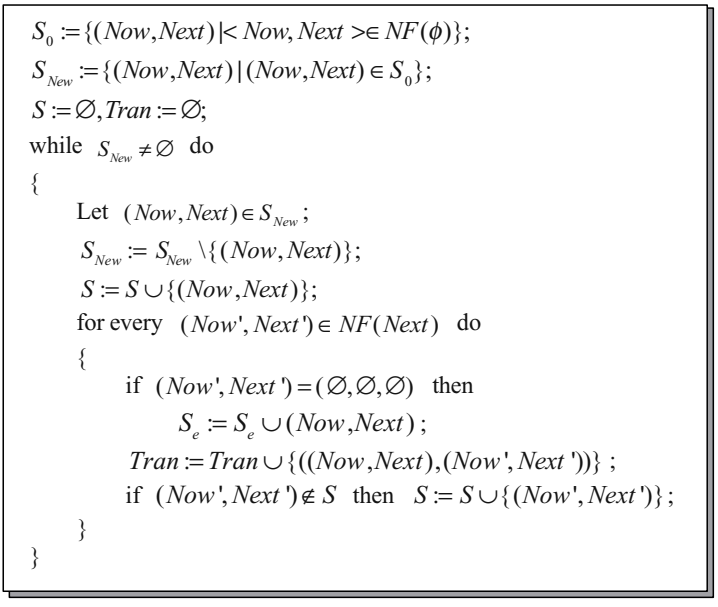

Fig. 3. Algorithm for constructing the states and transitions of the tableau automaton

is divided into two parts, the upper of which denotes $Now.D$, and the lower of which denotes $Now.E$. In order to simplifying the representation, only the formulas in Now have been depicted. The initial extended states are represented by an arrow not originating form any state, and the finite states are denoted by a filled circle inside a slightly larger unfilled circle put in the lower part of the states.

6 Conclusions and Future Work

This paper presents a new method that can automatically generate run-time test oracles for DRTS. Test oracles can check whether a distributed real-time system's traces are correct, based on real-time specifications written in $MITL_{[0,d]}$ formulas. If the test oracles can accept the timed event sequences, we can say that the runs are correct, i.e. the system runs correctly. Otherwise, if all current states can't accept any subsequence timed events that violate state or time constraints, or the tail of some trace does not arrive at final state, we say the system has some errors or some real-time specifications errors exist.

Compared with [9], whose computation is done at each end of cycles, our method does computation only if necessary (i.e. when system states change) so as to save precious computing time. Assertions inserted into the different parts of DRTS not only can obtain the inner events of DRTS, send out the values of states only when they change, but also can get the precise occurrence time of events.

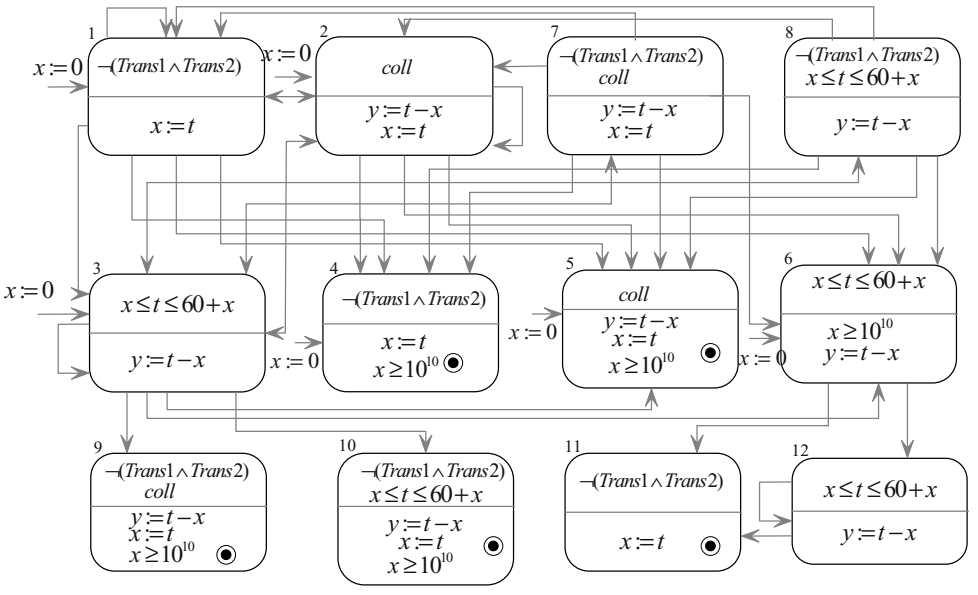

Fig. 4. The Optimized tableau automata of $\Box_{[0,10^{10}]}((Trans1 \wedge Trans2) \rightarrow \diamond_{[0,60]} coll)$

The ongoing works include:

1. The method of automatic test oracles generating needs to be optimized in order to reduce the complexity of tableau automata and checking costs. This can be done by optimizing rewrite rules and timed automata.
2. The real-time specifications used in this paper is the restrict version of MITL, $MITL_{[0,d]}$, that limits the ability of logic expression for properties of DRTS. For example, $\Box_{[0,100]}(\diamond_{[10,20]}p)$ can not be transformed into a test oracle. So how to extend the detection capability of test oracles is the main problem of future work.
3. DRTS are running on networked environments, so we must consider the delay of network transmission while acquiring the system traces. When the events and their timestamps reach the test oracle, there must be a special part to arrange them so that the whole timed sequence is reordered by time and are ϕ-fine.
4. We must evaluate the computing costs of assertions inserted into the DRTS so as to assure that assertions will not influence the normal running of DRTS. The computing costs of assertions are decided by the amount and the data structures of assertions. And the next works must include it.
5. Finally, more experiments in real environments are required.

References

[1] (http://www.ii.uj.edu.pl/progroz/dishard/home.html) 199

[2] Baresi, L., Young, M.: Test Oracles. Technical Report, CIS-TR01-02, Dept. of Computer and Information Science, Univ. of Oregon (Aug.2001) 199

[3] J. A.Stankovic: Misconceptions about real-time computing - a serious problem for next generation systems, IEEE Computer (1998) 200

[4] Geilen, M.: Formal Techniques for Verification of Complex Real-Time Systems. PhD thesis, Eindhoven University of Technology (2002) 200

[5] Dillon, L., Yu, Q.: Oracles for checking temporal properties of concurrent systems. In: Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering. (1994) 200

[6] Dillon, L., Ramakrishna, Y.: Generating oracles from your favorite temporal logic specifications. In: Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering. (1996) 200

[7] Geilen, M.: On the construction of monitors for temporal logic properties. In: K. Havelund and G. Rosu, editors, Proceedings of RV'01 - FirstWorkshop on Runtime Verification. Satellite Workshop of CAV'01, Electronic Notes in Theoretical Computer Science 55(2), Amsterdam, 2001. Elsevier Science, Paris, France (2001) 200

[8] Drusinsky, D.: The temporal rover and the atg rover. In: SPIN Model Checking and Software Verification, Proc, 7th SPIN Workshop, 1885 of Springer-Verlag Lecture Notes in Computer Science, Springer Verlag, Stanford, California (2000) 200

[9] Håkansson, J.: Automated generation of test scripts from temporal logic specification. Master's thesis, Uppsala University (2000) 201, 210

[10] R. Alur, T. F., Henzinger, T.: The benefits of relaxing punctuality. Journal of the ACM **43** (January 1996) 116–146 201, 202, 205

[11] Alur, R.: Techniques of Automatic Verification of Real-Time Systems. PhD thesis, Stanford University (1991) 202

[12] IEEE: ANSI/IEEE 802.3, ISO/DIS 8802/3. IEEE Computer Society Press (1985) 204

[13] Tanenbaum, A. S.: Computer Networks. Prentice-Hall, Englewood Cliffs, second edition (1989) 204