Verification of Concurrent Systems*

L.Su†
Computing Laboratory, University of Kent
ls68@kent.ac.uk

1 September 2003

Abstract

In recent years, people are interested in real time and distributed systems. A vital characteristic of such systems is that they are usually concurrent. There are various techniques that support formal modeling of concurrency. One is Process Algebra; other techniques include Temporal Logic and Timed Automata. Moreover, one of the most successful verification techniques is called model checking, which is a technique for verifying finite state concurrent systems and tracing errors. We investigate the deadlock detection, especially timelock detection in UPPAAL. We also give a formal definition of Timed Automata and its semantics, following a classification of deadlocks, and two Progress Requirements. We then provide an algorithm and implement the algorithm to detect zeno-timelock. At the end the software is tested for its input and cycle detector and we give a case study of CSMA/CD. It is specified in UPPAAL, and then we use the software to verify it.

1 Introduction

In this MSc project research has been done in the area of Formal Modeling of Concurrency, such as Process Algebra, Temporal Logics, Timed Automata, as well as the verification technique that is based on those specifications. Finally, a specification and automatic verification tool suit UPPAAL is investigated for its ability to check timelock freedom properties.

1.1 Problem Area

Concurrent and Distributed techniques and technologies are widely applied to the modem computer and communication systems. At the same time, such

*A dissertation submitted for the degree of MSc in Distributed Systems and Networks, Computing Laboratory, University of Kent

†This project is supervised by Howard Bowman. We would like to thank Zhe Li who gave a lot of valuable suggestions on the some proofs and cycle detection algorithm. As well as Jenni Brown and Paul Murphy helped correcting language errors in the text.

systems exhibit a great number of requirements, such as multimedia, quality of service, object orientation and especially real-time problems. These problems are of great complexity in the concept of concurrent and distributed systems, however, the formal methods approach is supposed to be an efficient and straightforward solution towards those requirements.

There are various techniques that support formal modeling of concurrency. One is Process Algebra, which is an algebraic approach to specify concurrent processes and formalize statements about them. The main algebraic approaches to concurrency are CCS, CSP, ACP and LOTOS.

Other techniques include Temporal Logic and Timed Automata. Temporal Logic is an extension of first order logic that is supposed to be the most appropriate way to express global requirements of distributed and concurrent systems, which enables propositions expressed over the passage of time. Timed automata are finite state machines with real valued clocks, used to model continuous time. A whole system is modeled as the parallel composition of its component automata.

One of the most successful verification techniques is called model checking, which is a technique for verifying finite state concurrent systems and tracing errors. Model checking can be performed automatically and always terminates with a positive or negative answer if sufficient memory resources are available. The model checker that we investigate is the UPPAAL model checker 'verifyta'. This paper presents a lot of work on deadlock detection, especially timelock detection in UPPAAL.

1.2 Motivation

The term timelock that is used in this work refers to a subset of deadlock in which time is blocked from advancing. One of the motivations to study timelock in the timed automata is that it has different properties from the deadlocks in traditional untimed automata. Occurrence of a timelock in the local system can seriously affect other systems, even if they are completely independent from the local system. Their time is blocked because of global time synchronization. Whereas the deadlocks that occur in traditional untimed automata only have a local effect. In addition, it is critical to detect, distinguish and eliminate timelocks at the system design stage. Currently a lot of verification tools are able to detect traditional deadlock and a subset of timelock. However, there is still no tool support for thorough timelock detection. Therefore, it is an interesting and usful topic to research.

In Section 2 - BACKGROUNDS we give a brief introduction of Concurrent System, Specification and Verification of Concurrency and tools for the Specification and Verification of Concurrent Systems especially UPPAAL.

In Section 3 - TIMED AUOTOMATA AND DEADLOCKS we give a formal definition of Timed Automata and its semantics, following a classification of deadlocks, and two Progress Requirements. These are basis to define the three disjoined types of deadlocks: the Time-action-lock, the Pure Actionlock and the Zeno-timelock. We also try to give proofs of these definitions mathematically.

While giving these definitions, we also provide examples that are modeled in UPPAAL, simulate and verify these examples using UPPAAL.

In Section 4 - DEADLOCK DETECTIONS we try to provide an algorithm and implement the algorithm to detect zeno-timelock. Strategies to handle deadlocks are stated and chosen carefully. Then a notion of Strongly Non-zeno is expressed and its ability to check zeno-timelock is used to implement a piece of software. The later part of this chapter explains the data structure, the input/output, the cycle detection and the algorithm to check the conditions of Strongly Non-zeno in details. At the end, the software is tested for its input and cycle detector. To compare with our algorithm, a method of deadlock detection in simulation stage is briefly explained.

In Section 5 - CASE STUDY: CSMA/CD we give an introduction of CSMA/CD and its specification in UPPAAL. Then verify the models by the software that provide in the pervious Chapter.

In Section 6 - EVALUATIONS conclusions are made. The possible future works and a conformance to original plan are explained in the end.

2 Backgrounds

2.1 Introduction of Concurrent System

In recent years, people are interested in real time and distributed systems. A vital characteristic of such systems is that they are usually concurrent i.e. they consist of many components executing in parallel. [1]

As it is stated in [9], there are two reasons why concurrency may be used:

1. Either for increased performance, e.g. an implementation on a parallel processor is often chosen since this will allow faster execution than on a sequential processor.

2. Or because concurrency allows a natural representation of a problem. Many problems are inherently parallel, i.e. care must be taken not to introduce any unnecessary ordering constraints, such that actions must be performed either sequentially or simultaneously.

Unfortunately, concurrent systems are very difficult to design correctly. For example, values, which one process is expecting to receive, are indeed the values that the other processes will send. One problem should avoid is deadlock in which one process are waiting to communicate while the other have terminated, consequently none of communications be done since their mismatch. And this is only one situation of deadlock.

That is crucial to provide techniques of formal methods to ensure deadlock freedom. The next section we will discuss the specification and verification of concurrent systems, which give a proof of deadlock freedom of such systems.

2.2 the Specification and Verification of Concurrent Systems

Follows the discussion in [9], a specification expresses the functional and non-functional requirements of a system which could be a piece of software or protocol. It states what the target system is intended to achieve and the attributes or features the system must process.

It is also mentioned in [9] that specifications are used for validation and verification purposes. It is validated to assure that the specification represents the intended use of the system and that the system as specified will meet its objectives. Then the specification forms a basis for developing a design of system that can be verified against the validated specification.

In resent years, model checking is a useful technique for verifying temporal properties of finite state concurrent systems, i.e. systems that the variables range over finite domains.

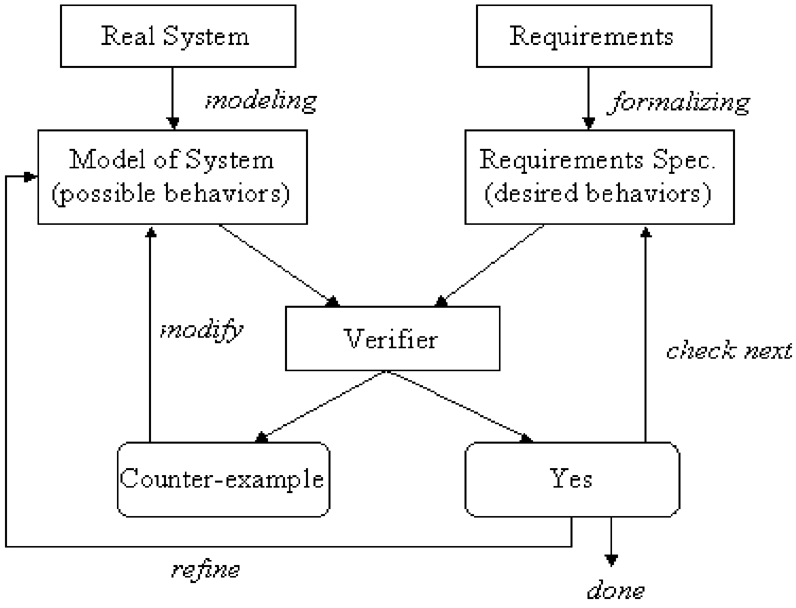

Figure 1: Model Checking

In [1], the methodology of traditional model checking is presented schematically in Figure 1.

In addition, model checking has been extended to real time systems, with time considered to be a dense linear order. A timed extension of finite automata through addition of a finite set of real valued clock variables is called timed automata. In [10], the meaning of denseness of time domain is express as that the state space of timed automata is infinite. Therefore, constructing a finite state labeled transition system, which represents all the behaviors of the timed automaton, is the basis of verification of timed automata. The corresponding model checking turns to work by searching over the entire state space of the transition system of timed automata. The model checker tests if a property, which is expressed by timed extension of CTL (Computation Tree logic), is satisfied or not.

2.3 Tools for the Specification and Verification of Concurrent Systems

A number of tools for automatic verification of real time and concurrent systems have been introduced and they are mature enough for applications both on academic and industrial developments.

One such tool is UPPAAL, which is a tool suit for the specification and automatic verification of real time systems. In the next section, we will discuss the general features of UPPAAL. In Section 3, we will look into detailed formal definition and semantics of timed automata that UPPAAL used for system specification language.

2.4 Introduction of UPPAAL

UPPAAL was developed in collaboration between BRICS (Basic Research in Computer Science - is a basic research center funded by the Danish government at Aarhus and Aalborg University) at Aalborg University and Department of Computing Systems at Uppsala University.

2.4.1 Overview

The tool provides support for validation via symbolic simulation and verification via automatic model checking. An overview of the tool is shown in Figure 2, which briefly describes the main components and features of UPPAAL.

UPPAAL2k 3.2.13 is released 11 December 2002. In addition to the version 3.2, the third beta release of version 3.4 has been put forward. The current version of UPPAAL has the following main features. [5]

1. A graphical interface based on Autograph allowing graphical specification (.atg) of systems.
2. A compiler (atg2ta) transforming graphical descriptions into a textual programming format (.ta).
3. A simulator (simta) provides a graphical visualization and recording of the possible dynamic behaviors of a system description, this allows inexpensive fault detection in early modeling stage.
4. A model checker (verifyta) for automatic verification of safety and bounded liveness properties by on-the-fly reach ability analysis.

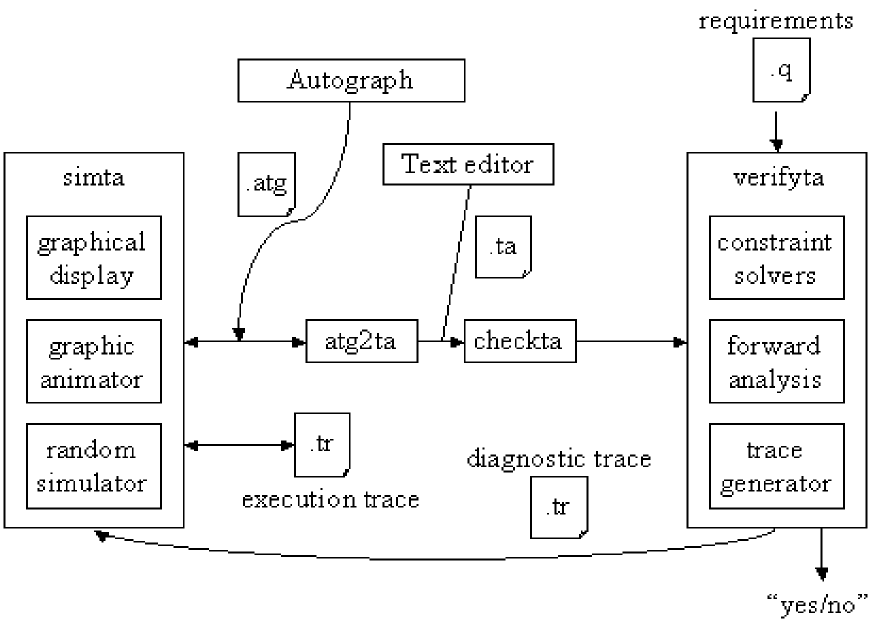

Figure 2: Overview of UPPAAL [12]

5. Generation of (shortest) diagnostic traces in case verification of a particular real time system fails. The diagnostic traces may be graphically visualized using the simulator.

2.4.2 UPPAAL Model

It is stated in [11] that UPPAAL is based on timed automata with finite state machine with real value clocks and integer values. The clocks are the way to handle time in UPPAAL. Time is continuous and the clocks measure time progress. [18] It is implemented as regions and the states are thus symbolic, which means that at a state we do not have any concrete value for the time, but rather differences. The value of a clock can be tested or reset. Time will progress globally at the same pace for the whole system.

It is also said in [11, 12] that a system modeled in UPPAAL is composed of concurrent processes, each of them modeled as an extended timed automaton. Where the automaton contains of locations and transitions. Transitions are used to change location. A Guard and a synchronization, which is attached to transitions, is used to control when to fire a transition. A guard is a condition on the variables and the clocks saying when the transition is enabled. The synchronization mechanism in UPPAAL is a CCS style two-way hand-shaking synchronization: given a synchronization channel a. Two processes take a transition at the same time, one will have a $a!$ and the other a $a?$. When taking a transition, it is possible to have actions, which are assignment of variables or reset of clocks.

3 Timed Automata and Deadlocks

In this part of the paper we give a formal definition of timed automata following the definition in [2, 3]. At the same time, the UPPAAL model and its semantics will be reviewed. Since some parts of the UPPAAL model are the same or very similar to the timed automata used in this paper, we will only state the differences between them.

3.1 Timed Automata

$\bar{A} = \{a?, a! | a \in A\}$ is a set of uncompleted actions (or half actions, as they are called in [3]). A is a set of completed (or basic [2] or internal [3]) actions. Two uncompleted actions, $a?$ and $a!$ can synchronize and generate a completed action a, by which UPPAAL performs a CCS style point-to-point communication. $A^+ = \bar{A} \cup A$ is the set of *all* actions. $\bar{a} = a$ if $a \in A$, $\bar{a}? = a!$ and $\bar{a}! = a?$ is a complementation notation over elements of A^+.

Let $\mathbb{R}^+$ denote the positive reals without zero and $\mathbb{R}^{+0} = \mathbb{R}^+ \cup \{0\}$ denotes the non-negative reals. C is the power set of clock variables, which range over $\mathbb{R}^+$.

A time automaton has the form (L, L_0, T, P). [2] In which,

L defines a finite set of locations, which are denoted as nodes in UPPAAL timed automata diagrams.

l_0 is a start location, which is denoted as a node with a smaller circle inside.

T is a transition relation, elements of which have the form (l_1, g, a, r, l_2), in which $l_1, l_2 \in L$, $g \in CC$ is a guard, CC is a set of clock constraints, r is a reset set and a is an action. When guard g holds, action a can be performed and if it is performed all clocks in set r will be reset.

P $:L \rightarrow CC$ is a function, which associates with every location a progress condition. It is called the invariant in UPPAAL, that is to say an automaton can only stay in a state when function P holds.

Semantically, timed automata are interpreted over transition systems which are triples, $(S, s_0, \rightarrow)$, where,

S is a set of states. A state has the form (l, v), in which $l \in L$ is a location, and v is a clock valuation;

$s_0 \in S$ is a start state;

$\rightarrow \subseteq S \times Y \times S$ is a transition relation, where $Y = A^{+} \cup \mathbb{R}^{+}$, i.e. there are two types of transitions, action transitions, e.g. (s_1, a, s_2), where $a \in A^{+}$, written: $s_1 \xrightarrow{a} s_2$ and time passing transitions, e.g. (s_1, x, s_2), where $x \in \mathbb{R}^{+}$, denoting the passage of x time units, written: $s_1 \xrightarrow{x} s_2$.

Transition $\rightarrow$ is interpreted over action or time passing transition systems according to these inference rules:

$$\frac{l \xrightarrow{g,a,r} l' \;\; g(u)}{(l,u) \xrightarrow{a} (l', r(u))} \text{ respectively } \frac{\forall d' \leq d \cdot P(l)(u+d')}{(l,u) \xrightarrow{d} (l, u+d)} \text{ where,}$$

a clock valuation $u \in C \rightarrow \mathbb{R}^{+0}$ is a mapping from clock variables C to $\mathbb{R}^{+0}$. d is a delay, $u + d$ is the clock valuation such that $(u + d)(x) = u(x) + d$ for all $x \in C$. r is a reset set and $r(u)$ denotes the clock valuation u' such that $u'(x) = 0$ when $x \in r$ and $u'(x) = u(x)$ otherwise.

The first inference rule describes the semantics of an action transition. Suppose in a state (l, u), from location l in a timed automaton, there is an action transition a with guard g and clock reset r, when the guard $g(u)$ holds it may do an action a instantaneously and go to the next state $(l', r(u))$ with the clocks in set r being reset.

A time passing transition can be interpreted by the second inference rule. If every delay equal or less than d is allowed by the progress condition associated with location l, the timed automaton can allow time to pass by d units and go to the next state $(l, u + d)$.

A significant different between UPPAAL semantics, which can be found in [5], and the semantics of timed automata in this paper is an adjusted constraint system in addition to the ordinary constraint system that the clock and variable assignments must satisfy to be able to take a transition. This adjusted constraint system ensure that it is in addition possible to reach the next location by taking a transition.

In order to explain this difference we use an automaton, where a location has a false invariant, as an example.

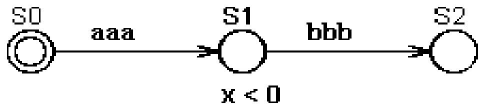

Figure 3: an example of a false invariant

As shown in Figure 3, the location S1 has an invariant of $x < 0$. Because clock x is a non-negative real, $x < 0$ is *false*. The reason why we use $x < 0$ rather than *false* here is that UPPAAL - 3.3.35 dose not allow *'false'* in the syntax. (The current beta version, UPPAAL - 3.3.36 includes the keywords *'true', 'false', 'and', 'or', 'not', and 'imply'* in the modeling languages. Previously these were restricted to the requirement specification language). According to the definition and semantics of our timed automata, this example will initially start at location S0 pass some time, do a transition *aaa* and go to the location S1. At location S1, it cannot stay and pass time but has to do a transition *bbb* straightaway then go to location S2. (When control is in S2, there is a deadlock state. But this does not affect the investigation of the false invariant at location S1 in this example, since every state associated with location S1 will be reached before a deadlock occurs)

However, in the UPPAAL simulator, a deadlock is found at the state when transition *aaa* is offered. In [5], the semantics, on which the UPPAAL simulator is based, define a constraint system and an adjusted constraint system. The constraint system describes when it is possible to take the transition, and the adjusted constraint system strengthens the constraint system in order to ensure that it is in addition possible to reach the targeting location at which control can stay. When the timed automaton in Figure 3 is simulated in the UPPAAL simulator, it reaches a deadlock state when transition *aaa* is offered.

3.2 Deadlock

As stated in [3], in a very broad sense, deadlocks are states where the system is unable to progress further. We would expect the system model to be able to run forever hence deadlocks can be seen as error situations. In untimed systems, deadlocks are states where the system will never be able to perform an action. However, in timed automata, the range of transitions has been broadened to time passing and action transitions. Consequently, the ways of violating the requirements of progress can vary in this case. So deadlocks in timed automata can be of different types. In a later part of this section, formal definitions of progress requirements will be given.

The UPPAAL model checker can detect a subset of those deadlocks. Tsoronis [1] and other researchers [2, 3] have stated their investigations of earlier versions of UPPAAL. Here similar investigations have been made with the current release UPPAAL 3.4 beta 2 (UPPAAL - 3.3.35), 14th May 2003.

Before giving the formal definitions of progress requirements and various types of deadlocks, we briefly review the terminology used by [2, 3, 4], and the following definitions are described in terms of this terminology.

Similarly to the definition of [3, 4], a trace of a timed automaton A starting from state s_0 is a finite or infinite sequence: $\rho = s_0 \xrightarrow{y_1} s_1 \xrightarrow{y_2} \cdots \xrightarrow{y_i} s_i \xrightarrow{y_{i+1}} \cdots$, where, $y_i \in A^+ \cup \mathbb{R}^+$ (y_i is an action transition if $y_i \in A^+$; y_i is a time passing transition if $y_i \in \mathbb{R}^+$); s_i denotes the *i-th* state; s_0 is the initial state of A; $\forall s_i \ in \ \rho \cdot s_i \in Reach(A)$, where $t_i \ in \ \rho$ if and only if t_i is an element within the sequence ρ.

Function $Reach(A)$ is the set of all reachable states of A. A state s is reachable if there exists a finite trace $\rho = s_0 \xrightarrow{y_1} \cdots \xrightarrow{y_i} s$, for $i \geq 1$. A state s is reachable also if it is the initial state such that $s = s_0$.

Let $Tr(A)$ denote the set of all traces of timed automata A, and define the function $delay(\rho)$ as: $delay(\rho) = \sum\{y_i \mid y_i \ in\ \rho \wedge y_i \in \mathbb{R}^+\}$. This function accumulates the duration of a trace by summing all the time passing actions.

3.2.1 Progress Requirements

In this section, we give two progress requirements in order to define different types of deadlocks. As those types of deadlock were described in the Venn diagram in [1], Figure 4 shows the relation between different types of deadlocks.

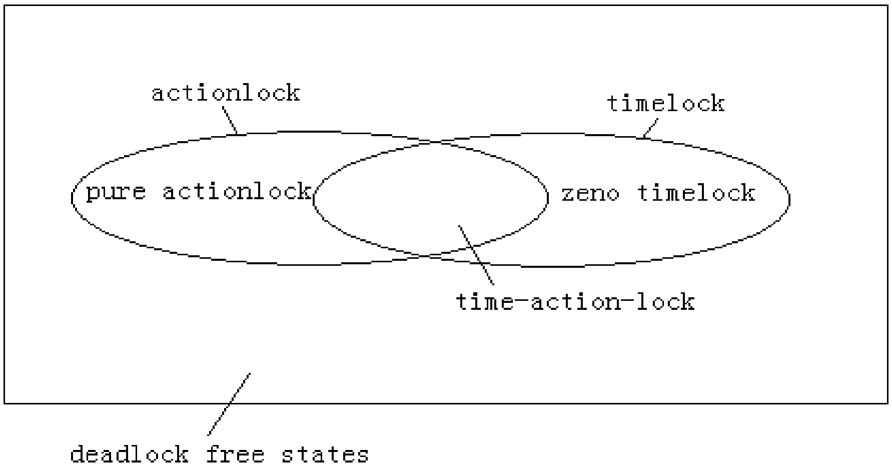

Figure 4: Deadlock Types

Time Progress Requirement

A timed automaton A can block time elapsing beyond time d if $\exists \rho \in Tr(A) \cdot (delay(\rho) < d \wedge \forall \sigma \in Tr(A) \cdot (\rho\ pref\ \sigma \Rightarrow delay(\sigma) < d))$ where $\rho_1\ pref\ \rho_2$ if and only if ρ_1 is a prefix of ρ_2. We call this *the time progress requirement*. However, this property does not require d units to be the minimum upper bound of the passage of time. This requirement is useful because we can use it to check if a timed automaton contains a timelock, where timelock is used as an informal term for a situation in which time can not pass beyond a certain point.

Action Progress Requirement

In [3], a definition of an actionlock state is given as follow: A state $s = (l, v)$ of a timed automaton A is an actionlock, if and only if, $\forall t \in \mathbb{R}^{+0} \cdot ((l, v + t) \in Reach(A) \Rightarrow \neg(l, v + t) \xrightarrow{a})$. This definition states that an actionlock state (l, v) is such that for any amount of delay at location l, no action can be performed. A timed automaton A is actionlock-free if none of its reachable states is actionlocked, so this definition of actionlock states can be used as an action progress requirement, i.e. if no state in $Reach(A)$ is actionlocked.

The next sections will give formal definitions of three types of deadlocks and check if they satisfy these two requirements.

3.2.2 Time-action-lock

In [3, 4], definitions of time progress and timelock are given. In these papers the term time-action-lock and zeno-timelock are used for the two forms of timelock.

Another type of deadlock apart from a pure actionlock is called a time-action-lock in [1] and this paper; this is normally caused by composition. It does not allow either action transitions to be performed or time to be passed.

The transition system of a timed automaton A contains a time-action-lock state s if and only if, $\exists s \in Reach(A) \cdot (\forall a \in A, \forall x \in \mathbb{R}^+ \cdot \neg(s \xrightarrow{a}) \wedge \neg(s \xrightarrow{x}))$.

Now we prove according to the definition of a time-action-lock state, $s = (l, v)$ is an actionlock state and from that state time cannot pass beyond a certain point.

Proposition:

If the state (l, v) is a time-action-lock state, it is an actionlock as well.

Proof:

We consider two cases separately for the state $(l, v + t)$

Case 1: (assume $t = 0$)

When $t = 0$, $(l, v+t)$ equals to the state (l, v). Because (l, v) is a time-action-lock state, $\neg((l, v) \xrightarrow{a})$. So the predicate $(l, v+t) \in Reach(A) \Rightarrow \neg((l, v+t) \xrightarrow{a})$ holds.

Case 2: (assume $t \in \mathbb{R}^+$)

When $t \in \mathbb{R}^+$, $(l, v+t) \notin Reach(A)$. The predicate $(l, v+t) \in Reach(A) \Rightarrow \neg((l, v+t) \xrightarrow{a})$ holds. So $\forall t \in \mathbb{R}^{+0} \cdot ((l, v+t) \in Reach(A) \Rightarrow \neg((l, v+t) \xrightarrow{a}))$ and (l, v) is an actionlock state.

Proposition:

If $\exists s \in Reach(A) \cdot (\forall a \in A, \forall x \in \mathbb{R}^+ \cdot \neg(s \xrightarrow{a}) \wedge \neg(s \xrightarrow{x}))$ holds, then $\exists \rho \in Tr(A) \cdot (delay(\rho) < d \wedge \forall \sigma \in Tr(A) \cdot (\rho\ pref\ \sigma \Rightarrow delay(\sigma) < d))$ will hold.

Proof:

Let state s be a time-action-lock state and $s \in Reach(A)$. So there exists a finite trace ρ that reaches state s. And let $d = delay(\rho) + \epsilon$, where ϵ is a sufficiently small positive value. So $delay(\rho) < d$. According to the definition of a time-action-lock state i.e. $\forall a \in A, \forall x \in \mathbb{R}^+ \cdot \neg(s \xrightarrow{a}) \wedge \neg(s \xrightarrow{x})$, hence if follows that $\forall \sigma \in Tr(A) \cdot \rho\ pref\ \sigma \Rightarrow \sigma = \rho$ holds. $delay(\sigma) = delay(\rho) < d$,. So the timed automaton A can block time elapsing beyond time d because $\exists \rho \in Tr(A) \cdot (delay(\rho) < d \wedge \forall \sigma \in Tr(A) \cdot (\rho\ pref\ \sigma \Rightarrow delay(\sigma) < d))$.

An example of a time-action-lock is shown in Figure 5. The upper automaton must perform the action $a!$ before 5 units of time while the bottom one can only do the action $a?$ after 5 time units. The system will reach a time-action-lock state at 5 time units since they have to synchronize on channel a and time can not pass beyond 5 time units.

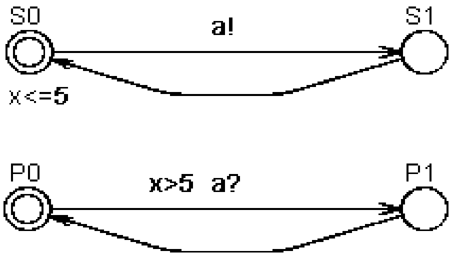

Figure 5: an example of a time-action-lock

Because the deadlock can cause the system to reach a state, in which no action transition can be performed, the querying of the satisfaction of *A [] not deadlock* will eventually fail. However, UPPAAL does not distinguish this kind of deadlock from the previous situation.

3.2.3 Pure Actionlock

One type of deadlock is called a pure actionlock [1]. Informally speaking, the term pure actionlock refers to a state of a system that cannot perform any action transitions, but the passage of time is still allowed.

The definition for an actionlock above is not a suitable definition for a pure actionlock, because of the character of implies ($\Rightarrow$), such that $P \Rightarrow Q$ is *true* if P is *false*, whatever Q is. Taking a time-action-lock as a counterexample, a timed automaton A reaches a time-action-lock state (l, v). That means from state (l, v), there are neither outgoing action transitions nor time passing transitions. As proved in 3.2.1, we know that (l, v) is an actionlock state.

The reason why the definition of an actionlock above is not suitable for a pure actionlock is that the predicate does not require time progress. However this requirement is the key point to distinguish a pure actionlock from a time-action-lock. The way of defining a pure action lock is to use the relation between the actionlock and pure actionlock. We can define a pure actionlock as the set of all actionlocks complementing the set of all timelocks. So we give the definition of a pure actionlock as follow:

The transition system of a timed automaton A contains a pure actionlock state (l, v) if and only if, $\exists (l, v) \in Reach(A), \forall t \in \mathbb{R}^{+0} \cdot (\neg((l, v+t) \xrightarrow{a}) \wedge ((l, v+t) \in Reach(A)))$.

In the definition above, x refers to the units of time passage. It is obvious that (l, v) is an actionlock. Now we are going to prove that from a pure actionlock state (l, v), time can pass arbitrarily.

Proposition:

If the transition system of a timed automaton A contains a pure actionlock state (l, v), A can pass time arbitrarily.

Proof:

Because (l, v) is a reachable state. There exists a finite trace ρ that reaches that state. Let $d = delay(\rho) + \pi$, where π is a unbounded big positive finite value (π can be as big as we need). So $delay(\rho) < d$. According to the definition of a pure actionlock state, the time passage units t can be arbitrary great from every reachable successor states of (l, v). So there must exists a time passage of t units that is greater than π, such that $delay(\rho) + t > d$. We could assume that the state of the terminate σ is the target of (l, v) by passing t units of time, such that $delay(\sigma) = delay(\rho) + t > d$. So $\neg\exists\rho \in Tr(A) \cdot (delay(\rho) < d \wedge \forall\sigma \in Tr(A) \cdot (\rho \ pref \ \sigma \Rightarrow delay(\sigma) < d))$, which means timed automaton A can pass time arbitrarily.

An example of a timed automaton with a pure actionlock is shown in Figure 6, in which it reaches a pure actionlock state when control is in location *stop*.

Figure 6: an example of a pure actionlock

The model checker *verifyta* can check this kind of deadlock by querying the satisfaction of *A [] not deadlock*, and obtaining a negative result. Also a trace can be generated and fed back into the simulator. This trace states that the timed automaton starts from the initial location *Start*, stays in this location within the time region $t \in [0, 10]$, passes some time and does the transition *aaa* at a point of time in the region $t \in [0, 10]$, and then goes to the location *Stop*. Since control is in location *Stop*, it can only pass time but cannot do any actions, and the simulator will pop out a dialog window saying it is a deadlock state.

3.2.4 Zeno-timelock

The last type of deadlock is a zeno-timelock [1]. In a deadlock of this kind, systems can still perform transitions (which can be either time passing transitions or action transitions) but cannot allow time to advance beyond a certain point. That means it can do an infinite number of transitions in a finite period of time.

Similarly to defining a pure actionlock, we give the definition of a zeno-timelock. A timed automaton A contains a zeno-timelock state (l, v) if and only if the following hold,

$\exists(l, v) \in Reach(A) \cdot (((\exists\rho \in Tr(A) \wedge ((l, v) = last \ \rho) \wedge \exists d \in \mathbb{R}^+ \cdot delay(\rho) < d \wedge \forall\sigma \in Tr(A) \cdot (\rho \ pref \ \sigma \Rightarrow delay(\sigma) < d)) \wedge (\neg\exists(l, v + t) \in Reach(A), \forall t \in \mathbb{R}^{+0}) \cdot (\forall a \in A, \forall x \in \mathbb{R}^+ \cdot \neg((l, v + t) \xrightarrow{a}) \wedge \neg((l, v + t) \xrightarrow{x}))).$

This definition states that if there is a zeno-timelock state (l, v) and a trace ρ reaches and terminates at that state, every trace that is extended from ρ cannot pass time beyond d units of time. Additionally, there does not exist a successor state of (l, v) that cannot perform neither action nor time passing transition, such as a time-action-lock state.

For example consider the automaton shown in Figure 7, where an infinite number of action transitions must be performed before 5 units of time, hence time cannot pass beyond 5 time units.

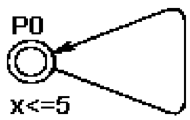

Figure 7: an example of a zeno-timelock

The automata in Figure 7 and Figure 8 are checked against the property of *A [] not deadlock* using the current released version UPPAAL 3.4 beta 2. The model checker still cannot detect a deadlock in the automata in Figure 7 and Figure 8.

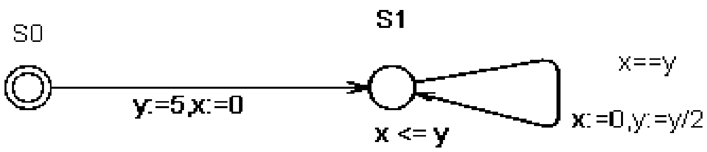

Figure 8: an example of a zeno-timelock, x is a clock and y is an integer variable

Because integer division is used in the assignment of integer value y at the transition from location S1 to S1, the automaton in Figure 8 is not the same model as the zeno paradox that we thought is should be, such that time can always pass half of the length of the previous step. Then we use the simulator to generate a trace of the automaton in Figure 8. From the initial state S0, it can pass some time and do a transition, by which the clock x then is reset to zero and the integer variable y is assigned to 5, and the control goes to the state S1. Control can stay in location S1 within the region $x \in [0, 5]$ and may take a transition at 5 time units, before then going back to the same location S1, then the clock x is reset to zero and y is divided into half. The variable y becomes 2 because of the integer division of y. The process starts again as the first run, but y is assigned to 2 at this time. In the following run y is assign to 1 and then to zero in the one after that. When y has the value of zero, the timed automaton in Figure 8 will be the same state as the timed automaton in Figure 7, and the value of y remains zero unchanged for each successor runs.

An alternative model of zeno-timelock is shown in Figure 9 in order to avoid an integer division, where y is a clock variable rather than an integer. But this automaton cannot be modeled in UPPAAL since the binary operation division '/' does not allow clock variables to be operands.

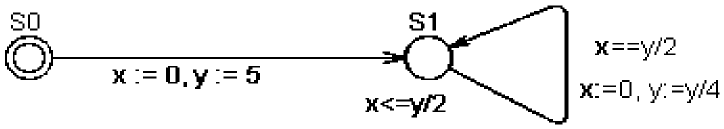

Figure 9: an example of a zeno-timelock, x and y are clock variables

4 Timelock detection

In this chapter, we are going to provide an algorithm to ensure timelock freedom and we are going to implement it, then to test it. As we have discussed in Section 3, deadlocks can be serious problems in systems. The reasons why we say that deadlocks are problematic are as follows:

1. Deadlocks can terminate the proceeding of systems, i.e. the occurrance of an actionlock will stop any further actions from being performed;

2. Deadlocks can have effects on other systems, which do not contain any deadlocks, when there is parallel composition between them, i.e. the occurrence of a timelock in one system will block time from progressing beyond a certain point and stop any actions following that state to be performed. Moreover, all actions in any system which run parallel with that timelocked system can not be performed if they need to be performed at or after that blocked point of time;

3. Deadlocks are hard to avoid, prevent, or even detect, and harder to cure. One reason is that deadlocks are of different types as we have investigated in Section 3 and each type of deadlock has different characteristics, and affects on the systems, so that different ways of detection and prevention are needed. Another reason is that deadlocks in differently structured systems, such as distributed system, are more complicated since the information that can help us to detect or prevent deadlocks is scattered in various places;

4.1 Strategies to Handle Deadlocks

Various strategies are used to handle deadlocks and are stated in [6]. Four of the best-known ones are listed and discussed below.

1. The ostrich algorithm (ignore the problem).
2. Detection (let deadlocks occur, detect them, and try to recover).
3. Prevention (statically make deadlocks structurally impossible).
4. Avoidance (avoid deadlocks by allocating resources carefully).

The ostrich algorithm is both theoretically and practically effective for many applications, however, this algorithm is not discussed in this paper because formal methods such as specification and verification attempt to provide mathematical support for the design of systems and guarantee systems operate correctly, whereas the ostrich algorithm does not.

Deadlock detection using formal methods can be done by simulation, that is to say, let deadlocks occur when we simulate the model, detect the deadlocks and try to correct them by redesigning the system and repeating the simulation in order to find more design errors. A technique of deadlock detection in simulation stage will be discussed later in this section.

Deadlock prevention is one of the main aims of specifying and verifying systems. Deadlock prevention can be performed before we run the system. Deadlocks can be found as soon as possible by either looking at the whole system or by looking at a certain part of syntax of the specification of the system. In the later part of this section, an algorithm to ensure timelock freedom will be presented for which a simulation was implemented in Java.

Finally, deadlock avoidance is not discussed in this paper since the information we need to allocate resources is nearly impossible to obtain from high level specifications of systems, such as timed automata.

4.2 Strongly Non-zeno

In order to give the algorithm for timelock detection, a lemma that is given in [4] is as below. It expresses the notion of strongly non-zeno.

A timed automaton A is timelock-free if for each state $s \in Reach(A)$, there exists a trace ρ starting from s, and some point in ρ, such that $delay(\rho) \geq 1$.

From the lemma above, we can informally define the notion of strongly non-zeno. A timed automaton A is strongly non-zeno if and only if that at least one unit of time elapses in every loop of A. For example the timed automaton in Figure 10 is strongly non-zeno.

A formal definition of strongly non-zeno can be regarded as an algorithm of timelock prevention. This algorithm can check a subset of zeno-timelocks. In [4] the formal definition of strongly non-zeno is presented as below.

Consider a timed automaton A.

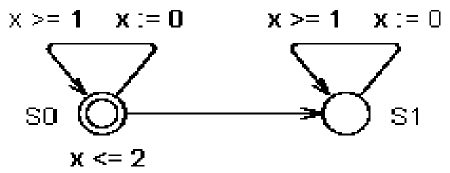

Figure 10: a strongly non-zeno timed automaton

A structural loop of A is a sequence of distinct edges $e_1 \dots e_m$ such that $target(e_i) = source(e_{i+1})$, for all $i = 1, \dots, m$ (the addition $i+1$ is modulo m). Modulo p means that if $a = mp + q, 0 \leq q \leq p-1$, then in modulo arithmetic $a = q$.

A is called strongly non-zeno if for every structural loop there exists a clock x and some $0 \leq i, j \leq m$ such that:

1. x is reset in step i, that is, $x \in reset(\rho)$; and
2. x is bounded from below in step j, that is, $(x < 1) \cap guard(e_j) = false$.

Strongly non-zeno is interesting and useful for deadlock detection and prevention because of the following reasons. (As they are introduced in [4])

1. These conditions of strongly non-zeno are sufficient but not necessary, since they only consider the static structure of the timed automata syntactically but not the reachable state space in the presence of timing constraints. Whereas we can believe that a timed automaton, which is strongly non-zeno, is a safe system by consideration of the timelock freedom.
2. On the other hand, strongly non-zenoness can be quite useful in practice, since these conditions are not expensive to check by simply looking into the syntax of timed automata. However, this algorithm can only check a relatively small subset of zeno-timelocks and some examples and discussion will be seen in the later part of this section.
3. Additionally, strongly non-zenoness is preserved by parallel composition, so that it can be efficiently checked on large systems. Firstly, we can decompose large systems and check each component individually. Secondly, some components may be untimed, that is, they can be modeled using simple finite state machines without clocks. These components can be considered strongly non-zeno by convention, so that their parallel composition with the rest of the system does not affect the strongly non-zenoness of the global system.

4.3 Implementation of Timelock Prevention

In this section, a Java implementation of timelock detection algorithm is introduced. This original program is a part of my MSc project, and in this paper some code will be included in order to explain the algorithm behind the computer program and help understanding the project itself.

This algorithm can be divided into three significant parts: the data structure representing timed automata syntactically, the cycle detection to find all the structural loops and check the conditions of strongly non-zeno.

4.3.1 Data structure of Timed Automata Syntax

Timed automata can be seen as directed graphs. It is vital to discuss the connectivity of the graphs of timed automata before we start discussing the algorithm.

In [8], it is defined that a graph is connected if there is a path between any two vertices. A directed graph is weakly connected if it would be connected by ignoring the direction of edges. A directed graph is strongly connected if there is a directed path between every pair of vertices. We assume that the graphs of timed automata are weakly connected graphs if there is no actionlock state in the transition system of timed automata. But we believe that not all the graphs of timed automata are strongly connected. Informal proof of this is a simple counterexample as shown in Figure 11. This automaton is actionlock-free but its initial state S0 is not strongly connected with the rest of the graph since no edge goes in S0. However, this automaton is acceptable by our definition of timed automata.

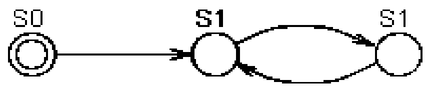

Figure 11: a non-strongly connected graph of timed automaton

Moreover, by assuming the graphs are weakly connected, we cannot guarantee actionlock-free. However, The detection of actionlock is not the issue we are going to investigate in this paper.

Following the definition of timed automata and the assumptions we have just made above, a class of timed automata can be shown in the Java like pseudo-code below.

class TimedAutomata {

  private String name;

  private Location[] l; // array of locations

  private Edge[] e; // array of edges corresponding to transition relations

private String[] clock; // array of clocks

private String[] bool; // other data type of timed automata, i.e. Boolean or integer

private String[] inte;

}

The communication channels between timed automata are defined in the system of timed automata rather than within a single timed automaton. But those channels should be considered when we initialize edges containing synchronizations.

The class of locations and edges with a similar manner are shown below.

class Location {

private String label; // a label that attached to each location

private int id; // a internal unique id used for distinguish locations

private String[] invariant; // array of invariant

private boolean initial, urgent, committed; // flags which indicate if a certain location is the initial location, a urgent location or a committed location

private Edge[] in; // array of incoming edges

private Edge[] out; // array of outgoing edges

}

respectively

class Edge {

private int id; // a internal unique id used for distinguish edges

private Location source, target; // source and target locations

private String channal; //synchronization or communication channal

private String guard;

private String[] reset; // reset set for clocks

private String[] assignment; //assignment for other data types, i.e. integers or booleans

}

According to [7], there are various methods of representing a graph by Adjacency List, Adjacency Matrix, Term Based, Object based, Object based with redundancy and Set based. In our implementation, object based with redundancy representation is chosen. Here, the edges hold information about which nodes they connect to and the nodes have information about which edges go in or out. The advantage of that representation is quick neighbor finding, which will be frequently used in our algorithm. The disadvantage is maintaining the redundant structures. However, in our algorithm, we will not change the structure of the graph, so that we do not have the burden of maintaining the redundant information.

4.3.2 Cycle Detection

The second task is that we have to detect all the structural loops in timed automaton. We introduce a cycle detection algorithm for the particular case of graph of timed automata. The graph is directed and weakly connected as we assumed above.

As defined in [8], a cycle is a path where the last vertex is adjacent to the first. The structural loop can be seen as cycle. If we can find all the cycles in the directed graph, we will find all the structural loops.

The graphs of timed automata are different from the general directed graph, so that we introduce a cycle detection algorithm ourselves. The basic idea behind the program is a Depth First Search, whenever we see a node we have visited before we can indicate a cycle. A Java like pseudo-code below can help understanding the algorithm.

```
public Edge[][] getStructalLoop() { // return array of structural loops, each
    structural loop contains an array of edges
    Stack solved;
    Stack unsolved;
    Location targetLocation;

    solved.push(l[0]); // l[0] is the initial location
    if (solved.peek().getOutEdge().getTarget() exists)
        targetLocation = solved.peek().getOutEdge().getTarget();
    if (there is other unvisited branches) {
        unsolved.push(null); // in order to separate from previously saved
            branches
        save other unvisited branches ;
        unsolved.push(solved.peek()); // save starting point of branches
    }
    while (targetLocation != null) {
        search solved stack for same location as targetLocation ;
        if (have visited) { // we found a cycle
            return edges in that cycle ;
            if (unsolved stack is not empty) { // there are unvisited branches
                do solved.pop();
```

```
                until (solved.peek() = unsolved.peek())
                 unsolved.pop();
                 targetLocation = unsolved.pop();
                if (unsolved.peek() == null) unsolved.pop();
                   else unsolved.push(solved.peek());
              }else // there is no unvisited branch
                 targetLocation = null;
          }else{ // has not formed a cycle yet
              solved.push(targetLocation);
             if (solved.peek().getOutEdge().getTarget() exists)
                 targetLocation = solved.peek().getOutEdge().getTarget();
             else
             if (unsolved stack is not empty) { // there are unvisited branches
                do solved.pop();
                until (solved.peek() = unsolved.peek())
                 unsolved.pop();
                 targetLocation = unsolved.pop();
                if (unsolved.peek() == null) unsolved.pop();
                else unsolved.push(solved.peek());
              }else // there is no unvisited branch
              targetLocation = null;
             if (there is other unvisited branches) {
                 unsolved.push(null); // in order to separate from previously saved branches
                save other unvisited branches ;
                 unsolved.push(solved.peek()); // save starting point of branches
              }
          }
      }
}
```

There is a difference between this algorithm and a general depth first search algorithm, such that we always initially start from the initial location and ignore other connected components that are unreachable from initial location. While a general DFS algorithm choose a node randomly as start point and then search over the connected component that this node belongs to. Then choose another node randomly as start point from unvisited nodes until all nodes have been searched. But the algorithm that we present will stop searching immediately after all nodes have been visited in the connected component that initial location belongs to. After implement the cycle detection algorithm, we have done a software testing.

4.3.3 Conditions of Strongly Non-zeno

The result from previous stage is a set of structural loops. In this stage the algorithm checks if both the conditions in definition of strongly non-zeno hold. A Java like pseudo-code below can help understanding the algorithm.

```
for each structural loop do {

    for each clock do{
        for each edge in that structural loop do {
            if (that clock in reset set of this edge) {
                conditionI = true;
                break;
            }
        }
        if (conditionI) {
            for each edge in that structural loop do {
                if (guard contains that clock) {
                    if (guard is a expression of clock x such that
                    (x > a || x >= a || x == a) && a >= 1){
                    conditionII = true;
                    break;
                    }
                }
            }
        }
    } until (conditionII)

}
```

The code for checking conditions of strongly non-zeno is much simpler and clearer than the one for cycle detection because in the later one we try to state the operation of those two stacks in detail.

As stated in previous section, the algorithm of cycle detection has many assumptions. The strongly non-zenoness has quite a few assumptions as well. Consequently, these assumptions limit the algorithm to check only a small subset of zeno-timelock. The limitations and assumptions we made are discussed and summarized as following.

1. The second condition $(x < 1) \cap guard(e_j) = false$ equals to $guard(e_j) \Rightarrow x \geq 1$. That is to say when transition e_j in that structural loop is offered, i.e. $guard(e_j) = true$, clock x can pass one unit of time, in addition to the first condition in strongly non-zeno that ensures the clock is reset to zero at some point in the structural loop. When these two conditions both

hold, we can ensure that time pass by 1 unit for each loop. We only take the clock value when transitions are offered into account because strongly non-zeno algorithm is a static algorithm, which only checks the syntax of timed automata. We are not able to know the clock value when control is in locations if we do not execute the system. Consequently, we only check edges rather than locations for strongly non-zeno. There are also reasons about composition which are out of the scope of this paper.

2. By understanding the first limitation of strongly non-zeno, we can see that the first condition in strongly non-zeno is compulsory in the algorithm since this condition gives us a fixed value of clock in the structural loop so that we can compare with the value of clock when transition e_j is offered. Using static methods such as only check the syntax of timed automata, we are also not able to know the value of clock unless it is explicitly written in the syntax i.e. guards, resets and invariants. In addition, some guards have the form of a clock equals to zero, for example $x == 0$, we can compare the value zero with the value of clock when transition e_j is offered.

4.3.4 Input and Output

In this section we are going to look at how a specification of timed automata is input into the detection program and how the verification result is output.

For most of the software, input and output is one of the most difficult parts to build. In order to have reliable and simple way of input a specification, we chose the XML files, which are generated from UPPAAL integrated tool suit, as our input. As Java did for portability of code, XML is supposed to do for portability of data.

XML is the Extensible Markup Language, and it allows simple representation of data and flexibility of data formatting, whereas XML specifies neither the tag set nor the grammar for that language, i.e. it allows the user to define the content of data in variety of ways as long as it conforms to a general structure that XML requires. XML aware applications and parsers then used the general structure to make sense of the XML document and perform some action with the data. [13]

Fortunately, XML has been paired with Java since its beginning and Java includes the most complete set of APIs to make use of XML directly within Java code. To get access to the XML data, there are two steps: parsing the XML document and manipulating the data within it.

Application can use XML document in various way. One of the usages is for communication, i.e. the same XML document and its data content can be used to transfer information between applications. This communication is inexpensive to achieve since the XML data is not tied to any type of application, language and platform, therefore XML provides a standard way of exchanging data.

The process of input a timed automaton into the time-lock detection program is shown in Figure 12. User can input a specification in the graphical interface of UPPAAL. The syntax checker will automatically check the input timed automata. In particular the use of clocks, auxiliary integer variables and channels must be in accordance with their declaration. Saving the system will result of a XML document containing a lightweight specification.

Then feed this XML file into our program in order to check if every structural loop is strongly non-zeno. Type the command line below to run the software in any computer where Java virtual machine has been installed.

Java StronglyNonZeno [XML URI]

The XML URI indicates the source of system specification to verify. It can be local or remote.

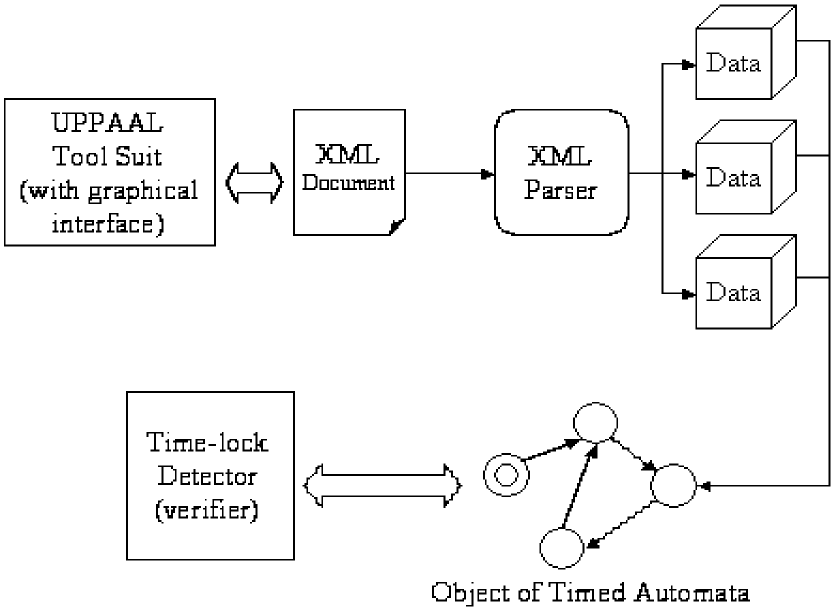

Figure 12: the architecture of XML document input

After the XML being loaded and parsed, we are able to use the content and attributes within the document to construct the data structure presented in Section 4.3.1 Data structure of Timed Automata Syntax. Once getting a linked list of Java objects representing the timed automaton, we are able to execute both cycle detection and strongly non-zeno check against them.

However, we have not mentioned the actual parser we chose for our implementation. There are two Approaches to Parsing XML documents: [13]

1. Simple API for XML (SAX). It provides an even-based framework for parsing XML data, which is the process of reading through the document and breaking down the data into usable parts. Events are sent as the XML document is parsed, e.g. StartDocument, StartElement, EndElement, characters, etc. It leaves the application program to determine what elements are important.
2. Document Object Model (DOM). It creates a tree view of the XML document.

In our program, DOM is chosen to parse the XML document since DOM has richer set of functions for interpreting content in XML document than SAX. The XML files that UPPAAL generates have complicated but universal structure that is easy to represent by a document tree, this feature matches the DOM tree. One significant drawback to DOM is it reads an entire XML document into memory and our program only needs to access the data once in the beginning, it introduces degradation of performance. However, this is not a big problem since the XML representation of timed automata specification is light weighed, i.e. it should not be a big problem if the target system is not too large.

So far we are able to access the data represented in XML document that UPPAAL generates, and to execute the cycle detection and strongly non-zeno algorithm on the timed automata. The results that are given by the program is output into a textual file with extension of *.result*. The resulting file contains all structural loops of each timed automaton and if each of them is strongly non-zeno.

In the next section a set of testing is done in order to check the implementation and the underlying algorithms. In addition, a case study of CSMA/CD protocol is specified in UPPAAL and verified by our program.

4.4 Software Testing

4.4.1 Testing of Input

Most of the testing has been done together with the testing of cycle detection and the case study. The result of the testing is that all the systems tested are correctly input and parsed. However, the specifications in UPPAAL are restricted by a set of principles in order to be resolved by our program. There should not be any comments in the specification. Users need to delete all comments manually before parsing the XML documents. In addition, when writing the guard, reset and invariant, operands and operator must be separated by space. The clock and variable must be written in the left side of the operator. Any guard, reset and invariant that is written in other formats will be ignored by our software and result in verification failure.

4.4.2 Testing of Cycle Detection

This part of testing is the most important since the cycle detection algorithm is introduce by us and has not been formally proved. We are not able to give formal proof of this algorithm since the limited time that given for the project, however we try to provide a testing systematically. The first target graph that has been tested is a single node without any edge, and then we added nodes and edges gradually. Figure13, Figure 14 and Figure 15 show that the target graphs that are tested and the program can find all structural loops in those graphs.

In Figure 13, the graph (a) is the simplest situation and the cycle detector found no cycle. The graph (b) adds one edge the detector found one cycle and the graph (c) adds two edges it found two loops. Whereas a graph, which has more than two edges are added in a single node, would make no difference from the graph (c) regarding to our cycle detection algorithm, then we can believe that the algorithm works for all the graphs with only one node but arbitrary number of edges.

The next stage of the testing is to add another node and to try all sorts of ways to connect it with the rest of the graph, checking them to see if the algorithm works. The graph (d), (e), (f) and (k) are the graphs that a node has added but disconnects with the rest of the graph, i.e. the initial location. The cycle detector can only find all the cycles in connected component that includes the initial location, however it cannot find any cycle in the component that we added to the original graph. We had assumed that the timed automata are weakly connected graphs. The graphs that have unconnected components are not discussed in this paper. Although the graph (h) is weakly connected, the initial location is the only reachable node and this graph is equivalent to the graph (d).

The third stage of the testing starts with the graphs (g), (i), (j), (l) and (m). The detector is able to find each cycle since every node in these graphs is reachable. Then we can look more complicated graphs that are shown in Figure 14.

In Figure 14, we tested the graph that has two or three node originally and another node and some edges are added. The lower node in each graph is the new node that is added to the rest of the graph. Graph (a) and (b) describes there is only edge(s) which go in or out of the new node. In these graphs no cycle are found. The graph (c), edges go in and out of the new node from or to the same node in the original graph, a cycle in each graphs has been detected. The graph (d), (e) and (f) are the situation that edges go in the new node from one location but leave to another location. The cycle is only found in graph (e).

The graphs in Figure 14 represent the most typical way of adding a node to a graph. Our cycle detector is able to find all the cycles that are introduced by the additional node and edges. Consequently, we think the cycle detection algorithm is able to work for most of automata. However, there are number of drawbacks of the algorithm, the automata in Figure 15 are two examples.

One drawback is that the algorithm is not suitable for a general directed graph without the assumptions, which restrict a graph in order to represent timed automata. The other drawback of the algorithm is that it can have repetition. Sometimes we need a timed automaton that has two edges, which go to the same location. An example is represented in graph (a) of Figure 15, and more examples can be found from the case study in later chapter. This

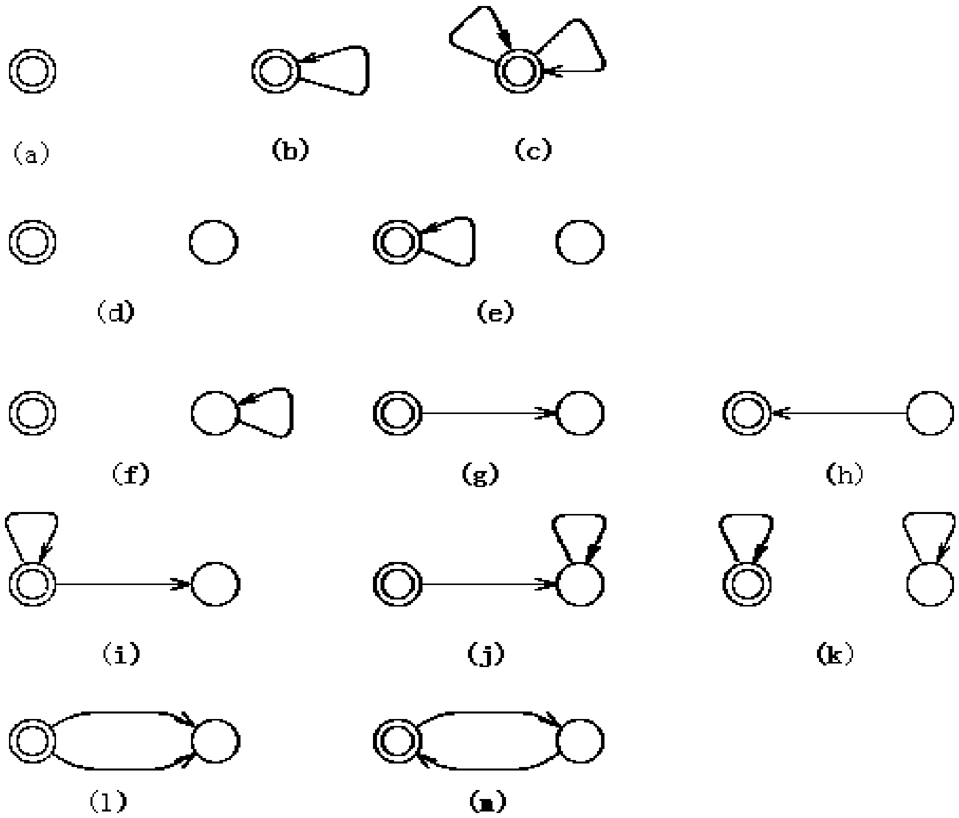

Figure 13: target graphs for software testing

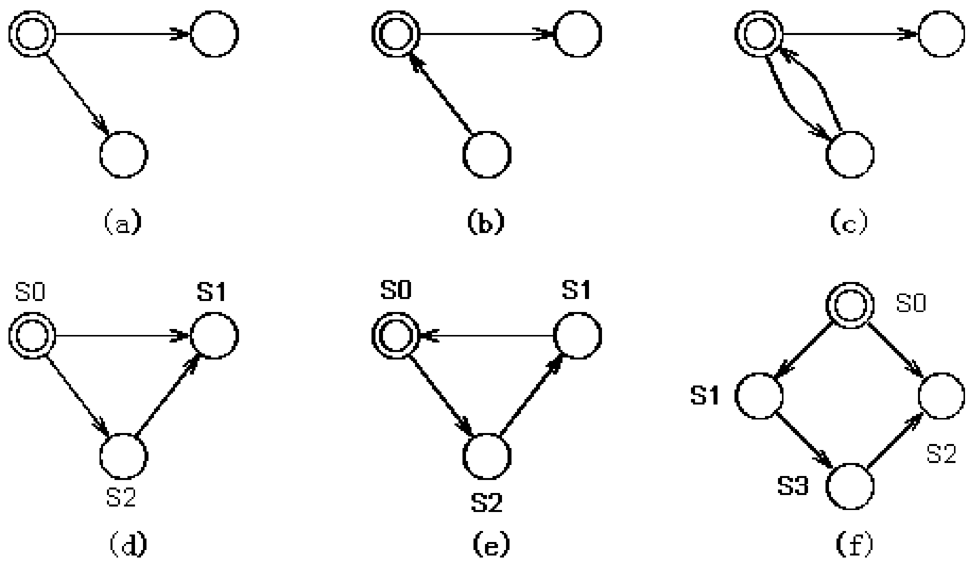

Figure 14: more complicated graphs for the testing

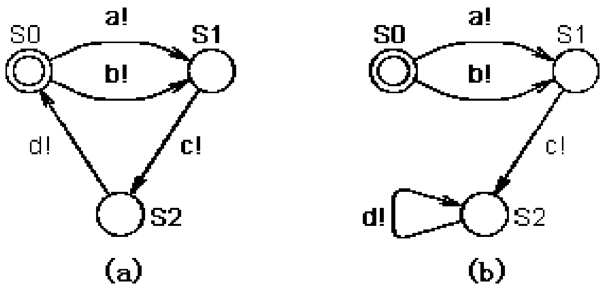

Figure 15: examples of the drawbacks of the algorithm

introduces repetition in finding cycle in successor steps. For the example of graph (b), it finds the cycle $S2 \xrightarrow{edge(d!)} S2$ via the $edge(a!)$ and it finds exactly the same cycle $S2 \xrightarrow{edge(d!)} S2$ once again later via the $edge(b!)$, where the notion of cycle is sequence of nodes and the edges that join them together.

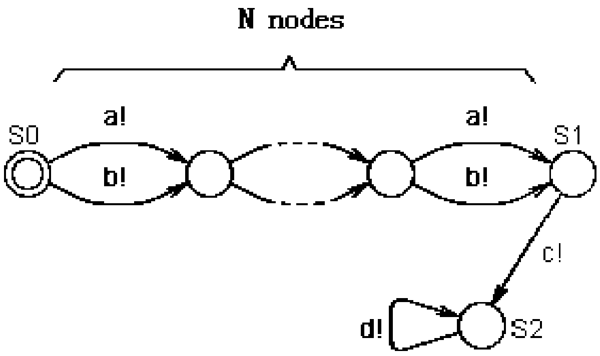

Figure 16: a problem of repetition

In addition, the drawback of repetition can be a serious problem when a great number of nodes, like the S1 in graph (b) of Figure 15, occur continuous before it detects a cycle. For instance, the timed automaton in Figure 16, the cycle, $S2 \xrightarrow{edge(d!)} S2$, 2^Ntimes.

4.5 Deadlock Detection in Simulation

A technique presented by Tsoronis [1] is to let the target system run in parallel with a timelock tester automaton and check whether time can pass by one unit in the timelock tester automaton. The tester is shown below: Figure 17

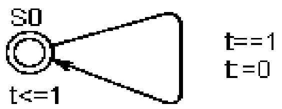

Figure 17: a Timelock Tester

The property that is checked together with the tester is:
$(Ttester.S0andTtester.t == 0) \rightarrow (Ttester.S0andTtester.t == 1)$

As stated in [1], this property means that every reachable state of the system (l, v) where $l = Ttester.S0$ and $v(t) = 0$, along every execution, leads to some state (l, v') where $v'(t) = 1$. Obviously, this technique is based on the idea of strongly non-zeno.

The method of timelock detection introduced by Tsoronis [1] still works with the new version of UPPAAL 3.3.35 and 3.3.36.

We are not going to discuss in depth of this method since it is not the essential of this project. Reader can find detailed discussion in [1].

5 Case Study - CSMA/CD

A specification and verification of the CSMA/CD protocol in UPPAAL has been introduced in [1]. The aim of this section is to follow the specification that is suggested in [1], and then to verify this protocol in our strongly non-zeno checker.

5.1 Introduction of CSMA/CD

It is stated in [14] that CSMA/CD (Carrier Sense Multiple Access with Collision Detection) protocol is widely used on LANs in the MAC sublayer. The idea of this protocol is that when a station has data to send, it first listens to the channel in order to make sure that no other station is transmitting data at this moment. When the channel is idle, it transmits a frame. If the station detects that the channel is busy, it waits until it becomes idle. This protocol is called 1-persistent because the station transmits with probability of 1 whenever it finds an idle channel. In this case study, we follow the 1-persisitent since it does not require non-determinism and probability in modeling. The other two kinds of CSMA protocol are nonpersistent CSMA and p-persistent CSMA. In nonpersistent CSMA, a station waits a random period when it detects a busy channel then repeats the algorithm. In p-persistent CSMA, a station transmits with a probability p when it finds an idle channel. However, in order to model nonpersistent CSMA or p-persistent CSMA, we need to specify probabilistic automata, which is out of the scope of this paper.

Collision Detection means that if two stations sense that the channel to be idle and start transmitting simultaneously, a collision will occur and both of them will see the collision at almost the same time. Rather than finish transmitting their frames, they should stop sending as soon as they detect the collision, since the frames are irretrievably garbled. Collision Detection saves time and bandwidth.

In the worst case a station cannot be sure that it is transmitting and would not interfere until it has transmitted for 2τ without detecting a collision, where τ is the time for a signal to propagate between the two farthest stations. Follow the assumption in [1], a 10Mbps Ethernet with a typical worst-case propagation of 26μsecs and packets with equal length of 1024 bytes. Thus, it takes about 782μsecs to send a packet.

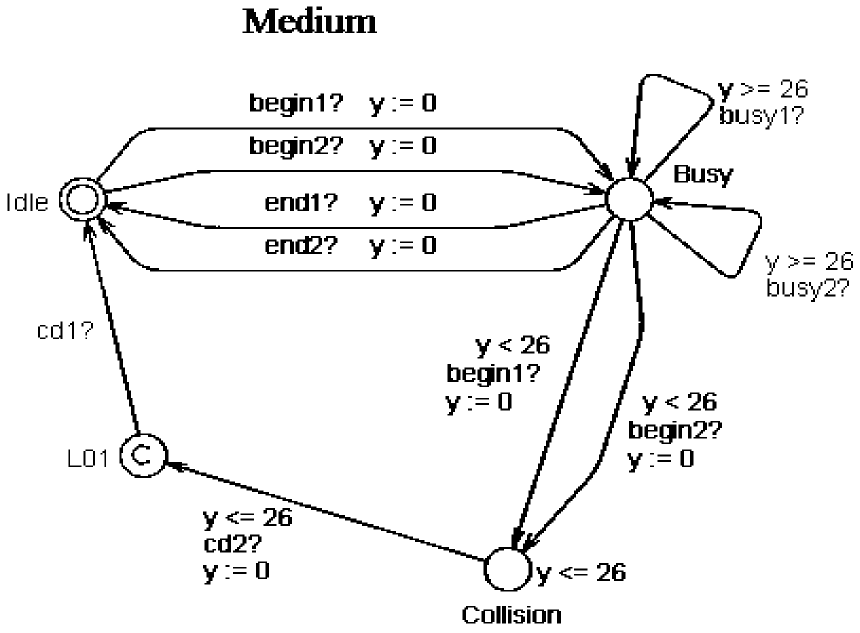

Figure 18: Timed automaton of a Medium

5.2 Specification

The communication system that we are going to model consists a number of stations, called *Senders*, and a single communication channel, called *Medium*. The target system, which we were tested, includes two *Senders* and a *Medium*. The *Senders* and *Medium* communicate via a set of synchronization channels: *begin*, *end*, *busy* and *cd*. The graphical representation of a Sender automaton and a Medium automaton are shown in Figure 18 and Figure 19 respectively.

The System Configuration:

System CSMACD

clock x1,x2,y;

chan send1,begin1,busy1,cd1,end1,send2,begin2,busy2,cd2,end2;

Sender1 := Sender(cd1,busy1,begin1,end1,x1);

Sender2 := Sender(cd2,busy2,begin2,end2,x2);

System Medium,Sender1,Sender2;

The *Medium* is initially in a state *Idle*. If a *Sender*(i) comes along at this moment, ready to send a frame and detect the *Medium* is in *Idle* state, it will

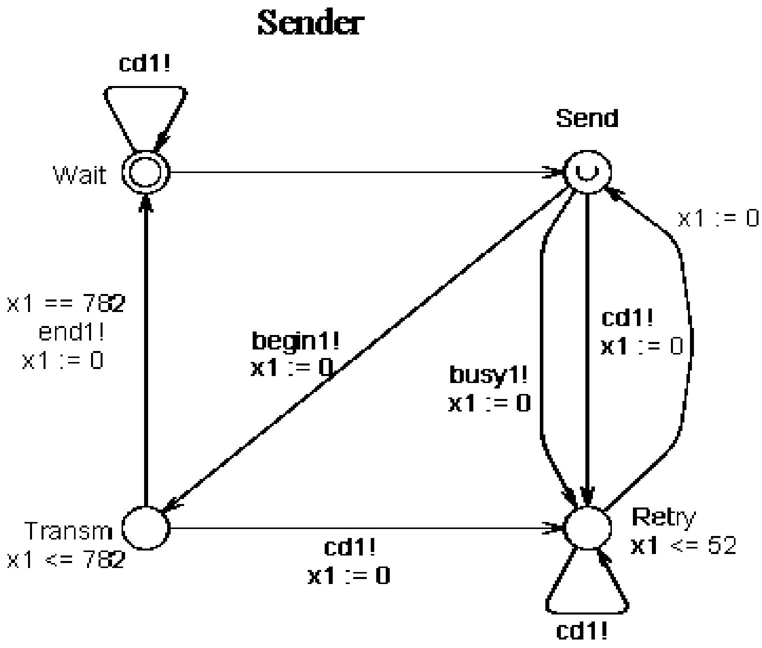

Figure 19: Timed automaton of a Sender

synchronize with the *Medium* at channel $begin(i)$. Then the *Medium* enters a *busy* state, where it allowing other Sender transmitting before 26 units of time. The result of this is a *Collision* state. On the other hand, after 26 units of time, all station in the channel detects the $busy(i)$ channel. It will return to the *Idle* state when the $Sender(i)$ finishes transmitting and notices the *Medium* by a $end(i)$ channel.

When Medium is in a *Collision* state, it notices all stations that collision has occurred by 26 units of time that is the longest time for all station to detect the collision. The *committed* location L01 in between the *Collision* state and *Idle* state achieves the UPPAAL-style multi-way synchronization. Neither time can pass nor interleaving with other locations is allowed in the period from *Senders* see the collision till they stop transmitting.

The *Sender* starts at the *Wait* state initially, waiting for data from upper layer, e.g. IP layer. If there is a collision in the channel, it keeps waiting. This situation can only happen when more than two *Senders* are in the system. Once there is no collision, it enters the *Send* state and is ready to send a frame. The *Send* location is made *urgent* since sender must detect the channel in order to transmit without delay. When an *Idle* channel is found, it goes to the *Transm* state. Otherwise it enters the *Retry* state if the channel is busy or a collision is occurring. In the state of *Retry*, it stays maximum 52 units of time and then goes back to the *Send* state. If it sizes the channel and transmits its frame, it takes 782 units of time to finish and goes to the *Wait* state, the repeating the algorithm.

5.3 Verification

The only property that we checked is Strongly Non-zenoness. The specification was input in UPPAAL and transformed to our program via a XML file. The timed automata of Medium and Sender then were check for time-lock freedom against the strongly non-zenoness.

The result is shown as following:

Structural loops from Medium:

Structural Loop 1 : $Busy \xrightarrow{edge(busy1?)} Busy$. This loop is not strongly non-zeno.

Structural Loop 2 : $Idle \xrightarrow{edge(begin1?)} Busy \xrightarrow{edge(end1?)} Idle$. This loop is not strongly non-zeno.

Structural Loop 3 : $Idle \xrightarrow{edge(begin2?)} Busy \xrightarrow{edge(end2?)} Idle$. This loop is not strongly non-zeno.

Structural Loop 4 : $Idle \xrightarrow{edge(begin1?)} Busy \xrightarrow{edge(begin1?)} Collision \xrightarrow{edge(cd2?)} L01 \xrightarrow{edge(cd1?)} Idle$. This loop is not stronly non-zeno.

Structural Loop 5 : $Idle \xrightarrow{edge(begin1?)} Busy \xrightarrow{edge(begin2?)} Collision \xrightarrow{edge(cd2?)} L01 \xrightarrow{edge(cd1?)} Idle$. This loop is not stronly non-zeno

Structural Loop 6 : $Busy \xrightarrow{edge(busy2?)} Busy$. This loop is not strongly non-zeno.

Structural Loop 7 : $Idle \xrightarrow{edge(begin1?)} Busy \xrightarrow{edge(end2?)} Idle$. This loop is not strongly non-zeno.

Structural Loop 8 : $Idle \xrightarrow{edge(begin2?)} Busy \xrightarrow{edge(end1?)} Idle$. This loop is not strongly non-zeno.

Structural Loop 9 : $Idle \xrightarrow{edge(begin2?)} Busy \xrightarrow{edge(begin1?)} Collision \xrightarrow{edge(cd2?)} L01 \xrightarrow{edge(cd1?)} Idle$. This loop is not strongly non-zeno.

Structural Loop 10 : $Idle \xrightarrow{edge(begin2?)} Busy \xrightarrow{edge(begin2?)} Collision \xrightarrow{edge(cd2?)} L01 \xrightarrow{edge(cd1?)} Idle$. This loop is not strongly non-zeno

Structural loops from Sender:

Structural Loop 1 : $Wait \xrightarrow{edge(cd1!)} Wait$. This loop is not strongly non-zeno.

Structural Loop 2 : $Wait \rightarrow Send \xrightarrow{edge(begin1!)} Transm \xrightarrow{edge(end1!)} Wait$. This loop is strongly non-zeno

Structural Loop 3 : $Send \xrightarrow{edge(begin1!)} Transm \xrightarrow{edge(cd1!)} Retry \rightarrow Send$. This loop is not strongly non-zeno.

Structural Loop 4 : $Retry \xrightarrow{edge(cd1!)} Retry$. This loop is not strongly non-zeno.

Structural Loop 5 : $Send \xrightarrow{edge(busy1!)} Retry \rightarrow Send$. This loop is not strongly non-zeno.

Structural Loop 6 : $Send \xrightarrow{edge(cd1!)} Retry \rightarrow Send$. This loop is not strongly non-zeno.

None of these two automata are strongly non-zeno according to the verification result. However, using the method that is introduced in Section 4.5 Deadlock Detection in Simulation, we found CSMA/CD is timelock free. [1]

The specification is simple enough to check the property of strongly non-zeno manually. Therefore the contribution of this case study is not this verification, but rather the algorithm behind this software. It is clear that the algorithm and its implementation we introduced are indeed generally applicable.

6 Evaluation

6.1 Conclusions

We have investigated the general techniques of specification and verification of concurrent systems. As well as an automatic verification tool UPPAAL, following a formal definition and proof of classification of deadlock that suggested in [1]. At the same time we have investigated the ability of deadlock detection in UPPAAL.

Unfortunately, we found that the UPPAAL verifier has inability of checking zeno-timelock, this made us to search other methods in order to verify zeno-timelock. The notion of strongly non-zeno is introduced and then we implemented a piece of software that based on this algorithm of zeno-timelock detection.

There are several contributions of this paper:

1. It gives the formal definitions of three types of deadlock: pure actionlock, time-action-lock and zeno-actionlock. And for each of them, one or more examples are investigated with the current release of UPPAAL. Moreover, we provided the formal proof for those definitions.

2. In this project, a number of case studies and examples, which were suggested in [1], are retested in the current release of UPPAAL. In addition, several differences appear, e.g. one difference, which appears in these experiments, is that UPPAAL 3.3.35 and 3.3.36 allow the inclusion of variables in guards and invariants, whereas the earlier versions of UPPAAL do not allow variables in guards and invariants.

3. We implemented the zeno-timelock detection algorithm that based on strongly non-zenoness. Our software used the XML file of UPPAAL to exchange information with UPPAAL. It allows inputting complex system with UPPAAL graphical interface easily.

4. While we implemented the software, we designed an algorithm for cycle detection. The algorithm is specially designed for the case of timed automata, rather than a general graph. This software and its algorithm were then tested systematically. The testing process and its result are also included in the paper.

5. Follows the specification of CSMA/CD in UPPAAL, which was suggested in [1], we verified it with our software. The success of the verification shows that this software and its algorithm are efficient, even if we are not able to give a formal proof of the correctness of them.

6.2 Future Work

This MSc project takes only five months, therefore considering the limit of time, a number of limitations can be found. However, some limitations are due to that the author has limited experience and knowledge of formal methods and mathematics. The main drawbacks of this project are listed as following.

1. Our proofs of deadlock classification are uncompleted. For instance, we have proved that the set of all the state, which are satisfy our definition of time-action-lock, is a subset of the intersection set of all the actionlock states and timelock states. But we have not proved the case that is the other way round. Therefore we have not proved that the definition describe the whole set of time-actionlock. Neither do the definition of the other two types of deadlock. A future work would be trying to complete these proofs.

2. We have tested the algorithm of cycle detection. But no formal proof has been provided for the correctness of the algorithm. Another future work is to verify this algorithm and analysis its complexity.

3. Last but not least, the input and output of the software need to be improved, i.e. allowing input any XML document that is produced by UPPAAL.

6.3 Conformance to Original Plan

At the start of preparing the project, a considerable amount of materials, i.e. [1], were read in order to locate the problem area of the project. Therefore one aim of this project is trying to solve limitations in [1].

In the original plan we have set out a target to provide an algorithm for time lock detection that can be implemented in UPPAAL. However, we fail to accomplish this aim since it is not possible to access the source code of UPPAAL.

References

[1] Theodoros Tsoronis, MSc dissertation, *Formal Specification and Verification of Real Time Systems with the Timed Automata Based Tools KRONOS and UPPAAL*, 2001

[2] H. Bowman, *Modeling Timeouts without Timelocks*, In ARTS'99, Formal methods for real-time and probabilistic systems 5th international AMAST workshop, 1999

[3] H. Bowman, *Time and action lock freedom properties for timed automata*, In Forte' 2001,Kluwer, 2001

[4] S. Tripakis, *Verifying progress in timed systems*, In ARTS'99, Formal Method for Real-Time and Probabilistic Systems, 5th international AMAST Workshop, 1999

[5] J.Bengtsson, K.G.Larsen, F.Larsson, P.Pettersson, and W. Yi, *UPPAAL - a tool suite for the automatic verification of real-time systems*, In Proceedings of Hybrid Systems III. LNCS 1066, Springer Verlag, 1996

[6] A.S.Tanenbaum, *Distributed Operating Systems*, Prentice Hall, 1995

[7] P.Rodgers, *Lecture handout of CO862 Gragh Theory*, Computing Laboratory, University of Kent, 2003

[8] S.S.Skiena, *The algorithm design manual*, Springer-Verlag, 1998

[9] R.L.Ibrahim, J.A.Ogden, S.A.Williams, *Should Concurrency be Specified?*, Specification and Verification of Concurrent Systems, Springer-Verlag, 1990

[10] M.Kwiatkowska, G.Norman, R.Segala, and J.Sproston, *Automatic Verification of Real-Time Systems with Discrete Probability Distributions*, In ARTS'99, Formal Method for Real-Time and Probabilistic Systems, 5th international AMAST Workshop, 1999

[11] *Uppaal2k: Small Tutorial*, http://www.docs.uu.se/docs/rtmv/uppaal/ - home page of UPPAAL, 16 October 2002

[12] H. Bowman, G. P. Faconti, J. P. Katoen, D.Latella and M. Massink. *Automatic Verification of a Lip-Synchronisation Protocol Using UPPAAL*, In the 3rd International Workshop on Formal Methods for Industrial Critical Systems. Amsterdam, The Netherlands, 1998. Jan Friso Groote, Bas Luttik and Jos van Wamel (Eds.).

[13] B.McLaughlim, *Java and XML*, O'Reilly & Associated, Inc, 2000

[14] A.S.Tanenbaum, *Computer Networks third edition*, Prentice Hall, 1995

[15] H. Bowman, G. P. Faconti, M. Massink, *Specification and Verification of Media Constraints using UPPAAL*

[16] H. Bowman, H. Cameron, P. King, S. Thompson, *Mexitl: Multimedia in Executable Interval Temporal Logic*, 1998

[17] H. Bowman, J. Derrick (editors), *Formal Methods for Distributed Processing: a Survey of Object-oriented Programming*, Cambridge University Press, 2001

[18] UPPAAL home page, http://www.docs.uu.se/docs/rtmv/uppaal/

[19] G. Coulouris, J. Dollimore and T. Kindberg, *Distributed Systems: Concepts and Design Third Edition*, Addison-Wesley, 2001