UTILIZATION OF TIMED AUTOMATA AS A VERIFICATION TOOL FOR REAL-TIME SECURITY PROTOCOLS

A Thesis Submitted to
the Graduate School of Engineering and Sciences of
İzmir Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Computer Engineering

by
Burcu KÜLAHÇIOĞLU

July 2010
İZMİR

We approve the thesis of **Burcu KÜLAHÇIOĞLU**

Prof. Dr. Sıtkı AYTAÇ
Supervisor

Assoc. Prof. Dr. Ahmet KOLTUKSUZ
Co-Supervisor

Assist. Prof. Dr. Ufuk ÇELİKKAN
Committee Member

Assist. Prof. Dr. Gökhan DALKILIÇ
Committee Member

Assist. Prof. Dr. Serap ATAY
Committee Member

08 July 2010

Prof. Dr. Sıtkı AYTAÇ
Head of the Department
of Computer Engineering

Assoc. Prof. Dr. Talat YALÇIN
Dean of the Graduate School of
Engineering and Sciences

ACKNOWLEDGEMENTS

Foremost, I would like to gratefully acknowledge the supervision of my co-advisor, Assoc. Prof. Dr. Ahmet Koltuksuz, for his patience, motivation, enthusiasm and valuable guidance throughout my research.

I would like to thank to Prof. Dr. Sıtkı Aytaç for his support and encouragement. My sincere thanks go to Asst. Prof. Dr. Serap Atay for her advices through my study.

I would also like to express my sincere appreciation to Selma Tekir for her patience and advices. She was always there to discuss my studies and motivate me during my research.

I would like to thank Mutlu Beyazıt for his motivation and jokes that made the study more enjoyable. My room mate Gürcan Gerçek also deserves my thanks.

I would like to give my special thanks to my family for supporting me not only throughout my education but my whole life. I would also like to express my deepest gratitude for my colleague and fiancé Murat Özkan for his academic support and encouragement.

Finally, I would like to thank to The Scientific and Technological Research Council of Turkey (TUBITAK-BİDEB) for supporting me during my master of science degree education.

ABSTRACT

UTILIZATION OF TIMED AUTOMATA AS A VERIFICATION TOOL FOR REAL-TIME SECURITY PROTOCOLS

Timed Automata is an extension to the automata-theoretic approach to the modeling of real time systems that introduces time into the classical automata. Since it has been first proposed by Alur and Dill in the early nineties, it has become an important research area and been widely studied in both the context of formal languages and modeling and verification of real time systems. Timed automata use dense time modeling, allowing efficient model checking of time-sensitive systems whose correct functioning depend on the timing properties. One of these application areas is the verification of security protocols.

This thesis aims to study the timed automata model and utilize it as a verification tool for security protocols. As a case study, the Neuman-Stubblebine Repeated Authentication Protocol is modeled and verified employing the time-sensitive properties in the model. The flaws of the protocol are analyzed and it is commented on the benefits and challenges of the model.

ÖZET

ZAMANLI ÖZDEVİNİM KURAMININ GERÇEK ZAMANLI GÜVENLİK PROTOKOLLERİNİN DOĞRULANMASINDA KULLANIMI

Zamanlı özdevinim kuramı, klasik özdevinirler (otomata) kavramına zaman değişkenini ekleyerek bu modeli genişleten bir kuramdır. Doksanlı yılların başlarında öne sürülen zamanlı özdevinim kuramı, hem biçimsel diller hem de gerçek zamanlı sistem modelleme ve doğrulama alanlarında geniş ölçüde çalışılmaktadır. Zamanı sürekli bir değişken olarak ele alan zamanlı özdevinirler, doğru çalışması zaman kısıtlarına bağlı olan zaman kritik sistemler üzerinde model denetimine olanak sağlamaktadır. Bu uygulama alanlarından biri de güvenlik protokollerinin doğrulanmasıdır.

Bu tezde, zamanlı özdevinim kuramının incelenmesi ve güvenlik protokollerinin doğrulanmasında kullanımı amaçlanmaktadır. Bir durum çalışması olarak, Neuman-Stubblebine Tekrarlı Kimlik Denetimi Protokolü'nün, zamana bağlı özellikleri de dahil edilerek modellenmesi ve doğrulanması sunulmaktadır. Protokolün doğrulama sonuçları incelenerek modelin artı ve eksileri üzerinde yorumlara da yer verilmektedir.

TABLE OF CONTENTS

LIST OF FIGURES

Figure **Page**

LIST OF TABLES

Table **Page**

CHAPTER 1

INTRODUCTION

Real time systems are designed to perform a certain task within certain timeliness requirements, such as real time controllers, multimedia applications and communication protocols. However for some cases, some real time systems may not behave as intended which constitutes an important problem. Thus, it is needed to verify the correctness of the system using some systematic methods.

Formal verification methods can be used to model and analyze the behavior of a real time system to verify whether it meets the specified requirements. Formal verification can be performed using *Theorem Proving* methods that solve the general validity of a formula by using logical inference, or *Model Checking* methods that analyze a finite model of a system whether it fulfills the desired property.

In the literature, most verification methods (Pnueli 1977, Hoare 1978, Vardi and Wolper 1986) use the qualitative notion of time involving the partial ordering of the occurrence of events. However, the correctness of a real time system also depends on its quantitative timed properties. For example, for a communication protocol, it is more expressive to specify that "the response should be received in 5 time units after it has sent the message" than "the response should be received after it has sent the message".

Since there is a need for the time-sensitive models in formal modeling and verification, some untimed formalisms are extended with timing information to obtain a model closer to the real world (Reed and Roscoe 1988, Alur and Henzinger 1989, Alur and Dill 1994, Cerone and Maggiolio-Schettini 1999). Among these, timed automata formalism is the most commonly used model, having mature and efficient automatic verification tools and an easily understandable syntax and semantics.

Timed automata theory is proposed as an extension to the automata theoretic approach for the modeling of real time systems (Alur and Dill 1994). It is a class of automata extended with clock variables which model dense time. Timed automata theory has become an important research area and been widely studied in the context of both the theory of formal languages and verification of real time systems.

The theory of timed automata allows us to create models of real time systems which can be verified using model checking methods. There are some timed automata tools designed for this aim which have been used for several industrial and academic case studies. UPPAAL (Bengtsson and Larsen 1996), which is used in this thesis and KRONOS (Yovine 1997) are the most popular automatic verification tools among these. Similar to the other model checking methods, model checking with timed automata involves building a finite state model of a system and verifying a specified property by traversing through all reachable states. The method has the advantages of being fully automatic, generating a counter example in case of a negative result; nevertheless, it suffers from the state space explosion problem.

Timed automata model checking is used for the verification of many real time systems such as power controllers, gear controllers and audio-video protocols. One of the most important application areas of timed automata is the verification of the security protocols which aim secure communications over a network and used to provide a goal such as the authentication or distribution of cryptographic keys. As the use of computers and the internet is considerably increasing, the correctness of the security protocols is getting more important. However, most of the security protocols are found to be flawed which brings the network into an insecure state.

For a security protocol, the quantitative timing properties are critical and an intruder can attack the protocol by exploiting the timing and the flow of the messages. Hence, the timing properties should be employed into the security protocol model. Some recent studies concentrate on the analysis of the security protocols with timing information on the case studies of Needham-Schroeder and Yahalom authentication protocols (Corin, et al. 2004, 2007). The studies in (Jakubowska, et al. 2005, 2008) examine Kerberos, TMN, Neuman-Stubblebine and Andrew Secure RPC protocols by not modeling them directly as timed automata, but translating a language specification of a security protocol automatically to timed automata without integer variables.

This study utilizes timed automata as a verification tool for security protocols including timing information. Our case study, models Neuman-Stubblebine repeated authentication protocol directly with timed automata; then, verifies its security properties using UPPAAL timed automata tool based on the goals of an authentication protocol. Studying on a repeated authentication protocol gives the opportunity to include the key expiration time in the model, in addition to the time needed for cryptographic operations and timeout intervals.

First, the theory of timed automata is introduced. Besides the basic definitions and the decidability properties of the model which are fundamental for verification, some variants of timed automata proposed in some studies are mentioned briefly, that provides a better theoretical understanding and reasoning on the model.

As it will be explained in the theory of timed automata, the theoretical timed automata structures are hard to implement. Hence, the next chapter is devoted to the symbolic data structures and algorithms for timed automata implementation. This chapter provides an insight about the implementation of timed automata tools and how they perform automatic verification algorithmically.

Later, model checking with timed automata is explained and the UPPAAL tool is introduced since the modeling and verification with this tool is used in next chapters.

Next, the case study on Neuman-Stubblebine repeated authentication protocol is presented. It is a repeated protocol that consists of an initial and a subsequent authentication parts. Since the complete model of the protocol is so large that some problems arise in verification step, the initial and the subsequent authentication parts are analyzed individually. In this chapter, first, the modeling assumptions are given and it is explained how to model and abstract away the cryptographic details. Then, the construction of the automata models for the initial and subsequent parts are explained in detail. The verification of these parts are provided based on goals of an authentication protocol and the attacks on the protocol we can or cannot detect are explained. Later, these parts are merged to obtain a complete model for the protocol.

Later, an analysis of the case study is provided including the benefits and the challenges of the model and some possible extensions that can be proposed. The most serious problem, state space explosion problem, is defined and restrictions and the limitations applied to avoid it in the case study are explained exhaustively.

Finally, the study is concluded with the comments on the model and the further perspectives for the analysis of the security protocols using timed automata.

CHAPTER 2

TIMED AUTOMATA THEORY

2.1. Modeling Time

Although many fields of science and engineering involve time concept, in computer science, time is totally not considered or abstracted by modeling only the required features of it. However, some kind of time modeling is necessary in many areas of computing such as in hardware design, parallel processing and complexity calculations (Furia, et al. 2010). In this study, the focus is the representation of time for the modeling, specification and verification of real time systems whose correct functioning depends on their timed properties.

In the literature, most of the modeling and verification methods (Pnueli 1977, Hoare 1978, Vardi and Wolper 1986) involve qualitative notion of time rather than the quantitative notion. Qualitative notion only includes the relative ordering between events that specifies which event comes after the other. However, quantitative notion describes a distance between the events which is required for the verification of real time systems. For example, for a communication protocol, it is more expressive to specify that "the event A should occur in 5 time units after the occurrence of the event B" rather then "the event A should occur after the occurrence of the event B".

To meet the need for formalisms with quantitative timing information, some untimed formalisms are extended with quantitative notion. These formalisms are: the extensions of the *linear time logics* (LTL) (Pnueli 1977) and the *computational tree logic* (CTL) (Emerson and Clarke 1982), namely *metric temporal logic* (MTL) (Alur and Henzinger 1990), *timed temporal logic* (TPTL) (Alur and Henzinger 1989), *real-time temporal logic* (RTTL) (Ostroff 1989), *explicit-clock temporal logic* (XCTL) (Harel 1990), *real-time computation tree logic* (RTCTL) (Emerson, et al. 1990) and *timed computation tree logic* (TCTL) (Alur, et al. 1993a); timed process algebras such as timed *Communicating Sequential Processes* (TCSP) (Reed and Roscoe 1988); timed Petri Nets (Cerone and Maggiolio-Schettini 1999) which is more appropriate for work-

flow processes (Srba 2008) and timed automata (Alur and Dill 1994) which is an extension to the automata-theoretic approach.

For a real time system modeling technique, the representation of time is an important design issue. Different approaches for the representation of time may be more appropriate for different kinds of systems or modeling objectives (Alur and Henzinger 1992, Furia, et al. 2010).

- Many formalisms assume that the states of a system are observed only at integral times; although, in a real-time system, events may occur at any point. This approach uses *discrete time modeling*, which maps onto the integer numbers domain. Clocks tick at regular intervals and events can occur at each clock tick, at the multiples of ε. Here, it is important how to define the interval between two clock ticks, ε. If ε is defined to be too large, then the model is too coarse; on the other hand, if ε is defined to be too small, then the state space will be too large.
- *Fictitious clock* approach, introduces a special tick event into the model. Time is modeled as a global state variable on the domain of the natural numbers and it is divided into several ticks. The time delay between two events is measured by counting the number of these ticks. Hence, the exact time delay between two events cannot be measured.
- In *continuous time modeling*, time is modeled with non-negative real numbers. Hence, events can occur at any time and delays may be arbitrarily small. Dense time is strictly more expressive than discrete time. However, it gives rise to an uncountable state space. Timed automata uses *continuous time modeling* which is more faithful to the nature of real-time systems.

In timed automata, the passage of time is modeled by the real-valued clock variables, which record the elapsed time. All clocks are synchronized. They run at the same speed, having the same derivative with respect to time, which is assumed to be equal to 1. The absolute time is implicitly assumed by the model and the relative timing can be measured explicitly by testing the clock variables (Furia, et al. 2010).

The clocks can be tested and set to a value at the transitions. Note that we concentrate on the classical timed automata, in which only reset (setting to value 0) of a clock is allowed. Also, in timed automata, a transition from one state to another is assumed to be instantaneous; in other words, time passes only in states, not in edges.

In timed automata, it is possible to have a *zeno* run which allows infinite number of events occurring in a finite amount of time. The zeno-timelocks, where time cannot pass beyond a certain point, also constitute zeno behavior resulting in the performance of an infinite number of actions in a finite period of time. (Penczek and Polrola 2006, Bowman and Gomez 2006)

2.2. Timed Automata

The class of timed automata is a special subclass of hybrid automata (Alur, et al. 1993b) that include both discrete and continuous variables. Same as the classical automata, timed automata have finite number of states (locations) and edges. In addition, it has some number of real-valued clock variables which model the passage of time.

Alur and Dill who first proposed the timed automata theory (Alur and Dill 1994), defined some restrictions on the clock variables. In this model, the clocks can be reset only when an edge is taken and only clocks reset to the value 0 is allowed.

Definition (Clock Constraint): Let X be a set of clock variables. Then set $C(X)$ of clock constraints is given by the following grammar:

$$\phi \equiv x \leq k \mid k \leq x \mid x < k \mid k < x \mid \phi \wedge \phi \mid \phi \vee \phi \text{ where } x \in X,\ k \in \mathbb{N}.$$

Clock constraints can be guards on edges that control whether it is allowed to take the transition in the current time. The guards can also be associated with locations and are called *location invariants*. The automata allowing the use of location invariants, introduces a more intuitive notion of progress. These automata can be called as *timed safety automata* but generally referred as *timed automata*.

In the timed automaton given in Figure 2.1, the clock constraints $x > 3$, $x < 10$ and $y = 9$ are the guards on edge. To take the edge from s_0 to s_1, event a must occur and the clock variable x must have a value greater than 3. If this transition is enabled, the clock variable x is set to 0. $x < 7$ is an invariant and forces to take the edge from s_0 to $s_1$when the clock variable x has a value smaller than 7. Note that using this invariant does not have the same effect as having the guard $x > 3 \wedge x < 7$ on the transition.

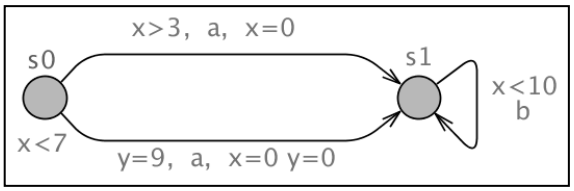

Figure 2.1. A Timed Automaton

Definition (Clock valuation): Clock valuation is a function $v : X \longrightarrow \mathbb{R}^+$, that assigns every clock to a real variable. Let v be a clock valuation. Then,

- Initially, $v_0(x) = 0$ for all x.
- Clock reset: $\begin{cases} v[Y = 0](x) = 0, & x \in Y \\ x & otherwise \end{cases}$
- Clock increment: $(v + d)\ flow\ of\ time\ (d\ units)$: $(v + d)(x) = v(x) + d$

The interpretation $v \vDash c$ means that the valuation v satifies the constraint c.

Definition (Timed Automaton): A timed automaton is a tuple $\langle \Sigma, S, S_0, S_F, C, E \rangle$ where

- Σ is a finite event alphabet
- S is a finite set of states
- $S_0 \subseteq S$ is a set of start states
- $S_F \subseteq S$ is a set of final (accepting) states
- C is a finite set of clocks
- $E \subseteq S \times S \times \Sigma \times 2^C \times \Phi(C)$, $\phi \in \Phi(C)$ are the edges where
  $\phi := x \leq k \mid k \leq x \mid x < k \mid k < x \mid \phi \wedge \phi \mid \phi \vee \phi$ with $x \in X,\ k \in \mathbb{N}$.

A transition can be written as $s \xrightarrow{g,a,r} s'$ whenever $(s, s', a, r, g) \in E$, where g is a guard, a an event and r a subset of clocks to be reset.

The following subclasses of automata are defined depending on the strictness on the inequalities of the clock constraints:

- ✓ *Open* timed automata can only have the clock constraints given as $\phi_1 := x < k \mid k < x \mid \phi \wedge \phi \mid \phi \vee \phi$. They are "acceptance robust"; when they accept a time trace, they accept also the neighbor timed traces.
- ✓ *Closed* timed automata can only have the clock constraints given as $\phi_2 := x \leq k \mid k \leq x \mid \phi \wedge \phi \mid \phi \vee \phi$. They are "rejection robust"; where rejected traces are robust under small perturbations.

A run of timed automaton A has the following form: $(s_0, v_0) \xrightarrow{t_0} (s_0, v'_0) \xrightarrow{a} (s_1, v_1) \xrightarrow{t_1 - t_0} (s_1, v'_1) \xrightarrow{a_1} \dots \dots \xrightarrow{a_{n-1}} (s_n, v_n)$, where each pair (a, t) is a timed event with $t \in \mathbb{R}$, which is the timestamp of the event $a \in \Sigma$. A run is accepted if $s_n \in S_F$.

Definition (Timed word): A timed word or a timed trace is a finite sequence of timed events with non-decreasing timestamps having the form $(a_0, t_0), (a_1, t_1), \dots, (a_n, t_n)$ with $t_i \leq t_{i+1}$ where $a_i \in \Sigma$ and $t_i \in \mathbb{R}$.

Timed words can be obtained from the runs of the timed automata where accepting timed words make up the language $L(A)$. An example timed word is given below, for the automaton in Figure 2.1.

$$s_0 \xrightarrow{\delta(3.8)} s_0 \xrightarrow{\delta(a)} s_1 \xrightarrow{\delta(2)} s_1 \xrightarrow{\delta(b)} s_1 \xrightarrow{\delta(3.2)} s_1 \xrightarrow{\delta(a)} s_0$$

x:	0	3.8	0	2	2	5.2	0
y:	0	3.8	3.8	5.8	5.8	9	0

Timed word (timed trace): $(a, 3.8), (b, 5.8), (a, 9)$

In a run, $\delta(a)$ with $a \in \Sigma$ is called an action transition and $\delta(t)$with $t \in R$ is a delay transition. If α in $\delta(\alpha)$ is equal to 0, then two consecutive transitions can be executed without time passing in between. Such runs are called *weakly monotonic*. But, it is sometimes more convenient to restrict the runs to contain non-zero passing times only, which are called *strongly monotonic*. Simply, weakly monotonic time traces ($WMTT$) allow several events may share the same time stamp where strongly monotonic time traces ($SMTT$) allow no two events happen at the same time (Penczek and Polrola 2006).

2.3. Timed Regular Languages

Inspiring from the theory of *regular languages* accepted by finite state automata, *timed regular languages* can also be defined for timed automata. A *timed language L* is a collection of timed words. *L* is timed regular if there is a timed automaton whose accepting timed traces (timed words) make up the timed language *L* (Alur, et al. 2004, Asarin, et al. 2002).

Definition (Timed Regular Language): A timed language L is a timed regular language iff $L = L(A)$ for some timed automaton A.

Definition (Untiming): If $u = \langle (a_1, t_1), (a_2, t_2), \dots, (a_n, t_n) \rangle$, then $untime(u) = a_1 a_2 .. a_n$.

Theorem: Given a timed automaton $A = \langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{S}_\mathrm{F}, \mathrm{C}, \mathrm{E} \rangle$, there exists an automaton over Σ which accepts $Untime[L(A)]$.

This theorem leads to the fact that, if L is a timed regular language, then $untime(L)$ is a regular language. An example not timed regular language can be given as a language of timed words where every a symbol is followed by some b symbol after a delay of 1. A timed automaton cannot be generated for that language since there can be unbounded number of a symbols in a timed word that needs the use of infinite number of clocks. Another example for a not timed regular language is the untimed language $\{ a^n b^n \mid n \; is \; an \; integer \}$ since it is not even a regular language.

The next theorem gives the closure properties of timed regular languages for union and intersection operations. Similar to the untimed automata, the union timed automata can be constructed by taking the disjoint union of all the automata, and the intersection automata can be obtained by constructing the product of the automata.

Theorem: A set of timed regular languages is closed under union and intersection.

However, the class of timed automata is not closed under complementation, which is included in next section.

2.4. Decidability and Complexity of Timed Automata

A certain property is *decidable* for a formal language if there is a procedure that can determine whether the property holds in the model. The amount of memory and the time required for the algorithm that solves this *decidability* problem gives the *complexity* of the problem.

For timed automata, the *emptiness*, *universality* and *language inclusion* problems are the most studied decision problems since they are also the fundamental problems for verification. The solution of the language inclusion problem requires the

complementability (therefore, *determinizability*) of timed automata, which also constitute important decision problems.

2.4.1. Emptiness Problem

The emptiness problem asks whether the language generated by a formal model is empty. The following is the definition of the emptiness problem for timed automata.

Definition (Emptiness Problem): It is the problem of "given a timed automaton A, is the set of timed traces of A empty?".

In a verification task, given an implementation and a specification, the reachability problem is used to test whether a state which satisfies the specification is reachable. The emptiness problem is fundamental for verification tasks since it is reducible to the reachability problem that tests whether a state can be reached in a model.

However, the solution of this problem is more cumbersome than untimed automata. Since the configurations of timed automata are infinite, even a very simple automaton generates infinitely many reachable states. Figure 2.2 gives a simple timed automaton giving rise to an infinite transition system where naive explicit state search is not reasonable. The idea is constructing a model on which finite state analysis is possible so that we can examine the decidability of the reachability problem.

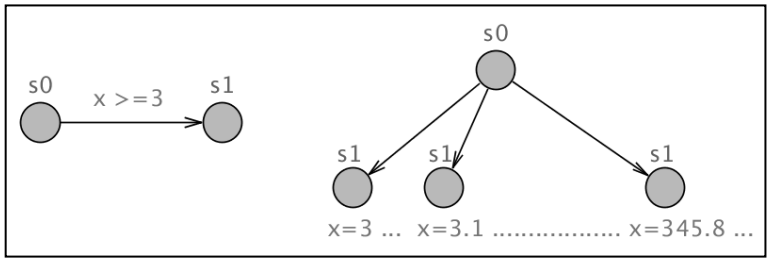

Figure 2.2. Illustration of an Infinite State System

The decidability of the emptiness problem for timed automata is proved by constructing a transition table that mimics the runs of A. To achieve this goal, state space is partitioned into finitely many equivalence classes so that equivalent states

exhibit similar behaviors. Infinite number of clock valuations are partitioned into finitely many clock regions, whose definition is given below.

Definition (Clock Regions): Let $n, K \in N$ be fixed. Define an equivalence relation $\sim$ on $(\mathbb{R}^+)^n$. $\sim$ partitions into finitely many equivalence classes called regions (Alur, et al. 1990, Alur and Dill 1994).

Let for any $x \in \mathbb{R}^+$, $fract(x)$ denote the fractional part of x and $\lfloor x \rfloor$ denote the integral part of x; that is $x = \lfloor x \rfloor + fract(x)$. The equivalence relation $\sim$ is defined over the set of all clock interpretations for C.

$(x_1 \ldots, x_n) \sim (x'_1 \ldots, x'_n)$ if

- ✓ for all $1 \leq i \leq n$
  - either $\lfloor x_i \rfloor = \lfloor x'_i \rfloor$
  - or both x_i and x'_i are greater than K (exceeds c_x)
- ✓ for all $1 \leq i, j \leq n$ with $x_i , x'_i \leq K$
  - $fract(x_i) \leq fract(x_j) \leftrightarrow fract(x'_i) \leq fract(x'_j)$
- ✓ for all $1 \leq i \leq n$ with $x_i \leq K$
  - $fract(x_i) = 0 \leftrightarrow fract(x'_i) = 0$

Figure 2.3 shows the clock regions of a timed automaton with two clock variables x and y, having "2" as the maximum clock constant in the clock constraints for x and "1" for y.

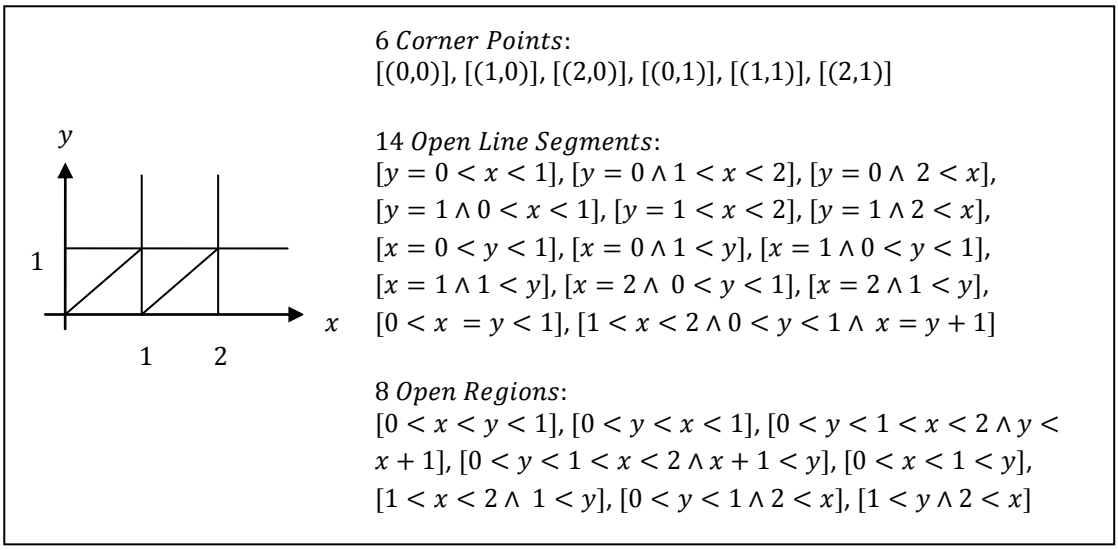

Figure 2.3. Clock Regions of a Timed Automaton

A region automaton $R(A)$ can be constructed that is equivalent to the semantics of a timed automaton A with respect to reachability. This region automaton $R(A)$ records the state of A, and the equivalence class of the current values of the clocks. Before formally defining a region automaton, let us define the time successor of a region which is simply the region that can be reached by the passage of time.

Definition (Time Successor): A clock region α' is a time successor of a clock region α iff for each $v \in \alpha$, there exists a positive $t \in \mathbb{R}$ such that $v + t \in \alpha'$ (Alur and Dill 1994).

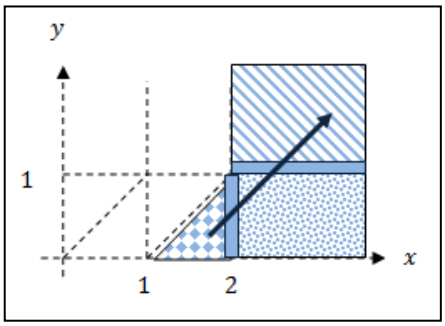

Figure 2.4. Time Successors of a Clock Region

Figure 2.4 shows the time successors of the clock region $[\,(1 < x < 2), (0 < y < x - 1)\,]$, which are, other than itself, $[\,(x = 2), (0 < y < x - 1)\,]$, $[\,(x > 2), (y = 1)\,]$ $[\,(x > 2), (0 < y < x - 1)\,]$, $[\,(x > 2), (y > 1)\,]$. This means, when the clock valuation is in that region, the passage of time can bring us to the one of these time successors. Time successors of a region can be easily found by drawing a line parallel to $x = y$ line, starting from this region to the upwards.

Definition (Region Automata): For $A = \langle \Sigma, S, S_0, S_F, C, E \rangle$, a region automaton $R(A)$ is a transition table over the alphabet Σ:

- ✓ the states: $\langle s, \alpha \rangle$
- ✓ the initial states: $\langle s_0, [v_0] \rangle$
- ✓ R(A) has an edge $\langle \langle s, \alpha \rangle, \langle s', \alpha' \rangle, a \rangle$ iff there is an edge $\langle s, s', a, \lambda, \delta \rangle \in E$ and a region α''; such that

  (i) α'' is a time successor of α,

  (ii) α'' satisfies,

  (iii) $\alpha' = [\lambda \rightarrow 0]\,\alpha''$.

Figure 2.5 gives the region automaton for the given timed automaton. The initial state of the region automaton is the state $s0$ where both the clocks are equal to 0. The transition from $s0$ to $s1$ can be taken when x is greater than 0. Notice that, the greatest constant in the time constraints is 1. Hence, this constant will be considered while generating the regions. In order for the clock x to get greater than 0, some time should elapse in state $s0$. So, the event a can taken when $0 < y < 1$, $y = 1$ or $y > 1$. Since the clock x is reset at the transition, we can reach to the state $s1$ having the clock regions $0 = x < y < 1$, $x = 0$ and $y = 0$, $x = 0$ and $y > 1$. The rest of the region automaton is constructed in a similar way by considering the time successors and transitions for the states in region automata.

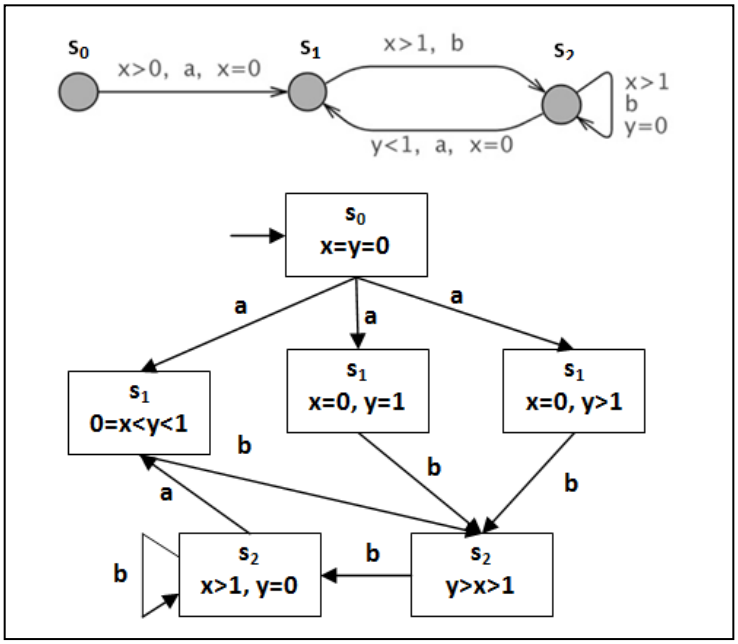

Figure 2.5. Region Automaton of a Timed Automaton

The region automaton of a timed automaton A can recognize the language $Untime[L(A)]$. Hence, the emptiness problem of a timed automaton may be examined by checking the emptiness of its region automata. In order to calculate the complexity of this problem, we should calculate the number of clock regions in a region automaton.

Let c_x be the largest constant in the clock constraints of the clock $x \in C$ where C is the set of all clocks. For each region, there is one clock constraint from the set $\{x = c \mid c = 0,1,..,c_x\} \cup \{c - 1 < x < c \mid c = 0,1,..,c_x\} \cup \{x > c_x\}$. The number of ways to choose this value for each clock is $\prod_{x \in C}(2c_x + 2)$. The ordering of the

fractional parts of the clocks may be chosen in $|C|!$ ways (For example, for 3 clocks we may have 3! permutations as $x < y < z$, $x < z < y$, $y < x < z$, $y < z < x$, $z < x < y$, $z < y < x$). The fractional part of a clock may also be equal to the fractional part of its predecessor (such as $x \leq y < z$), the number of ways to choose this is bounded by $2^{C.}$

Lemma: Given a timed automaton A, the number of regions of $R(A)$ is bounded by $O[\ |C!|\ .2^{|C|}.\ \prod_{x\epsilon C}(2c_x + 2)]$.

Theorem: Given a timed automaton A, the emptiness of $L(A)$ can be checked in time $O[(|S| + |E|).|C!|.2^{|C|}.\ \prod_{x\epsilon C}(2c_x + 2)]$.

Theorem: The problem of deciding the emptiness of the language of a given timed automaton A, is $PSPACE - complete$ (Alur, et al.1990, Alur and Dill 1994).

Since the number of regions is exponential in the number of clocks for the region automata, it suffers from a combinatorics explosion. Hence, this structure is not feasible to implement. Instead of this, symbolic data structures and algorithms are used for implementation of timed automata to perform reachability tests easier on them, as explained in Chapter 3.

2.4.2. Universality Problem

Definition (Universality problem): It is the problem that asks "given a timed automaton A, does it accept all timed traces?".

Universality problem for timed automata is undecidable, which is proved for the classical timed automata in general sense, by reducing the problem to the halting problem of two-counter machines (Alur and Dill 1994). Before giving the proof, let us define the two counter machines. Then we will examine, given a 2-counter nondeterministic machine, how to construct the corresponding timed automaton.

Definition (Two-counter machine): A nondeterministic 2-counter machine M is a triple $(\{\ b_0, b_1\ , \dots , b_k\ \}, C, D)$ with a sequence of n instructions and two counters C, D.

In a two-counter machine, each b_i **(i)** can increment or decrement one of the counters and jumps nondeterministically to one of the possible next instructions, or **(ii)**

tests one of the counters for emptiness and jumps unconditionally to the next instruction. A configuration of M is a triple (b_i, c, d) where $b_i \in \{ b_0, b_1, \dots, b_k \}$, $c \in C$, $d \in D$. A halting computation of M is a finite sequence of configurations starting with $(b_0, 0, 0)$ and ending with $(b_k, -, -)$. The problem of deciding whether a given nondeterministic 2-counter machine has a halting computation, is $NP - hard$.

Theorem: Given a timed automaton A, universality problem for A is undecidable.

Proof: The universality problem for timed automata is examined by reducing this problem to the halting problem of two-counter machines. The automaton is encoded on the alphabet $\{b_1, \dots, b_n, a_1, a_2\}$, where $\sigma = b_{i_1} a_1 a_2 b_{i_2} a_1 a_2 \dots$ is a halting computation of M, having $\langle i_1, c_1, d_1 \rangle$, $\langle i_2, c_2, d_2 \rangle$. The strings (σ, τ) is in a timed language L_{undec} such that, in this string, the subsequence of σ, corresponding time interval $[j, j+1)$ encodes the j^{th} configuration in a way that, for all $j \geq 1$,

- ✓ if $c_{j+1} = c_j$ then for every a_1 at time t in the interval $(j+1, j+2)$, there is an a_1 at time $t+1$
- ✓ if $c_{j+1} = c_{j+1}$ then for every a_1 at time t in the interval $(j+1, j+2)$ except the last one, there is an a_1 at time $t-1$
- ✓ if $c_{j+1} = c_{j-1}$ then for every $a_1$at time t in the interval $(j, j+1)$ except the last one, there is an a_1 at time $t+1$
- ✓ similar requirements hold for a_2

Let us say that given a 2-counter machine M, the words corresponding to the halting computations of M make up the language L_{undec}. The idea is to construct the automaton A_{undec} (this is the disjunction of several automata each demonstrating an unacceptable behavior for a two counter machine. These are the automata **(i)** A_0 that accepts (σ, τ) for some $j \geq 1$, there is no b symbol at time j, or the subsequence of σ in the interval $(j, j+1)$ is not of the form $a_1^* a_2^*$, the string having the substring $a_2 a_1$, **(ii)** A_{init} accepts (σ, τ) iff the substring of σ corresponding [1, 2) is not b_1, meaning that $\sigma_1 \neq b_1$ or $\tau_1 \neq 1$ or $\tau_2 < 2$, **(iii)** for each $1 \leq i \leq n$, we construct an automaton A_i which checks whether the next instruction is valid with respect to current instruction), accepts the complement of L_{undec}. Hence, if the two-counter machine does not halt, we say that L_{undec} is empty, implying that the automaton is universal (Alur and Dill, 1994).

Special Cases:

There are some studies on universality problem with some restrictions on the timed automata where some of them yield to the decidability of the problem. Generally, in the verification studies, no such restrictions are applied on the timed automata model. However, we briefly cover the theorems stating the decidability results for some cases to see a direction of research on timed automata and have a better insight of the model.

Considering the time traces being weakly or strongly monotonic, the decidability of the universality problem is dependent upon the open and closed subclasses of timed automata. The theorems below, give the decidability results of open and closed timed automata over $WMTT$ and $SMTT$. The proofs for these theorems use the digitization properties of timed automata and can be found in (Ouaknine and Worrell 2003b).

Theorem:

- **(i)** The universality problem for open and closed timed automata over $SMTT$ is undecidable.
- **(ii)** The universality problem for open timed automata over $WMTT$ is decidable.
- **(iii)** The universality problem for closed timed automata over either $WMTT$ or $SMTT$ is undecidable.

Universality problem for timed automata is also studied by adding some restrictions on the resources of a timed automaton which are the number of states, size of event alphabet and clock constraints. The study in (Adams, et al. 2007) examines the problem with minimal resources. The following theorems give these decidability results.

Theorem:

- **(i)** Over $WMTT$, the universality problem is undecidable for timed automata with a single state, a single-event alphabet, and clock constants 0 and 1 only.
- **(ii)** Over $SMTT$, the universality problem is undecidable for timed automata with a single state, a single-event alphabet, and clock constants 1, 2 and 3 only.

The next theorems state that, for timed automata over finite words, the one-clock universality problem is decidable with non-primitive recursive complexity. However, if ε-transitions or non-singular post-conditions are allowed, then the one-clock

universality problem is undecidable over both finite and infinite words. The proofs for these theorems can be found in (Adams, et al. 2005, 2007). The universality for the timed automata with one clock and ε-transitions is undecidable.

(iii) The universality for the timed automata with one clock and with non-singular post-conditions is undecidable.

(iv) The universality for the timed automata with a single clock is decidable and has non-primitive recursive complexity (the problem does not lie in the complexity class $TIME(f(n))$ for any primitive recursive function $f(n)$) over finite timed words.

(v) The universality for one-clock nondeterministic timed automata is undecidable over infinite timed words (Lasota and Walukiewicz 2008).

2.4.3. Language Inclusion Problem

Definition (Language Inclusion Problem): Given two timed automata A_1 and A_2, it is the problem of checking whether $L(A_1) \subseteq L(A_2)$.

Language inclusion problem is important for the verification of systems. In order to perform verification of a system, the behavior of the implementation and the requirements that the system should satisfy can each be represented as a set of traces. Then, it can be checked whether the implementation satisfies the specification, in other words, whether the set of specification traces includes the set of implementation traces. Given a specification A_S and an implementation A_I, the implementation meets its specification iff $L(A_I) \subseteq L(A_S)$.

Corollary: The language inclusion problem for the timed automata is undecidable.

The undecidability of the problem can be easily seen by reducing the universality problem to the language inclusion problem. The automaton A is universal iff $L(A_{univ}) \subseteq L(A)$.

Special Cases:

Similar to the universality problem, some studies yield decidability of the language inclusion problem for special cases.

Theorem:

(i) The language inclusion problem over $WMTT$ is only decidable when B is a closed timed and A is an open timed automata.

(ii) The language inclusion problem over $SMTT$ is undecidable in all cases (Ouaknine and Worrell 2003).

The problem also becomes decidable with some restrictions. The proofs for the following theorems can be found in (Ouaknine and Worrell 2004).

Theorem: The language inclusion problem asking whether $L(B) \subseteq L(A)$ is decidable where

(i) the timed automaton A has at most one clock.

(ii) the timed automaton A has 0 as the only constant used in the clock constraints.

In addition to these results, this problem is proved to be decidable for some subclasses or variants of timed automata which are given in sections 2.5 and 2.6.

2.4.4. Complementability, Determinizability and Minimization

2.4.4.1. Complementability

Complementing an untimed deterministic finite state automaton is simple and the complement automaton can be easily obtained by just exchanging the accepting and non accepting states. However, this is not the case for a timed automaton.

Theorem: The class of timed regular languages is not closed under complementation.

Proof: Given two timed automaton A and B, $L(B) \subseteq L(A)$ iff the intersection of $L(B)$ and the complement of $L(A)$ is empty. Assume that the set of timed automata is closed under complementation. Then, $L(B) \nsubseteq L(A)$ iff there is an automaton C such that $L(B) \cap L(C)$ is nonempty, but $L(A) \cap L(C)$ is empty. This follows that the complement of the inclusion problem is recursively enumerable, contradicting the undecidability of the inclusion problem (Alur and Dill 1994).

The study in (Tripakis 2006) examines the problems asking whether a timed automaton is complementable, determinizable, or minimizable; which also ask for a solution automaton if the answer is "yes". The proofs are based on the reduction of the problems to the universality problem.

Theorem: The problem "Given a timed automaton A does there exist a timed automaton such that $(B) = \overline{L(A)}$? If so, construct such B" is not Turing computable. In other words, there does not exist an effective procedure which given a timed automaton A it constructs B such that $L(B) = \overline{L(A)}$ if it exists, or says "no" if such a timed automaton does not exist.

Proof: Given a timed automaton A, if its complement timed automaton B exists, then we can compute B. $L(A)$ will be universal iff $L(B) = \emptyset$. If B does not exist, then $L(A)$ is not universal, because the empty language can be accepted by a timed automaton with no accepting states.

2.4.4.2. Determinizability

Definition (Deterministic timed automata): A timed automaton A is deterministic iff **(i)** the set of initial locations is a singleton, **(ii)** $\forall s \in S$ and $\forall a \in \Sigma$, if there are two edges $(s, s_1, a, R_1, \varphi_1)$ and $(s, s_2, a, R_2, \varphi_2)$, then the guards φ_1 and φ_2 are mutually exclusive.

If A is a deterministic timed automaton, then for every timed word $\rho \in L(A)$, there is a unique run over A accepting it. Figure 2.6 gives an example deterministic timed automaton.

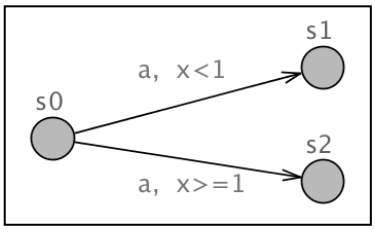

Figure 2.6. A Deterministic Timed Automaton

Theorem: There does not exist an effective procedure which given a timed automaton A outputs a deterministic timed automaton B such that $L(B) = L(A)$ if B exists, or says "no" if such a timed automaton does not exist.

Let us assume that such a procedure exists. **(i)** If B exists, since it is deterministic, one can construct the automaton C which is $\overline{L(B)}$, by interchanging the accepting and non-accepting states. Now, $L(A)$ is universal iff $L(C) = \emptyset$. **(ii)** If B does not exist, then $L(A)$ is not universal since the universal language can be accepted by a deterministic timed automaton with a single accepting state, no clocks, and a self loop for each letter in Σ (Tripakis 2006).

Figure 2.7 gives an example of timed automaton that cannot be complemented or determinized. Since there is no bound on the number of a's that can occur in any time interval, any timed automaton capturing the complement of $L(A)$ would require an unbounded number of clocks to keep track of the times of all the a's within the past one time unit (Alur and Dill 1994).

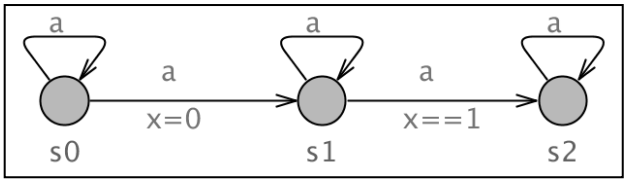

Figure 2.7. An Automaton that cannot be Determinized and Complemented

2.4.4.3. Minimization

Minimization problems for a timed automaton A, ask whether it is possible to create an equivalent timed automaton B with a reduced number of clocks or lessened magnitude of clock constraints.

Theorem: There does not exist an effective procedure which given a timed automaton A with n clocks (where $n \geq 2$), outputs a timed automaton B with $n - 1$ clocks such that $L(B) = L(A)$ if it exists, or says "no" if such a timed automaton does not exist.

Proof: The universality problem can be reduced to the minimization problem. Given A, if such an automaton B exists, construct a timed automaton having $n-1$ clocks. If B exists with zero clocks, check whether the untimed language of B is equal to Σ^*. Since it has no clock, it has no constraints on them. If $L_U(B) = \Sigma^*$, then $L(A) = L(B) = U$. If such B does not exist, A is not universal since universal automaton can be reduced to an automaton with no clocks.

Theorem: There does not exist an efficient procedure which given a timed automaton A with c as the maximum constant in clock constraints, and outputs a timed automaton B with $c-1$ as the maximum constant in clock constraints such that $L(B) = L(A)$ if it exists, or says "no" is such a timed automaton does not exist.

Proof: If exists such an automaton, size of the constraint can be reduced until it is zero or to a constant that cannot be reduced any more. To construct B having constants at most zero can be reduced to the problem in previous theorem (Finkel 2006).

If it was possible to minimize a timed automaton into a reduced model, than we could be able to perform verification easier which would further improve the efficiency of the timed automata model.

2.5. Variants of Timed Automata

In the previous sections, the classical definition of timed automata is given. Then, its closure and decidability results are explained. It can be seen that, although it is a powerful tool for real time verification, it has some drawbacks in the formal language concepts. It is not closed under complementation and determinizability is undecidable. Clock reduction is possible for some timed automata but the possibility is also undecidable. For the verification aspects, the emptiness problem is decidable but universality problem is not decidable in general.

In the literature, some variants of timed automata are proposed and studied whether they yield better decidability results or make up a determinizable class of timed automata. They are some modified or restricted classes of the classical timed automata, some of which lead to the decidability of some problems.

Before going on these variants of the model, it is important to note that the timed automata tools implement and case studies use the classical timed automata model.

Although these proposed variants are not implemented, we give brief explanation of them to provide a better understanding of the timed automata model.

2.5.1. ε-transitions on Timed Automata

A silent action ε is a non observable event and the transition labeled with that action is called a *silent transition*. Timed automata with ε-transitions, shown as TA_ε, are strictly more expressive than the timed automata without ε-transitions, TA. Figure 2.8 gives an example TA_ε that accepts the timed words with even timestamps, that cannot be expressed with a timed automaton (Bérard, et al. 1998) (Note that UPPAAL tool that is used for drawing this figure, in fact does not accept the events Σ (the event alphabet) and ε.).

Theorem: Given a TA_ε, it is undecidable to determine whether there exists a TA B such that $L(A) = L(B)$.

For a TA_ε, the complementability, determinizability and computability of the minimal number of clocks needed to recognize its ε-timed regular language are also undecidable (Bouyer, et al. 2009).

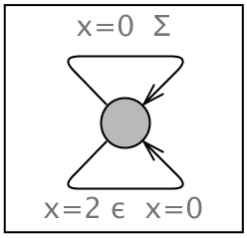

Figure 2.8. A TA_ε that cannot be Expressed with a TA

2.5.2. Diagonal Constraints and Updates on Timed Automata

In classical timed automata, only reset to 0 is allowed as clock update. The studies in (Bouyer, et al. 2000, 2004) concentrate on *updatable timed automata* which allows for different kinds of updates that may allow simpler and more concise representations of some real-time systems. They examine the decidability results of these models by considering the *diagonal* properties of timed automata.

A timed automaton is *diagonal-free* if its clock constraints are defined by: $\varphi ::= x \sim c \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$ where $x \in X,\ c \in Q$ and $\sim \in \{<, \leq, =, \neq, >, \geq\}$. Diagonal automata can also contain any sub formula of the form $x - y \# t$ where x and y are clocks, and $\# \in \{<, =, >\}$. Note that the diagonal-free automata have the same expressive power as the automata with diagonal constraints, since there exists a diagonal free timed automaton equivalent to a diagonal automaton (Alur and Dill 1994, Bérard, et al. 1998).

The summary of the decidability results for updatable timed automata are given in Table 2.1. The decidable classes are not more powerful than classical automata, and even bisimilar automata for them can be constructed.

Table 2.1. Decidability of Diagonal-free and Diagonal Automata with Updates

Updates	Diagonal-free Constraints	Diagonal Constraints
$x = c, x = y$	PSPACE-Complete	PSPACE-Complete
$x = x + 1$		Undecidable
$x = y + c$		
$x = x - 1$	Undecidable	
$x < c$	PSPACE-Complete	PSPACE-Complete
$x > c$		Undecidable
$x \sim y + c$		
$y + c < x < y + d$		
$y + c < x < z + d$	Undecidable	

2.5.3. Robust Timed Automata

The idea for *robust timed automata* comes from the real-time systems that cannot realize the exact time t but can have a physical error ε which are insensitive to small errors. Robust timed automata accept tubes rather than trajectories where a tube is defined as an open set of trajectories that consists of a bundle of sufficiently similar trajectories. If a tube includes a trajectory, then it also includes its neighbors; hence the system accepts trajectories of time $(t - \varepsilon, t + \varepsilon)$ (Gupta, et al. 1997, Henzinger and Jean-Francois 2000).

Similar to the classical automata, the emptiness problem of robust timed automata is $PSPACE$ complete and the robust universality problem is undecidable.

2.5.4. Event Clock Automata

Event clock automata (ECA) is a determinizable subclass of timed automata. In an ECA, the clocks can record the time elapsed after an event's last occurrence, or the clocks can be used to predict the time of the next occurrence of an event. The first subclass of the event clock automata is called *event-recording automata (ERA)* where the latter one is called *event-predicting automata (EPA)* (Alur, et al. 1999).

Although nondeterministic automata are more expressive than its deterministic counterpart, nondeterministic event clock automata and the deterministic one are equally expressive. For every event-clock automaton A, there is a deterministic event-clock automaton that is equivalent to A.

The event clock automata have a decidable language inclusion problem, it is $PSPACE - complete$ to check whether $L(B) \subseteq L(A)$ for two event-clock automata.

2.5.5. Alternating Timed Automata

Alternating timed automata (ATA) is obtained by introducing universal transitions in the same way as it is done for *Alternating Finite Automaton (AFA)*. It is known that the class of timed automata is not closed under complementation. Instead of restricting the model to deterministic timed automata or event-clock automata by restricting reset operations, alternating timed automata provides a model that is closed under Boolean operations.

ATA is defined by the tuple $A = \langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{S}_\mathrm{F}, \mathrm{C}, \mathrm{E} \rangle$ with $\delta: Q \times \Sigma \times \Phi \longrightarrow \mathcal{P}(Q \times \{nop, reset\})$ is a finite partial function and for every q and a, the set $\{\, [\sigma]: \delta(q, a, \sigma)\ is\ defined \,\}$ gives a finite partition of $(\mathbb{R}_+)^C$.

Note that the class of languages recognized by one-clock alternating timed automaton is incomparable with the class of languages recognized by timed automata. The emptiness problem is decidable for one-clock ATA. However, the emptiness over infinite words for one-clock ATA is undecidable (Lasota and Walukiewicz 2008).

2.5.6. Integer Reset Timed Automata

Integer Reset Timed Automata ($IRTA$) are a syntactic subclass of timed automata where clock resets are restricted to occur at integer valued time points. This subclass of timed automata is shown to be useful since they used for verification of some real time systems as given in (Suman, et al. 2008).

$IRTA$ can be determinized to 1-clock deterministic $IRTA$ which is complementable. It also yields to a decidable language inclusion problem such that if B is a timed automaton and A is an $IRTA$, than $L(B) \subseteq L(A)$ is decidable.

The study in (Suman and Pandya 2009) examines $IRTA$ with silent transitions, $\varepsilon - IRTA$ which is closed under complementation. However, it is undecidable to determine whether for a given timed automaton there exists an $\varepsilon -$ IRTA equivalent to it.

2.6. Digitization of Timed Automata

We know that timed automata uses continuous time modeling which is more faithful to the nature of real-time systems. Although timed automata tools, most case studies and our study involve dense time, this section gives brief information on the digitization concepts since they are important in terms of leading the decidability of language inclusion problem.

Digitization techniques reduce the dense-time model to discrete time. Although dense time is more expressive, digitization makes the model checking easier. The "digitization" concepts are introduced and defined by Henzinger et al. (Henzinger, et al. 1992b, Bosnacki 1999).

Definition (Digitization): Given $x \in \mathbb{R}$ and $\varepsilon \in (0,1]$, let $[x]_\varepsilon = \lfloor x \rfloor$ if $x \leq \lfloor x \rfloor + \varepsilon$, otherwise $[x]_\varepsilon = \lceil x \rceil$. For any precisely timed sequence $\rho = (\sigma, T)$ and $\varepsilon \in (0,1]$, let the ε-digitization $[\rho]_\varepsilon: (\sigma_0, [T_0]_\varepsilon) \rightarrow (\sigma_1, [T_1]_\varepsilon) \ldots$. For example, let us have a precisely timed trace $p: (p, 4.7) \rightarrow (q, 0.8) \rightarrow (p, 3.3)$. Then, this trace will have 0.5-digitization $[\rho]_{0.5}: (p, 5) \rightarrow (q, 1) \rightarrow (p, 3)$.

Digitized timed automata only consider integral values for the clocks. Figure 2.9 shows the time regions constructed for a digitized automaton.

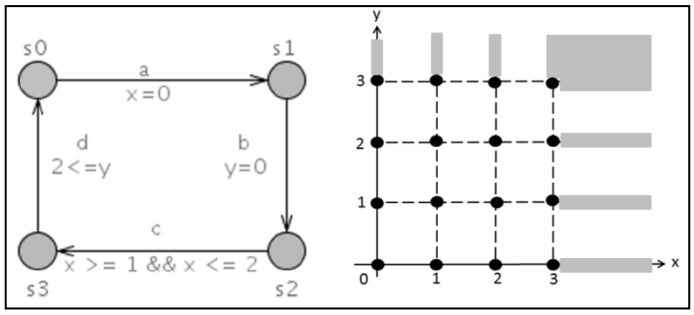

Figure 2.9. Time Regions for a Digitized Timed Automaton

Definition (Digitizability): A dense time property is *digitizable* iff it is closed both under digitization and under inverse digitization. A set of timed traces T is *closed under digitization* if for any $0 \leq \varepsilon \leq 1$, $[T]_\varepsilon \subseteq T$. T is *closed under inverse digitization* if whenever a timed trace $u \in TT$ such that $[u]_\varepsilon \in T$ for all $0 \leq \varepsilon \leq 1$, then $u \in T$.

Theorem: Let the set of timed traces T be closed under digitization, T' be closed under inverse digitization and $\mathbb{Z}T$ be the set of all integral timed traces of T (those timed traces of T whose events are integral time stamps). Then, $T \subseteq T'$ if and only if $\mathbb{Z}T \subseteq \mathbb{Z}T'$ (Ouaknine and Worrell 2003a).

This theorem yields a decidable language inclusion problem and underlines the importance of being able to determine the digitization properties of timed automata. The problem of closure under digitization is decidable for both open and closed timed automata; however, the problem of closure under inverse digitization is undecidable for closed timed automata.

CHAPTER 3

IMPLEMENTATION OF TIMED AUTOMATA

As explained in the previous chapter, the class of timed automata is proved to be decidable for real time system verification by using region-based technique. However, this technique is hard to implement since it gives rise to an explosion in the number of regions depending on not only the number of components in a system but also the largest time constant and the number of clocks used to specify timing constraints. Hence, instead of implementing these structures, symbolic representation of states and on-the-fly model checking are preferred resulting in space and time savings.

On-the-fly model checking means dynamically building the state space during the model checking process, depending on the property to be model checked (Bozga, et al. 1998, Larsen, et al. 1995, Henzinger, et al. 1992a). In these on-the-fly algorithms, some symbolic data structures are used, which make the reachability analysis to be implemented in an efficient way.

This chapter concentrates on the zone and Difference Bounded Matrix (DBM) data structures and the zone-based reachability algorithm that UPPAAL timed automata tool uses.

3.1. Symbolic Data Structures

In the implementation of timed automata, instead of clock regions, zones are used to obtain a finite representation of the infinite state space.

Definition (Zone): Let $\mathbb{T}^X$ be the set of all clock valuations over a finite set of clocks X. A zone is a subset of $\mathbb{T}^X$ defined by a general clock constraint.

A zone, which is simply a disjunction of inequalities between clock variables, is the maximal solution set of clock assignments satisfying some constraint. As an example, Figure 3.1 shows the zone defined by $x_1 > 3 \ \wedge \ x_2 \leq 2 \ \wedge \ x_1 - x_2 < 4$.

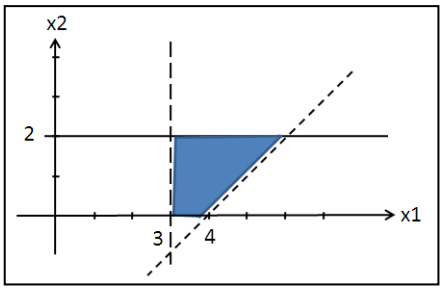

Figure 3.1. An example zone

Given a timed automaton, its zone automaton can be constructed in a similar way as for the region automaton. Figure 3.2 gives the zone automaton for a timed automaton. As it is seen in the figure, unlike the region automaton, the symbolic states in the zone automaton have at most one successor for each event. Although zone automata may be bigger than region automata, in most cases zone automata have less number of states to explore. Because, the number of clock regions on a region automaton depends on the magnitudes of the constants of the clock constraints but the number of zones relatively is not affected by this fact (Alur 1999).

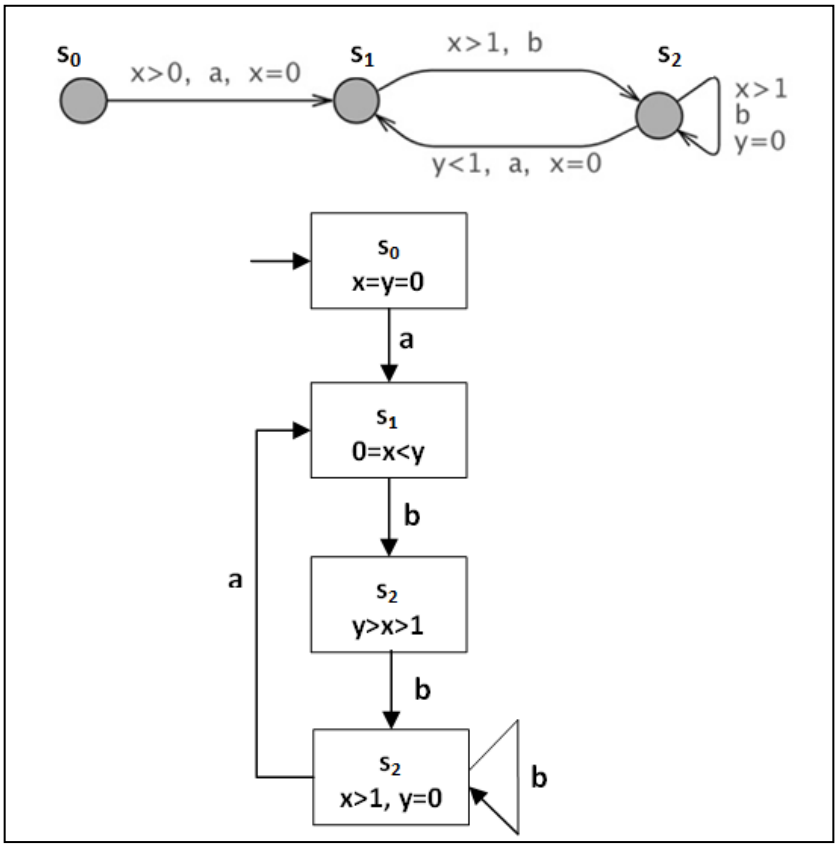

Figure 3.2. Zone Automaton of a Timed Automaton

A symbolic state in a zone automaton can be shown as a pair $\langle s, D\rangle$, where s is a location and D is the maximal set of clock assignments satisfying a clock constraint. Given a symbolic transition relation on the symbolic states; if the initial state $\langle s_0, \{u_0\}\rangle$ may lead to a set of final states according to the symbolic transition relation, all the final states $\langle s_f, D_f\rangle$ should be reachable according to concrete operational semantics (*soundness*). If a state is reachable according to concrete operational semantics, it should be possible to conclude this using the symbolic transition relation (*completeness*) (Bengtsson and Yi 2004).

Zones can be efficiently represented in memory using Difference Bounded Matrices (DBM) that provides an easier implementation of zones. It allows testing for language inclusion of zones, computation of intersection of two zones, future of a zone, image of a zone after reset and the $k-$approximation of a zone (Bengtsson and Yi 2004, Bouyer and Laroussine 2008).

Definition (Difference Bounded Matrix – DBM): A DBM for n clocks is an $(n+1)$-square matrix of pairs $(m; \prec) \in \mathbb{V} = (\mathbb{Z} \times \{<, \leq\}) \cup \{(\infty, <)\}$. It keeps the upper bound for the difference of each pair of clocks where each element of the matrix is defined as:

$$m_{i,j} = \begin{cases} c & if\ (x_i - x_j \leq c)\ is\ a\ constraint \\ +\infty & elsewhere \end{cases}$$

A DBM $M = (m_{i,j}, \prec_{i,j})_{i,j=1\ldots n}$ defines a matrix, with x_0 is always equal to 0 : $v : \{x_1, \ldots\ x_n\} \longrightarrow \mathbb{T} \mid \forall\ 0 \leq i, j \leq n, v(x_i) - v(x_j) \prec_{i,j}\ m_{ij}$ where $\gamma < \infty$ means there is no bound on it. For example, the zone defined by the equations $x_1 > 3\ \wedge\ x_2 \leq 5\ \wedge\ x_1 - x_2 < 4$ can be represented by:

$$\begin{pmatrix} (0;\leq) & (-3;<) & (\infty;<) \\ (\infty;<) & (0;\leq) & (4;<) \\ (5;\leq) & (\infty;<) & (0;\leq) \end{pmatrix}$$

The DBM representation of a zone is not unique. The canonical form of a DBM can be obtained by tightening the clock constraints by using the upper bounds on the clock differences.

In practice, most of the upper bounds are redundant since some of the constraints may be derived from the other ones.

Definition (Minimal Constraint Systems): It is an equivalent reduced system of a constraint system with minimal number of constraints.

For all zones, finding their minimal constraint systems and storing them in memory may reduce the memory consumption. (Behrmann and Bengtsson 2002) gives more information on how to compute the minimal constraint system of a zone.

3.2. Symbolic Reachability Analysis

For the verification of timed automata, a fundamental problem is the emptiness problem that is equivalent to the reachability problem that tests whether a state can be reached in a model. On-the-fly reachability algorithms calculate the states on-the-fly rather than pre-computing. Thus, only the needed part of state space is computed. The use of easy-to-implement structure DBMs and on-the-fly algorithms that glance symbolically on these structures make the timed automata implementable.

The reachability analysis can be performed using forward or backward analyses. The backward analysis starts from final configurations, and computes the predecessors step-by-step iteratively. It checks whether an initial state is eventually computed or not. If such an initial state is computed, than goal location is reachable. Similarly, forward analysis starts from initial configurations and tries to reach some target by computing the successors. If such a final location is computed, it means that the goal location is reachable, and if not computed, it means that the goal location is not reachable. Theoretically, forward analysis algorithm termination is not guaranteed; however, it has the advantage that it is convenient for on-the-fly model checking with useful features like integer variables (Bengtsson and Yi 2004, Bouyer and Laroussine 2008, Mukhopadhyay and Podelski 1999).

UPPAAL tool uses forward reachability analysis. Figure 3.3 gives the zone-based symbolic reachability algorithm for timed automata. This algorithm keeps the $Wait$ list holding the tuple of location and zone $\langle s, D \rangle$ to be visited. Starting from an initial location, we glance the states that are in the $Wait$ list and insert the visited ones to the $Passed$ list. If the zone D is a subset of a zone D' in the $Passed$ list, this will also be added there. Then, the state and zone tuples that are the successors of this state and zone are added to the $Wait$ list since they are reached throughout the glancing of the

states. If the current state that we are examining is a final state, since it is reached, the reachability algorithm returns “yes”, otherwise returns “false”.

The verification algorithm with forward analysis may have a termination problem since the relation $\rightsquigarrow$ is not finite. In order to solve this problem, k −normalization operation that guarantees the algorithm to terminate by limiting the number of computed zones is applied. The symbol $\rightsquigarrow_{k,G}$ in the algorithm is used for the normalization operation.

$Passed = \emptyset, Wait = \{\langle s_0, D_0 \rangle\}$
while $Wait \neq \emptyset$ do
    take $\langle s, D \rangle$ from $Wait$
    if $s = s_f \wedge D \cap \phi_f \neq \emptyset$ then
        return “YES”
    if $D \subseteq D'$ for all $\langle s, D' \rangle \in Passed$ then
        add $\langle s, D \rangle$ to $Passed$
        for all $\langle s', D' \rangle$ such that $\langle s, D \rangle \rightsquigarrow_{k,G} \langle s', D' \rangle$ do
            add $\langle s', D' \rangle$ to $Wait$
        end for
    end if
end while
return “NO”

Figure 3.3. Algorithm for the Symbolic Reachability Analysis

Zone normalization uses the fact that once the clock value is greater than the maximal constant in the clock constraints, it is not important how greater it is. k-normalization operation yields finitely many k-normalized zones defined by a k-bounded constraints having constants between $-k$ and k. Figure 3.4 gives the zone $1 < x < 4 \wedge 2 < y < 4 \wedge x - y < 1$ and its k-approximation for $k = 2$.

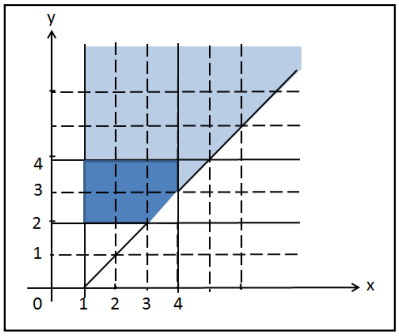

Figure 3.4. A Zone and its k-approximation

More detailed explanation on the zone normalization, possible problems with diagonal constraints and the normalization method to overcome the problem can be found in (Bouyer 2002, 2003, 2004, Bouyer, et al. 2005). Moreover, the implementation details of the normalization method, property checking and transformation operations to implement algorithms on DBMs are given in (Bengtsson and Yi 2004) and used in the verification engine of UPPAAL tool.

CHAPTER 4

MODELING AND VERIFICATION WITH TIMED AUTOMATA

4.1. Formal Verification Methods

Formal verification is the study of proving or disproving that a system meets certain specifications, using formal methods. There are three basic steps for formal verification: **(i)** building a formal model of the system, **(ii)** stating the properties of the system to be verified in a specification language and **(iii)** proving whether the model is correct with respect to the specifications.

Two main approaches for verification of systems are *Theorem Proving* and *Model Checking*. Theorem proving methods prove the general validity of a formula, by using logical inference. Model checking involves building a finite model of a system and checking that the model fulfills the desired property by traversing through all reachable states. Both methods are widely used for verification with some automated tools; however theorem proving methods can be quite troublesome and impractical for complex designs and model checking suffers from state space explosion. This study concentrates on model checking method (Clarke, et al. 1999), that has the advantages of being fully automatic, generating counter example in case of a negative result and not requiring complicated proofs to be written.

Timed automata allows for an efficient model checking method (Bouyer and Laroussine 2008) which can be used to analyze time-sensitive features such as execution times, communication times and response times. Similar to the other model checking methods, it checks a finite model of a system for correctness against some certain requirements. It works by exploring all possible state transitions from the initial state of the system to check whether a specified property is satisfied. It has some drawbacks caused by the generation of huge state spaces as explained in section 6.2.1, which can be understood better after the case study. This section gives brief information about how model checking is performed using timed automata, before moving on the case study on the modeling and verification of a security protocol.

4.2. Modeling with Timed Automata

The first task of model checking is, designing a formal model of the system to be verified. Timed automata formalism, models the system as a network of processes which is composed of several components each having a transition system, namely as a *network of timed automata*.

A network of timed automata, that models a concurrent system, consists of some number of timed automata running in parallel that may communicate and synchronize on some events. It is possible to implement a product timed automata which represents this composite process of the network of timed automata.

Definition (Product or Parallel Composition of Timed Automata): The product of the automata $A_1 = \langle \Sigma, \mathrm{S}_1, \mathrm{S}_{01}, \mathrm{S}_{\mathrm{F}1}, \mathrm{C}_1, \mathrm{E}_1 \rangle$ and $A_2 = \langle \Sigma, \mathrm{S}_2, \mathrm{S}_{02}, \mathrm{S}_{\mathrm{F}2}, \mathrm{C}_2, \mathrm{E}_2 \rangle$ is denoted as $A_1 || A_2$, and defined as $A = \langle \Sigma_1 \cup \Sigma_2, S_1 \times S_2,\ \mathrm{S}_{01} \times \mathrm{S}_{02}, \mathrm{S}_{\mathrm{F}1} \times \mathrm{S}_{\mathrm{F}2},\ \mathrm{C}_1 \cup \mathrm{C}_2, \mathrm{E} \rangle$, where E is defined by:

- ✓ For $a \in \Sigma_1 \cap \Sigma_2$, for every $(s_1, s'_1, a, R_1, \varphi_1) \in E_1$ and $(s_2, s'_2, a, R_2, \varphi_2) \in E_2$, E has $((s_1, s_2), (s'_1, s'_2), a, R_1 \cup R_2, \varphi_1 \cap \varphi_2)$
- ✓ For $a \in \Sigma_1 \backslash \Sigma_2$, for every $(s_1, s'_1, a, R, \varphi) \in E_1$ and every s in S_2, E has $\big((s_1, s), (s'_1, s), a, R, \varphi\big)$
- ✓ For $a \in \Sigma_2 \backslash \Sigma_1$, for every $(s_2, s'_2, a, R, \varphi) \in E_2$ and every s in S_1, E has $((s, s_2), (s, s'_2), a, R, \varphi)$

Simply, in the product automata, the transitions of the timed automata that does not correspond a shared action are interleaved, and the transitions with a shared action are synchronized. Figure 4.1 illustrates a simple example of a product automaton. In this network of timed automata, there are two timed automata representing $user$ and $door$ processes, which synchronize on $press$ and $open$ events. For instance, when the user presses the button, the door goes into the opening state and the user waits for the door to open. The product automaton is given below, which is the composite process of these two systems. The clock resets, guards and the invariants take place in the composite automata as given in the definition.

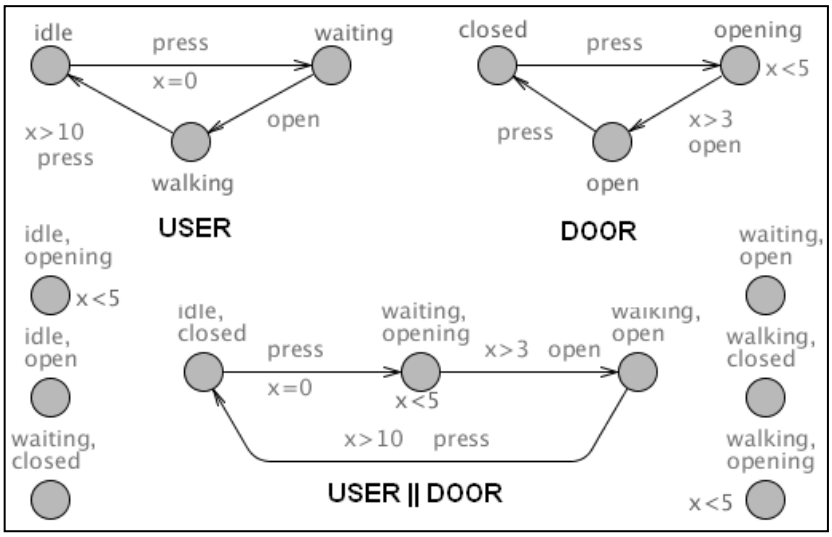

Figure 4.1. An Example Network of Timed Automata

4.3. Specification and Verification with Timed Automata

To perform verification, the specifications that must be satisfied by the system should be stated in a formal specification language. Then, it is possible to use a formal verification method to check whether the model satisfies the requirements. Verification of an automata model can be performed using a homogenous or heterogeneous approach.

- In *homogenous approach*, the requirements of a system are specified as an automaton, similar to the system itself. Then, the behavior of the system model and the specification automata are compared. Let us say that the specification is given as the automaton A_S and the implementation as A_I. Then, the implementation meets its specification iff $L(A_I) \subseteq L(A_S)$. In this case, verification problem reduces to the emptiness check of $L(A_I) \cap L(\overline{A_S})$. For untimed automata, the complementation is decidable and the method is straightforward. In case of timed automata, this problem can be solved algorithmically when A_S is given as a deterministic timed automaton.
- The *heterogeneous (or dual language) approach*, combines the automata formalism with a descriptive formalism suitable for specifying its properties. For example, the timed automata tool KRONOS uses TCTL (Timed Computation Tree Logic) and UPPAAL uses CTL (Computation Tree Logic) to specify the

requirements. Then, a reachability check is performed on the model to test whether the automata model satisfies the requirements (Furia, et al. 2010).

Note that in both cases, on-the-fly reachability algorithms are used for emptiness and reachability checking since the problem is PSPACE-complete. Instead of pre-computing the product automaton, it is computed and traversed on-the-fly, as explained in the previous chapter.

4.4. A Timed Automata Tool: UPPAAL

UPPAAL (Bengtsson and Larsen 1996) is a timed automata tool developed by Uppsala and Aalborg Universities. It extends timed automata with C-like data types such as integers and arrays and allows using urgent and committed locations that ease the modeling of a system. It has rich documentation related to both its implementation details and usage which can be found in (Larsen and Pettersson 1997, Amnell and Behrmann 2001, Behrmann and Bengtsson 2002, Behrmann and David 2004, Behrmann et al. 2006). The tool is in continuous development since its first official beta version in 1999. In this study, the latest version UPPAAL 4.0.11, released in February 2010 is used.

UPPAAL is freely available and it provides an integrated environment for modeling, validation and verification of real-time systems with a Java interface and C++ verification engine. It is an efficient and mature tool that has been used in many academic and industrial case studies including the modeling and verification of bounded retransmission protocol (D'Argenio, et al. 1996, 1997), a collision avoidance protocol (Jensen, et al. 1996), TDMA protocol startup mechanism (Lönn and Pettersson 1997), audio-video protocols (Bengtsson, et al. 1996, Havelund, et al. 1997, Bengtsson, et al. 2002), a gear controller (Lindahl, et al. 1998), lip synchronization algorithm (Bowman, et al. 1998), a power controller (Havelund, et al. 1999), commercial field bus protocol (Wang and Yi 2000), QoS properties in multimedia streams (Bordbar and Okano 2003) and WAP gateway (Hessel and Pettersson 2006). Some recent studies (Corin, et al. 2004, 2007) use UPPAAL for verification of security protocols such as Needham-Shroeder and Yahalom authentication protocols.

4.4.1. Modeling with UPPAAL

A system can easily be modeled as a network of timed automata, using the graphical user interface and writing some simple C-like coding for system definitions. Then, the system execution can be simulated by visualizing the possible dynamic behaviors of the system.

In the UPPAAL modeling language, the actions in a network of timed automata are partitioned into a set of input and output actions, which provide the communication between the automata. The output (or send) statement over channel a is denoted as $a!$ (emission) and an input (or receive) statement over channel a is denoted as $a?$ (reception). Two edges in different processes can synchronize if one is emitting and the other is receiving on the same channel.

UPPAAL provides easy modeling of a system using C-like data structures, committed and urgent states and synchronization channels. It also allows transitions to have *guards*, *synchronizations* or *updates* and allows states to have *location invariants*. The syntax of these expressions and the restrictions on them are explained in detail in UPPAAL documentation (Behrmann and David 2004).

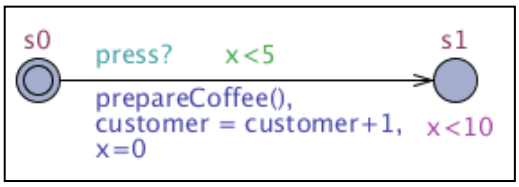

Figure 4.2. An Example Timed Automaton Designed using UPPAAL

Figure 4.2 gives a simple example of timed automata designed using UPPAAL. In the figure, the state with double border lines demonstrates that it is the initial state. When the automaton takes the input over the $press$ channel, it goes into state $s1$ if the guard $x < 5$ is satisfied. When the transition is taken, it performs the specified updates. As it is seen, besides resetting clocks, UPPAAL has the advantage of calling functions and using assignment expressions as an update which allows for the easy modeling of a system. The state $s1$ has an invariant demonstrating that the automaton can stay in this state as long as $x < 10$.

4.4.2. Specification and Verification with UPPAAL

UPPAAL automatically performs verification of timed automata models based on constraint solving and on-the-fly techniques. In case of a negative result, it generates a diagnostic trace that can be loaded to the simulator and examined how the property is violated.

The properties to be checked with UPPAAL verification engine are specified using a subset of *Computation Tree Logic* (CTL). This query language consists of *path* and *state formulas*.

State formulas describe side-effect free expressions that can be evaluated pertaining to a state (e.g. $i == 3$, is true in a state whenever i equals 3). They can be used to test whether a process is in a particular state using expressions on the form $P.s$, where P is a process and s is a state.

Deadlock-freedom is a special property for systems which are supposed to operate indefinitely. Absence of deadlock can be checked using a special state formula, $A[\,]\ not\ deadlock$. In general, deadlocks are states where the system is unable to progress further. The study in (Bowman and Gomez 2006) classifies the deadlocks in timed automata, as **(i)** *pure-actionlocks*, analogue to the deadlock in untimed specifications where the system cannot perform any action transitions but time can progress, **(ii)** *time-actionlocks* in which neither action nor time transitions can be performed and **(iii)** *zeno-timelocks* (also called *pure timelocks*) where a system can still perform transitions that may be action or time transitions but time cannot pass beyond a certain point. The state formula $A[\,]\ not\ deadlock$ guarantees the absence of actionlocks. If this expression is not verified, the system may have a pure-actionlock or a time-actionlock.

Path formulas quantify over paths of the model. They can be classified into reachability, safety, liveness and bounded liveness properties (see Figure 4.3).

- **Reachability properties *(something will possibly happen)***: Given a formula, they check, whether it can possibly be satisfied by any reachable state. For example for a security protocol, a reachability property may ask whether a process can enter the critical section. The path formula $E\langle\rangle\ \varphi$ is used to express there is a path that, starting from initial state, reaches a state where φ is eventually satisfied.

- **Safety properties *(something bad will never happen)*:** They are the properties required to always hold. For example, "it cannot happen that both processes are in their critical sections simultaneously" defines a safety property. The path formulae A [ ]φ expresses that for all paths, φ will always hold (something good is invariantly true), and the formulae E [ ]φ expresses that for some paths, φ will eventually hold (there should exist a minimal path such that φ is always true).
- **Liveness properties *(something will eventually happen)*:** These properties are characterized by the fact that no event can really violate them. The path formula A⟨⟩ φ states that φ is eventually satisfied. Besides this *eventually* liveness properties, a more useful form is the *leadsto* or *response* property. It is written as $\varphi \dashrightarrow \psi$ that means whenever φ is satisfied, then eventually ψ is satisfied. For example for a communication protocol, "whenever a message has been sent then eventually it will be received" forms a liveness property. Bounded form of the liveness properties, $leadsto\ within$ states that whenever φ holds, then for all paths thereafter ψ must also hold within some time. Such a property can be: "whenever a message has been sent, then eventually it will be received within two time units".

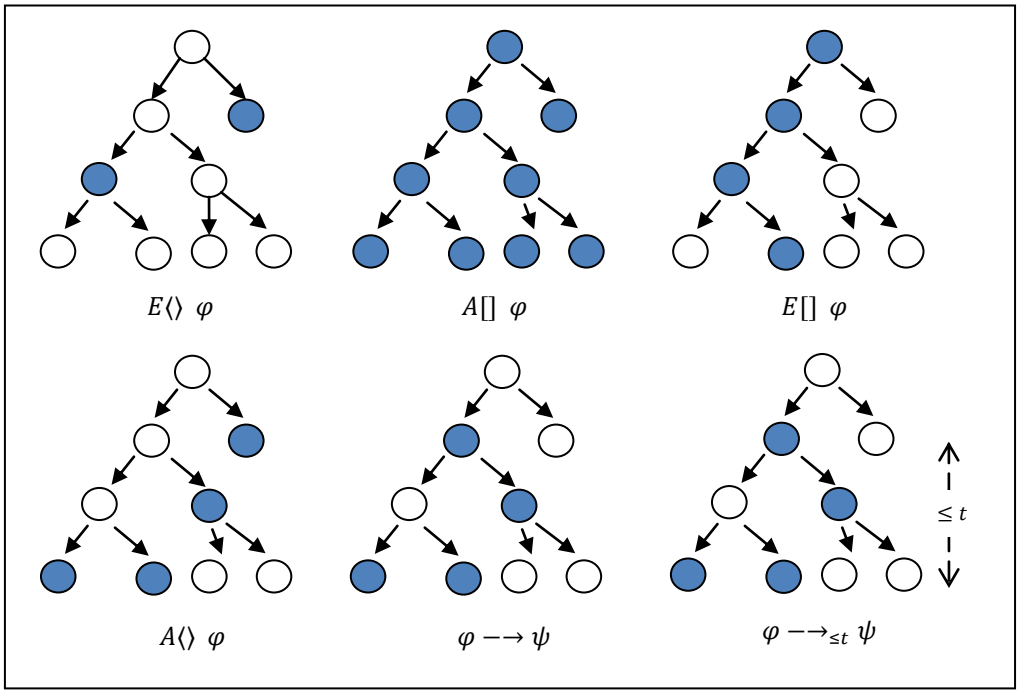

Figure 4.3. Path formula for Reachability, Safety and Liveness Properties

CHAPTER 5

A CASE STUDY: USING TIMED AUTOMATA FOR MODELING AND VERIFICATION OF NEUMAN-STUBBLEBINE REPEATED AUTHENTICATION PROTOCOL

5.1. Related Work

Timed automata formalism is a widely used model for the verification of real time systems with many case studies including audio-video protocols, gear controller, lip synchronization algorithm and power controllers. This study focuses on the modeling and verification of security protocols using timed automata.

Some recent studies analyze security protocols with quantitative timing properties involving the use of timed automata. The studies in (Jakubowska, et al. 2005, 2008) examine Kerberos, TMN, Neumann-Stubblebine, Andrew Secure and Wide Mouthed Frog protocols by not modeling them directly as timed automata, but translating a language specification of a security protocol automatically to timed automata without integer variables. Then, the translated timed automata model is used as input for the model checker KRONOS (Yovine 1997) and VerICS (Dembinski, et al. 2003). Similarly in (Benerecetti and Cuomo 2009), a model checking tool is presented which translates a security specification language into timed automata and uses the UPPAAL tool as the verification engine. Additionally, a case study on Wide Mouthed Frog protocol is provided.

Similar to our case study, these studies perform verification using timed automata tools. However, our approach is closer to the studies in (Corin, et al. 2004, 2007) which model Needham-Schroeder and Yahalom protocols directly with timed automata and use UPPAAL tool for verification. Our case study models Neuman-Stubblebine repeated authentication protocol which allows for employing the key expiration time in the model. Since it is a large protocol involving two parts, the model tends to grow enormously. Directly modeling provides us full control over the timed

automata model and also enables us to make use of the full expressiveness and data structures of UPPAAL. Moreover, we are not required to have an expertise on a specification language to model a protocol.

5.2. Modeling Security Protocols using Timed Automata

Timed automata provide a model close to the real system. A timed automata model for a protocol can be generated by building a finite state machine whose states and transitions simulate the behavior of the protocol principals. Then, all possible execution traces are explored to analyze whether the protocol has some security flaws.

Model checking a protocol involves an analysis based on the protocol specification, independent of the cryptographic operations. Hence, the cryptosystem is assumed to be perfect and the mathematical details of the cryptology are abstracted away. It is focused on the protocol steps and the protocol messages sent by the principals. Since we are related to the sequence of steps in the protocol specification, the flaws related to the different combinations of the messages are examined by modeling an intruder which aims to attack the protocol.

Modeling cryptology may require some coding such as assigning and testing some local/global variables and performing some operations. UPPAAL makes it easier to model a protocol directly with timed automata since it allows such simple coding. Performing most of the operations as the updates of the transitions and checking the values of the variables in the guards to decide whether they comply with the protocol specification are very helpful to model a protocol.

To create a model of the protocol, it is crucial to examine the description of the protocol and the behavior of the principals. In a server involved authentication protocol such as the Neuman-Stubblebine authentication protocol used in our case study, the initiator A, authenticates itself to the responder B, using the trusted authentication server S; and obtains a session key to communicate with B. By modeling each principal as an automaton, a network of timed automata is obtained which models the protocol execution. Note that a network of timed automata deadlocks if no more transitions can be involved in a run. Hence, a behaviour that is not specified in the protocol, causes a deadlock and does not generate the complete the protocol execution trace.

The principals A, B and S communicate with each other by sending or receiving some messages over the network. The messaging can be modeled by using binary synchronization channels. The sender automaton emits an output signal that will be captured by another automaton receiving the signal. The message sent/received can be kept in a global variable so that each principal automaton can access the message variable. Figure 5.1 demonstrates a general view for the network of timed automata. For example, when the initiator emits the $init\_msg!$ signal, this means that it has created and sent the message (it also assigns the message to the global message variable). The network takes this message over $init\_msg?$ and emits $resp\_msg!$ signal, which is captured by the responder over $resp\_msg?$. Then, the responder accesses the global message variable and reads it.

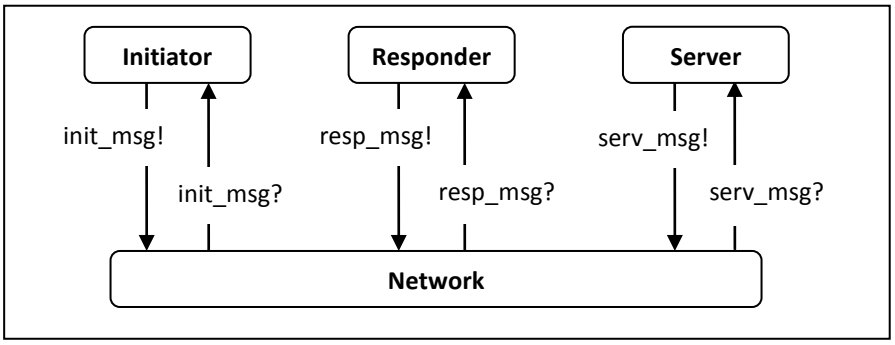

Figure 5.1. General View of Automata for Principals and the Network

5.3. Neuman-Stubblebine Repeated Authentication Protocol

Neuman-Stubblebine protocol (Neuman and Stubblebine 1993) is an authentication protocol that involves a session key exchange and mutual authentication between two principals. Our study performs a case study on this protocol since it is a repeated protocol that involves an expiration time that can be studied using timed automata and it requires less number of message transfers then the similar nonce-based Kehne-Langendorfer-Schoenwalder repeated authentication protocol (Schonwalder, et al. 1992).

The protocol consists of two parts. First, the initial authentication part is executed which provides mutual authentication. In this part, the initiator acquires a ticket to be used in the subsequent part of the protocol. The subsequent part is used to

re-authenticate the principal identities without using the server. This part can be repeated several times until the ticket expires.

The following symbols are used in the protocol specification given for the initial and subsequent authentication parts:

- A, B and S are the principals where S is the key distribution server.
- N_a, N_b, N'_a, and N'_b are the nonces.
- K_{as}, K_{bs}, and K_{ab} are the keys where the subscript letters denote the principals whom the key is for (For example K_{as} is shared between A and the server S).
- t_b is the expiration time for the session key.
- $\{X, Y\}_k$ means, X concatenated with Y, encrypted with the key k.

1. Initial Authentication Part:

The initial part requires the exchange of four protocol messages. A initiates the protocol by sending its identity A and a nonce $\mathrm{N_a}$. After B receives this message, it sends its identity and a nonce created by B as clear text and A's name, nonce and a suggested expiration time for the credentials as a block encrypted with the key $\mathrm{K_{bs}}$. The server can decrypt this message since it knows $\mathrm{K_{bs}}$, and assures that they are created by B. Then, the server sends A a ticket, and B's nonce. It also sends the identity of B, A's nonce, a session key $\mathrm{K_{ab}}$, the expiration time $\mathrm{t_b}$ encrypted with $\mathrm{K_{as}}$. A decrypts the block encrypted with $\mathrm{K_{as}}$and verifies the $\mathrm{N_a}$ is same with the $\mathrm{N_a}$ in message 1. In the last message, it sends the ticket and $\mathrm{N_b}$ to B, proving its identity.

1. $\mathrm{A} \rightarrow \mathrm{B}: \ \mathrm{A}, \mathrm{N_a}$
2. $\mathrm{B} \rightarrow \mathrm{S}: \ \mathrm{B}, \{\mathrm{A}, \mathrm{N_a}, \mathrm{t_b}\}_{\mathrm{K_{bs}}}, \mathrm{N_b}$
3. $\mathrm{S} \rightarrow \mathrm{A}: \ \{\mathrm{B}, \mathrm{N_a}, \mathrm{K_{ab}}, \mathrm{t_b}\}_{\mathrm{K_{as}}}, \{\mathrm{A}, \mathrm{K_{ab}}, \mathrm{t_b}\}_{\mathrm{K_{bs}}}, \mathrm{N_b}$
4. $\mathrm{A} \rightarrow \mathrm{B}: \ \{\mathrm{A}, \mathrm{K_{ab}}, \mathrm{t_b}\}_{\mathrm{K_{bs}}}, \{\mathrm{N_b}\}_{\mathrm{K_{ab}}}$

This initial authentication provides mutual authentication between the principals. After this initial part, the initiator A possesses the ticket $\{\mathrm{A}, \mathrm{K_{ab}}, \mathrm{t_b}\}_{\mathrm{K_{bs}}}$and the session key $\mathrm{K_{ab}}$ that can be used for subsequent authentications.

2. **Subsequent Authentication Part:**

In this second part, A uses the ticket to authenticate itself to the responder. B checks the sender's identity, shared key and the expiration time of the ticket. If it is valid, the authentication is provided between the principals.

1. $\mathrm{A} \rightarrow \mathrm{B}: \ \mathrm{N'_a}, \{\mathrm{A}, \mathrm{K_{ab}}, \mathrm{t_b}\}_{\mathrm{K_{bs}}}$
2. $\mathrm{B} \rightarrow \mathrm{A}: \ \mathrm{N'_b}, \{\mathrm{N'_a}\}_{\mathrm{K_{ab}}}$
3. $\mathrm{A} \rightarrow \mathrm{B}: \ \{\mathrm{N'_b}\}_{\mathrm{K_{ab}}}$

The Neuman-Stubblebine protocol is exposed to some security flaws (Clark and Jacob 1997, Hwang, et al. 1995).

Attack 1: The initial part of the protocol is exposed to a *type flaw* attack, where the responder B accepts the nonce N_a as the key K_{ab}.

1. $I(A) \rightarrow B: \ A, N_a$
2. $B \rightarrow I(S): \ B, \{A, N_a, t_b\}_{K_{bs}}, N_b$
3. $omitted$
4. $I(A) \rightarrow B: \ \{A, N_a, t_b\}_{K_{bs}}, \{N_b\}_{N_a}$

Attack 2: The subsequent part of the protocol is subject to a *parallel session attack*. Here, the initial ticket is recorded from a legitimate run of the protocol.

1. $I(A) \rightarrow B: \ N'_a, \{A, K_{ab}, t_b\}_{K_{bs}}$
2. $B \rightarrow I(A): \ N'_b, \{N'_a\}_{K_{ab}}$

1'. $I(A) \rightarrow B: \ N'_b, \{A, K_{ab}, t_b\}_{K_{bs}}$

2'. $B \rightarrow I(A): \ N''_b, \{N'_b\}_{K_{ab}}$

3. $I(A) \rightarrow B: \ \{N'_b\}_{K_{ab}}$

Our case study focuses on the modeling the protocol as a network of timed automata composed of the protocol principals and an intruder, then examines whether our timed automata model is able to detect the flaws on the protocol.

5.4. Modeling the Neuman-Stubblebine Initial Part

In the first step of the protocol modeling, the initial authentication part, which requires the transmission of four messages, is modeled. Then, the model for the

subsequent part that requires three messages for re-authentication is generated. Finally, these two parts are combined and a complete model for the protocol is obtained.

5.4.1. Modeling Assumptions

Our timed automata model has the following assumptions:

- The principals show no behavior other than the behavior described in the protocol specification
- Principals know their secret keys they use with server
- A kind of black box security protocol analysis approach (Cremers 2006) is used in the protocol modeling. Cryptographic functions are considered as abstract black boxes, immune to cryptanalysis. Perfect cryptology is assumed such that:
    - Nobody can decrypt the messages unless they know the secret keys.
    - A ciphertext $\{m\}_K$ can be generated by principal possessing m and K.
    - Nobody can guess the secret keys or newly generated nonces.
- The medium does not introduce errors, message modification can only occur in the existence of an intruder.
- The intruder has the capabilities of the powerful Dolev - Yao intruder (Dolev and Yao 1981) which has the ability to eavesdrop, replay, modify or inject messages. All communication is assumed to occur over an insecure network.
- Network delays are not taken into consideration since they are negligible with respect to the time consuming operations such as encryption or decryption.

5.4.2. Modeling Cryptology

Since the cryptosystem is assumed to be perfect, we used an abstraction for the cryptographic operations. These cryptographic abstractions are held in the local functions of each principal, $gen\_nonce()$, $encrypt(int\ plaintext, int\ key)$ and $decrypt(int\ ciphertext, int\ key)$, described in local declarations of each automata.

Each principal can call these functions when needed in the protocol execution and the result of the operation is assigned in their local variable $result$. During the execution of these operations, time elapses, so when one of these functions is called, the

principal goes in a new state and gets the result in the specified amount of time. Here, we make use of the advantage of the timed automata that can model the delay and deadline requirements. A principal's automaton will have the states and transitions as given in Figure 5.2, to generate a nonce, perform encryption or decryption. The states marked with "C" are the committed states where the transition is taken as soon as we enter the state and time does not pass. The states in the middle are not committed since they allow the passage of time during the cryptographic operations.

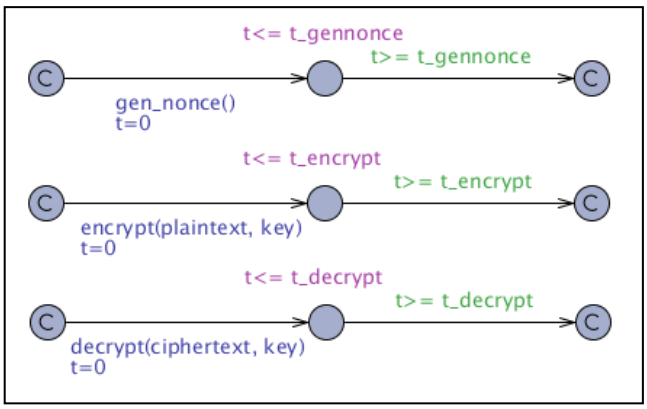

Figure 5.2. Timed Automata for Cryptographic Operations

A message sent/received by a principal is represented as an integer, which contains the information described in the protocol specification. As it is seen in the protocol specification, a protocol message consists of a combination of the agent ids, nonces or encrypted blocks. The creation of the messages and the encryption/decryption model used in the case study are similar to the model used in (Corin, et al. 2004, 2007).

Nonce Generation:

In the modeling of the initial authentication, the initiator and the responder each generates one nonce value by calling their $gen\_nonce()$ function. This function returns a result by incrementing the global $nonce$ variable.

Encryption/Decryption:

Two arrays $plain$ and key are used for encryption and decryption, where the first one holds the plaintexts and the latter holds the keys. When a block is encrypted, the plaintext is placed in the $plain$ array and the key is placed in the key array. The corresponding index is returned as the ciphertext, which is the result of the encryption.

Figure 5.3 shows an example scheme. Here, N_b which has the value 15 is encrypted by using the key $K_{ab} = 13$. Let us say that the current index value is 7. Then, when the function $encrypt(N_b, K_{ab})$ is called, $plain[7]$ will contain the value 15 and $key[7]$ will contain 13. Decryption is performed again by using these arrays. Let us say that $decrypt(block, K_{ab})$ is called with $block = 7$ and $K_{ab} = 13$. Then, $key[7]$ will be compared to the value of K_{ab}; in other words it is checked whether the specified key is correct. If this key is not same with the corresponding key of the ciphertext, then the block cannot be decrypted. If it is same with the value in $key[7]$, then the block can be decrypted, in other words $plain[7]$ is returned as the plaintext which has the value 15.

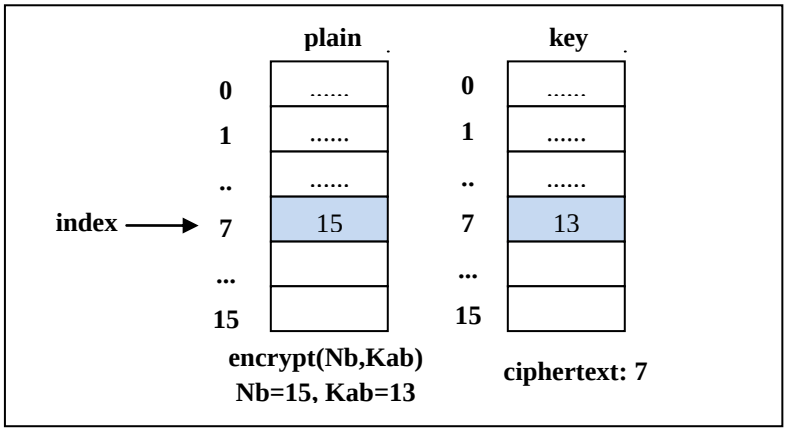

Figure 5.3. Modeling Encryption and Decryption

Representing Protocol Variables:

As mentioned above, a protocol message is represented as an integer in the model. UPPAAL uses 16 bit integers where the leftmost bit is the sign bit. In order to contain the whole message in an integer and have simplicity in the model, it is necessary to limit the number of bits to represent the blocks contained in a message. To figure out the number of bits to use for each variable, the largest message and the largest block to be encrypted/decrypted (since the plaintexts will be contained in an integer array, they also must not exceed the size of the integer) is taken into the consideration. The largest message of the protocol is in the third step of the protocol, server S sends A the message: $S \rightarrow A: \{B, N_a, K_{ab}, t_b\}_{K_{as}}, \{A, K_{ab}, t_b\}_{K_{bs}}, N_b$. and the largest block in the protocol specification is $\{B, N_a, K_{ab}, t_b\}_{K_{as}}$. It is seen that, the message can at most contain two encrypted blocks and a nonce; and the largest block contains an identity, a nonce, a key and an expiration time value.

In order not to exceed the number of bits of an integer, the nonces, keys and the indices (in other words, ciphertexts or encrypted blocks) are represented by using four bits. The agent ids and the t_b are represented by two bits to be able to fit the message content in an integer. Representing the ids with two bits does not create a problem since the initiator, the responder and the intruder has the agent ids $A = 1$, $B = 2$ and $I = 3$ respectively. However, the time duration that can be represented by only two bits is not enough to complete the protocol execution; because the protocol needs at least eight time units to finish the execution even if each cryptographic operation is assumed to take one time unit. Hence, the expiration time is used as $tb \times 10$ in the calculations.

The keys shared between the principals K_{as} and K_{bs} are defined as $Key\_Base + id$, where $Key\_Base = 10$. In the implementation, the possible values for the nonces, id's, indexes and keys are restricted (see Table 5.1) since there may be some problems in the model when some of these values coincide. In section 6.2.2, it is explained why some predefined values are used and what would be the problems when they coincide.

Table 5.1. Variables and Their Values Used in Cryptology Modeling

Identities (A, B, Intruder)	1, 2, 3
Secret Keys (Kas, Kbs, Kab)	11,12,13
Nonce Values*	10, 14,15
Index Values	[4, 9]
0 means a value is not set or wrong	
* 1 and 2 are also used as nonces for the subsequent part of the protocol	

The lengths of these variables are $Nonce\_Length = Key\_Length = Index\_Length = 4$ and $Agent\_Length = Time\_Length = 2$. To reduce the state space of the intruder (see section 6.2.1) and have simplicity in the model, instead of these six variables, only two global length variables $Length1 = 4$ and $Length2 = 2$ are used in our model. These values do not change during the execution and they are defined as constant integers which makes us save from space.

Creating and Reading Messages:

Up to now, it is decided on the number of bits to represent each part of a message and how to encrypt/decrypt blocks with a secret key. Now we can go on with how to create and read messages.

A message is created and read by using shift and or operations (Corin, et al. 2004, 2007). Parts of the protocol message are appended to the message by shifting the message to the left as the length of the part that will be appended. Then, the new part is appended using the or operation. Hence a message is created and ready to be sent. To extract information from a received message, shift and and operations are used. This time the message is shifted to the right and the and operation with the mask is applied. The mask variables are also defined as global constant integers similar to the length variables used to obtain the specified number of rightmost bits: $Mask1 = 15$ (0000 0000 0000 1111) and $Mask2 = 3$ (0000 0000 0000 0011) For example, when A wants to create the message A, N_a, it shifts N_a left for $Length1$ times, and or s the result with N_a. Figure 5.4 demonstrates how to create this message and Figure 5.5 shows how the server reads the message 2 of the protocol.

A creates the message $A \rightarrow B: A, N_a$

$the\ msg\ is\ initially\ 0$	0000 0000 0000 0000
$msg = A\ (where\ A = 1)$	0000 0000 0000 0001
$msg = msg << Length1$	0000 0000 0001 0000
$msg = msg \mid N_a\ (where\ N_a = 14)$	0000 0000 0001 1110

$A\ sends\ the\ msg = 30\ to\ B$

Figure 5.4. Creating a Message

S reads the message $B \rightarrow S: B, \{A, N_a, t_b\}_{K_{bs}}, N_b$

0000 0010 0100 1111	$the\ msg\ received\ is\ 591$
0000 0000 0000 1111	$Nb = msg\ \&\ Mask1$
0000 0000 0000 0100	$block\_Kbs = (msg >> Length1)\ \&\ Mask1$
0000 0000 0000 0010	$B = (msg >> (2 * Length1)\ \&\ Mask2$

now, S knows N_b and B. Then, S decrypts $\{A, N_a, t_b\}_{K_{bs}}$ by calling $decrypt(block\_Kbs, Kbs)$ where $block\_Kbs = 4,\ K_{bs} = 12$ and extracts A, N_a and t_b from the plaintext in a similar way.

Figure 5.5. Reading a Message

5.4.3. Initiator, Responder and Server Automata

In the Neuman-Stubblebine protocol, the messages are transferred between the initiator, responder and the trusted server. The steps of the protocol are implemented by mainly these three automata, that model the creation of messages, sending and receiving messages, extracting information and checking them using its knowledge.

The knowledge bases of the principals are modeled using the local variables of each principal. For example, the principal A has K_{as} as the initial knowledge. Then, it generates N_a, owns a ticket, learns K_{ab}, and t_b and these values are added to this principal's knowledge base. The principal B, has K_{bs} as the initial knowledge and gets a claimed id, N_a, generates N_b, t_b and learns K_{ab}. It should keep this knowledge to be used in the later steps of the protocol (e.g. while checking the values received in the fourth step of the message). Hence, it is necessary to keep each principal's knowledge in their local variables.

In order to be able to analyze the timing properties, each principal automaton has its local clock variable to keep the time elapsed for the cryptographic operations and check timeouts, in addition to a global clock representing the total time passed.

The Initiator:

The initiator A, involves in the following three steps of the protocol:

1. $A \rightarrow B: \ A, N_a$
3. $S \rightarrow A: \ \{B, N_a, K_{ab}, t_b\}_{K_{as}}, \{A, K_{ab}, t_b\}_{K_{bs}}, N_b$
4. $A \rightarrow B: \ \{A, K_{ab}, t_b\}_{K_{bs}}, \{N_b\}_{K_{ab}}$

The Initiator automaton is given in Figure 5.6, which is activated by the $Init$ automaton that emits the $start!$ signal. The Initiator, first generates a nonce which is performed as given in Figure 5.2. It creates the message A, N_a by assigning the message to the global variable msg and signals $init\_msg!$ to indicate that it has sent the message. This signal will be captured by the network and will be transmitted to the responder. After sending the message, in state $A5$, initiator waits for the protocol message 3 from the server. When the message $\{B, N_a, K_{ab}, t_b\}_{K_{as}}, \{A, K_{ab}, t_b\}_{K_{bs}}, N_b$ is sent by the server, the network emits $init\_msg!$ which will be captured by the $init\_msg?$ of the initiator that brings it to state $A6$. In this transition, initiator gets the block encrypted with K_{as}, the ticket which is $\{A, K_{ab}, t_b\}_{K_{bs}}$, and N_b. It decrypts the

block with the key K_{as} which is shared by the initiator and the server. Decryption of a block is performed by its function similar to the generation of a nonce. Since it is a time consuming operation, time elapses during decryption. So, we use a location invariant in the state that waits for decryption. The transition from $A6$ to $A7$, has a guard. In order to take the transition, besides the time constraint, B's identity sent by the server must be same as the identity that A wants to communicate with, and the nonce value must be same as the one generated and sent by itself. If this guard is satisfied, the initiator gets the K_{ab} and t_b values and encrypts N_b with K_{ab}. Then, it creates the message $\{A, K_{ab}, t_b\}_{K_{bs}}, \{N_b\}_{K_{ab}}$ by concatenating the ticket and this encrypted block, emits the $init\_msg!$ signal indicating that it has sent the message. After this step, the initial authentication execution finishes for the initiator and it sets its local variable $finish1$ to $true$, which will be used for verification step.

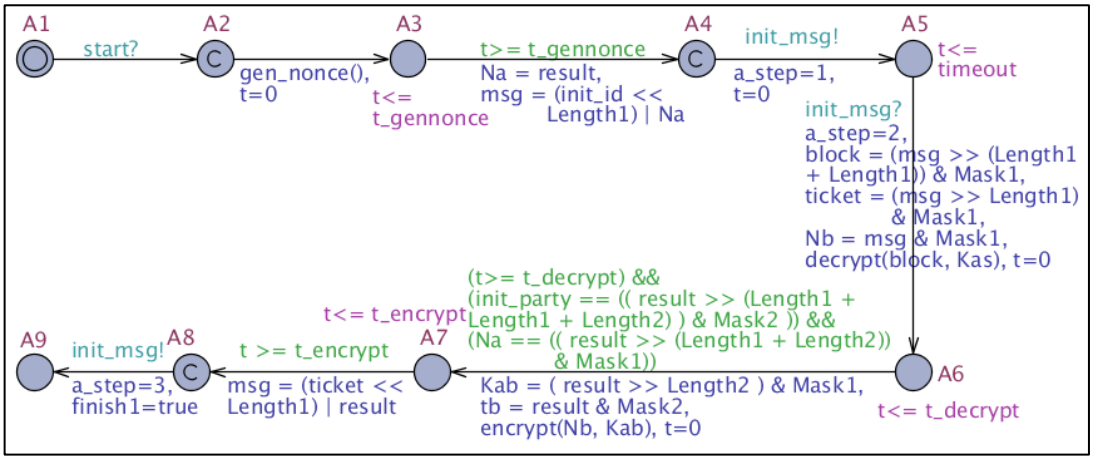

Figure 5.6. The Initiator Automaton for the Initial Authentication Part

Let us assume that the initiator has received a wrong message from the server in the third step. Then, since the N_a will be different from the one the initiator itself generated, the guard will not be satisfied and the transition from state $A6$ to $A7$ will not be taken. Hence, the automata will deadlock and the $finish1$ value will be 0 which means that there is something wrong with the execution of the protocol.

In the automaton, some states are defined as committed since it allows for accurate modeling of atomic behaviors and avoids unnecessary interleavings. $A2$ and $A4$ are such committed states. On the other hand, there are some states such as $A3$ and $A5$ where time should elapse in some states such as or interleavings should be allowed.

From the point of analysis of timeouts, some modifications on state $A5$ can be proposed where the initiator waits for the protocol message 3. Total time waited for this message can be measured in order to be used for the analysis of possible attacks. Figure 5.7 shows the changes in states $A5$, $A6$ and in their transitions. The local variable $t\_waited1$, keeps the time waited for the message which is incremented when the local clock $t2$ becomes 1. This time recording is performed until the timeout is reached since the protocol run does not continue when the message does not arrive in timeout interval. The guard $t2 == 0$ in the transition outgoing from A5 ensures that $t\_waited1$ is incremented each time $t2$ gets 1 by preventing the automaton to take this transition before incrementing t_waited for the last time unit.

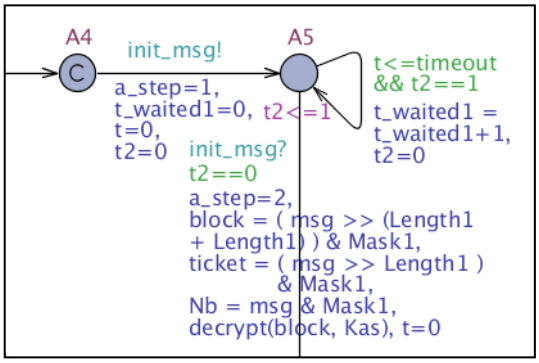

Figure 5.7. Measuring the Time Waited for a Message

Note that, in the initial authentication part of our case study, only the waiting time for the Responder automaton is included in the model since it is used to detect the attack of the protocol and we have state space explosion problem when we included these transitions in both principals.

The Responder:

The responder B, involves in the following three steps of the protocol:

1. $A \rightarrow B : A, N_a$
2. $B \rightarrow S : B, \{A, N_a, t_b\}_{K_{bs}}, N_b$
4. $A \rightarrow B : \{A, K_{ab}, t_b\}_{K_{bs}}, \{N_b\}_{K_{ab}}$

The automaton for responder B is given in Figure 5.8. It starts the protocol execution after it gets the first protocol message over $resp\_msg?$ signal. The creation of the automaton is performed in a similar way as for the initiator.

As for the timeout interval, in state $B7$, B waits for the protocol message 4, $\{A, K_{ab}, t_b\}_{K_{bs}}, \{N_b\}_{K_{ab}}$ from A. The variable $t\_waited1$ keeps the total time waited for the message which is incremented when the local clock $t2$ becomes 1. This part is similar to the piece of automata given in Figure 5.7 and used for protocol verification using timing information.

The responder B finishes its execution by setting its local variable $finish1$ to 1, which means that the claimed identity is authenticated, only if the last protocol message is correct. This check is done using a guard and the transition to state $B9$. If the values are correct, it gets the key K_{ab}, decrypts the block $\{N_b\}_{K_{ab}}$ to check whether it is same with the nonce generated by itself.

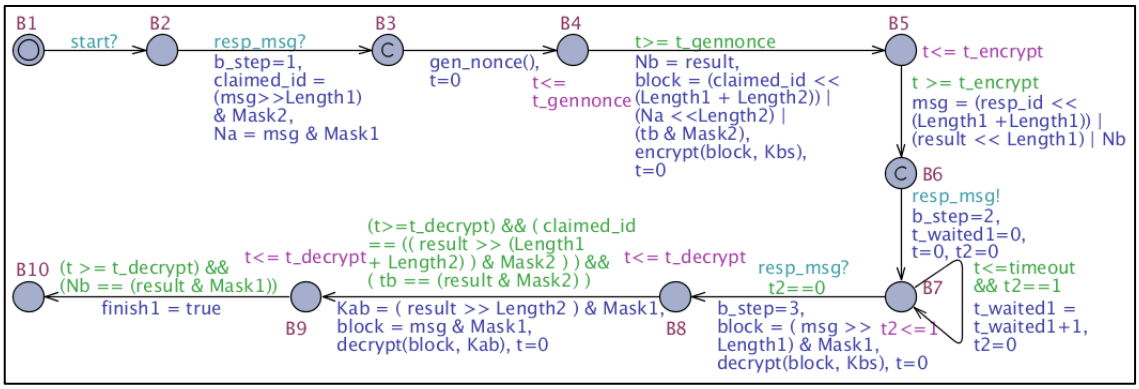

Figure 5.8. The Responder Automaton for the Initial Authentication Part

The Server:

The server S, involves in the following two steps of the protocol:

2. $B \rightarrow S : B, \{A, N_a, t_b\}_{K_{bs}}, N_b$
3. $S \rightarrow A : \{B, N_a, K_{ab}, t_b\}_{K_{as}}, \{A, K_{ab}, t_b\}_{K_{bs}}, N_b$

The server automaton, given in the Figure 5.9, is only involved in the initial authentication and no addition to the automaton is done for the subsequent authentication part.

The server initially knows its shared keys between principals which is calculated by $K_{xs} = id\boldsymbol{x} + Key\_Base$. It performs key distribution between two principals (having ids $id1$ and $id2$), which are generated by $K_{ab} = id1 + id2 + Key\_Base$.

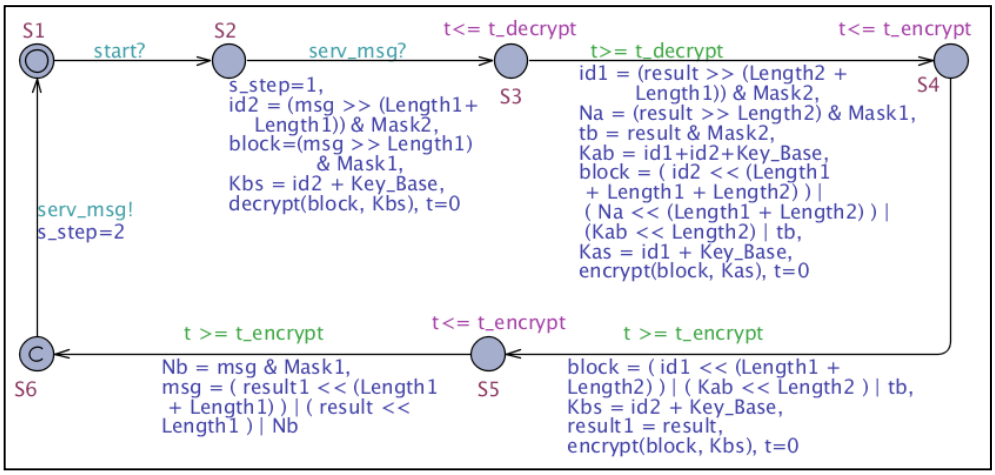

Figure 5.9. The Server Automaton

In the model, in order to reduce the state space of the intruder, some more guards are introduced in the Intruder automata, indicating the current step of the protocol execution. For this aim, the principals keep track of the protocol steps they have executed, in terms of the sent and received messages. The variables a_step, b_step and s_step are added to the initiator, responder and the server automata respectively, to use in these guards. The values of these variables denote that:

$a\_step = 1$ A has sent the protocol message 1
$a\_step = 2$ A has received the protocol message 3
$a\_step = 3$ A has sent the protocol message 4
$b\_step = 1$ B has received the protocol message 1
$b\_step = 2$ B has sent the protocol message 2
$b\_step = 3$ B has received the protocol message 4
$s\_step = 1$ S has received the protocol message 2
$s\_step = 2$ S has sent the protocol message 3

5.4.4. Dolev-Yao Intruder

The flaws of a security protocol are examined by modeling an *intruder* (also called *attacker*, *spy* or *enemy*) who wants to exploit the features of a protocol. The Dolev-Yao intruder model (Dolev and Yao 1981) is an easily applicable intruder model that is frequently used in the formal verification methods. This model assumes that the intruder has the full control of network and delivers the messages sent from one identity to another.

The Dolev-Yao intruder has the abilities to:

- **Deliver Messages -** transmit the message to the intended recipient without any modification.
- **Block Messages** - intercept a message by not delivering it to its recipient.
- **Decompose Messages -** decompose an overheard message and improve its knowledge using the constituent parts of the received messages. The message can still be delivered to intended recipient, without alteration.
- **Perform Encryption/Decryption -** encrypt or decrypt the information in its knowledge base. (Note that decryption is possible only if he knows the correct secret key. He also cannot guess the keys or the generated nonces.)
- **Compose Fake Messages -** derive new messages by composing some constituent parts or encrypted/decrypted blocks.

The intruder automaton should be able to represent the knowledge of the Dolev-Yao intruder which can be used to compose fake messages. The intruder's knowledge includes the identities of the principals, its own keys and nonces, origin and destination of all messages, the states of all principals. In addition since it can read all the messages, it also knows every part of the messages, everything it can generate by encrypting something with something that may be used as a key, everything it can generate by decrypting something (provided that it knows the correct key) and every concatenation of data it knows.

Our intruder automaton is composed of the pieces of automata each fulfilling the one of the capabilities listed above.

5.4.4.1. Delivering the Messages

The piece of automaton given in Figure 5.10 models the message transmission. Here, it is seen that the messages can be received by the network from the initiator over the $init\_msg?$ channel, from the responder over the $resp\_msg?$ channel and from the server over the $serv\_msg?$ channel where these signals are emitted by the principals when they have sent a message. On the other direction, these received messages can be transmitted to the initiator over the $init\_msg!$ channel, to the receiver over the $resp\_msg!$ channel or to the server over the $serv\_msg!$ channel. It can be seen that a

message can be sent to any principal using this automaton. Hence, besides the correct recipient, it is possible to send a message to any principal that the intruder wants. When one of the output signals are emitted by the network, the corresponding principal captures the signal and understands that the message is sent to him. Then, it reads the message which is kept in the global variable msg.

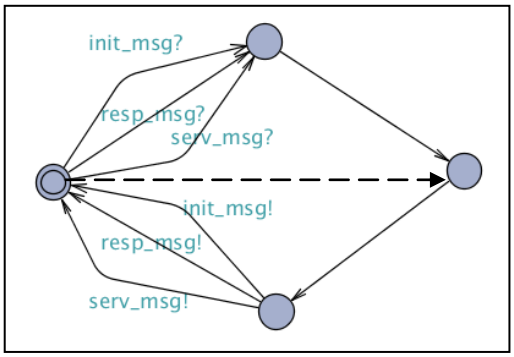

Figure 5.10. A Simple Network Model

Note that the intruder may also have the ability to create and send its own messages; hence the dotted transition can also be employed. In our protocol model, if the intruder needs to send its own message, then it receives a signal over a channel, discards it and sends its own message. However, in the full protocol model including both the initial and subsequent parts, the intruder may learn the secret key in the initial part and use it by initiating the execution of subsequent part. To initiate a communication, this transition is needed and employed in the complete protocol model, which is given in section 5.8.

5.4.4.2. Decomposing Messages

The intruder can capture the messages, decompose them into its constituent parts and add to its knowlegde base. For example, when the message A, N_a is captured by the intruder, it has the ability to read the initiator's identity and its nonce N_a, hence, learns them. The piece of timed automata given in Figure 5.11 is used to enable the intruder improve its knowledge by adding the information extracted from the messages sent.

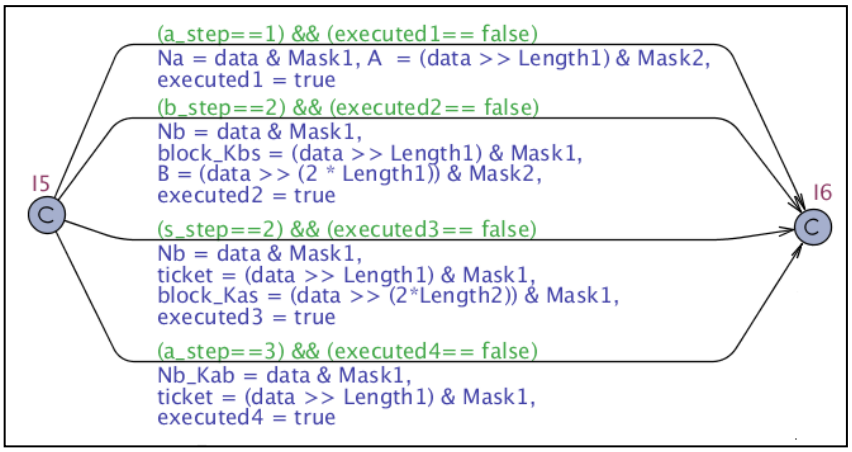

Figure 5.11. Intruder Decomposing Messages

In fact, the intruder can nondeterministically take one of the transitions from $I5$ to $I6$ to read a message. However, this increases the state space so large that it gets impossible to verify some queries. Because of that, in order to limit the possible number of transitions, we added some guards to these transitions. This guards restrict the transitions to be taken only if the corresponding protocol step has been executed. It is important that these limitations do not lessen the power of the intruder, but decreases the number of infeasible executions which make the state space grow enormously.

In the figure, there are four transitions that may be taken when one of the four steps of the protocol's initial authentication is executed. For example the intruder can examine the message A, N_a only if $a\_step = 1$ which means that A has sent the first protocol message A, N_a. It is also avoided to take the same transition again once the intruder added the contents of a message to its knowledge base.

5.4.4.3. Performing Encryption and Decryption

The intruder has the ability to generate a nonce, do encryption and decryption using the parameters in its knowledge base. Here it is important to model the intruder such that it cannot guess the principals' nonces, secret keys and cannot decrypt a message without knowing the key (due to the perfect cryptosystem assumption).

The piece of automata given in Figure 5.12 selects the parameters to apply encryption or decryption. It can use any variable in its knowledge base by nondeterministically selecting the plaintext/ciphertext and the key. Again, to avoid a state space explosion, the guards are used which allow the use of a variable only if it is

set (it is not set if the value of a variable is 0. See Table 5.1 for the possible values of the variables). In this automaton, A, B and I are the identities, N_i is the nonce generated by the intruder.

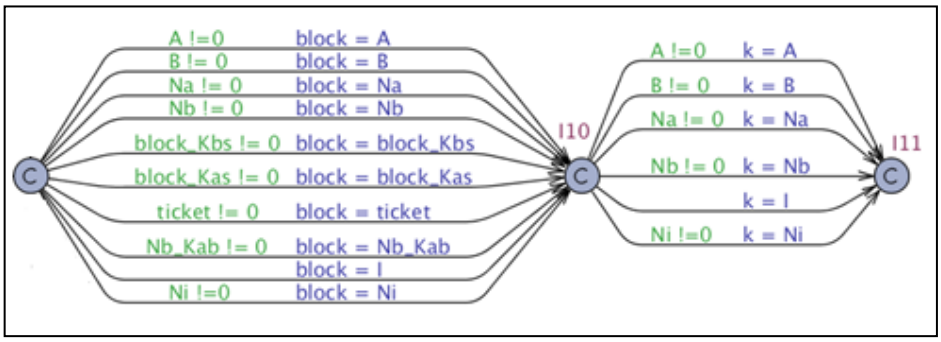

Figure 5.12. Intruder Parameter Selection for Encryption/Decryption

5.4.4.4. Composing Fake Messages

The Dolev-Yao intruder can also create new messages and inject them into the network. So, the model will be expanded with a message creation part for the intruder, where it can populate each constituent part of a message with some known information. Figure 5.13 shows that piece of timed automaton. A local variable $data2$ is used to keep some constituent parts which is initially 0. While creating a new message, $data2$ can be set to any variable that the intruder knows. Then, in order to append a new content to the message, it can be shifted left for $Length1$ or $Length2$ times depending on the length of the content to be appended.

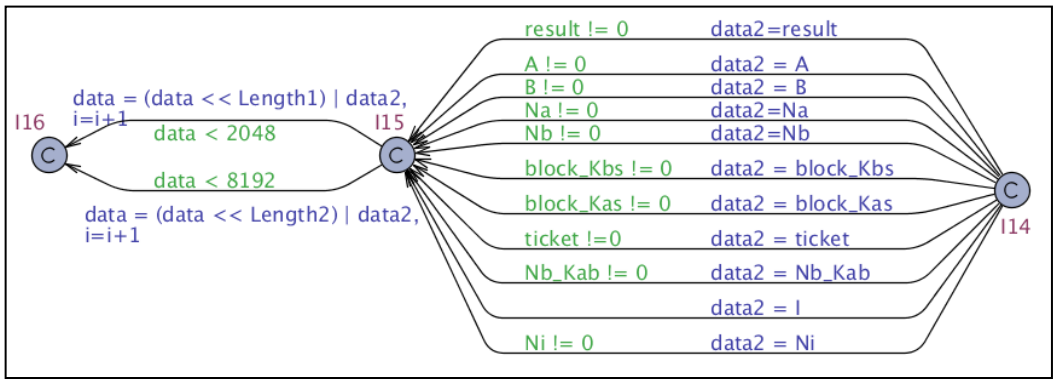

Figure 5.13. Intruder Deriving New Messages

Note that in section 5.4.2, it is mentioned that only two length variables are used to reduce the state space of the intruder. If it was not reduced, there would be more number of transitions from $I15$ to $I16$ which causes the state space to increase enormously (see Figure 6.2).

By using this message creation part, the intruder can generate any messages using its knowledge or the result which is obtained as the result of an encryption or decryption operation.

5.4.4.5. The Dolev-Yao Intruder Model

Our Dolev-Yao intruder model (see Figure 5.14) combines all these features where the intruder can deliver, block, decompose messages, perform encryption/decryption on nondeterministically selected parameters and generate fake messages using its knowledge base.

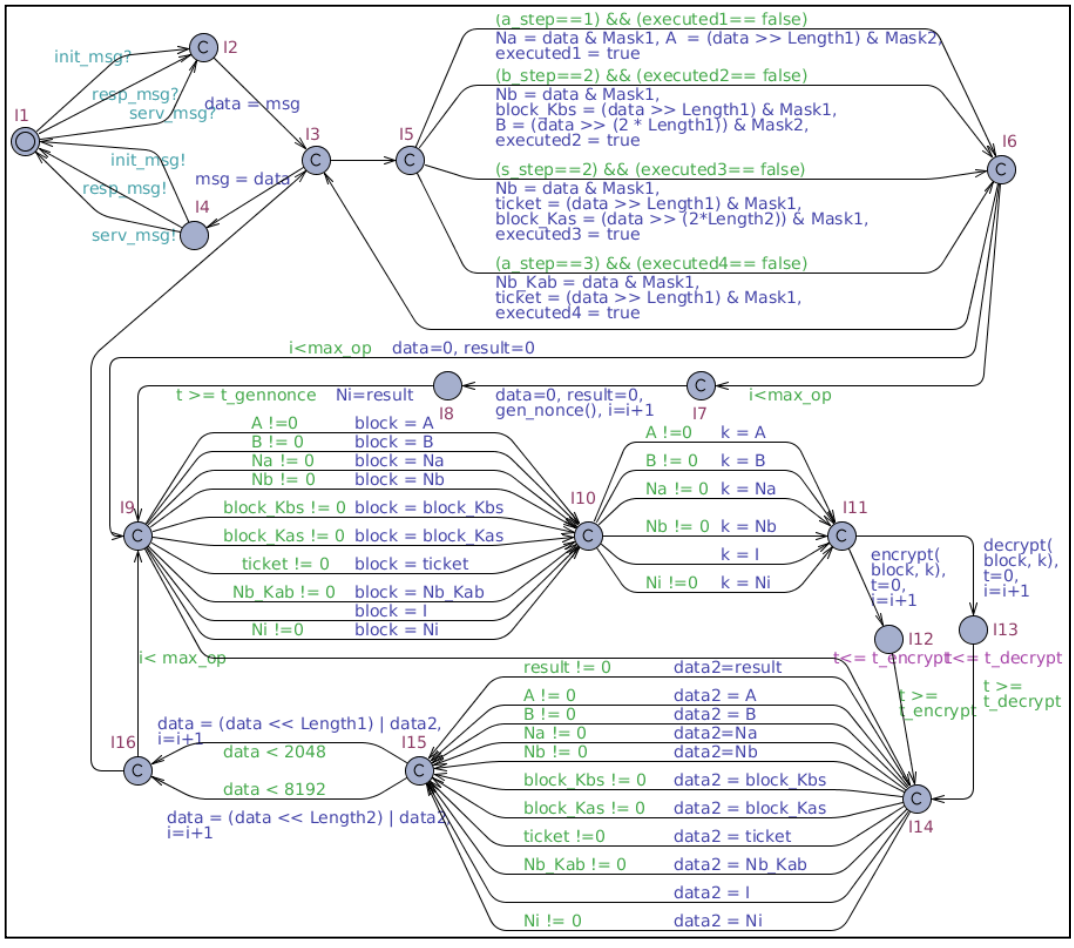

Figure 5.14. The Intruder Model for the Initial Authentication Part

The automata parts of the intruder are merged in a way that it enables the possible executions of the intruder. For example, the intruder can append more than one encrypted blocks to a new message. Then, a transition should be added from the message creation part to the parameter selection part for encryption/decryption.

The model contains many loops and it can perform as many operations as it wants. This enlarges the state space so that the model verification goes out of memory. So, the number of operations that intruder may perform is restricted by the variable max_op. Nonce generation, encryption, decryption and shifting for message creation increments the number of operations performed. The intruder cannot perform more than max_op number of these operations. Note that this number should not be so large to cause state space explosion and should not be so small to by-pass a possible attack.

In addition, the model should prevent possible errors. For example a possible error caused by data loss may occur when the data is right shifted although its rightmost bits contain data. This can be avoided by using a guard that checks if the rightmost bits to be shifted contain data or not.

In the generation and explanation of the model, a possible state space explosion is frequently mentioned and considered. It is a very struggling problem for model checking that we came up with which is explained in section 6.2.1 in detail.

5.5. Validation and Verification of Neuman-Stubblebine Initial Part

Validation of a model is concerned with building the *right model* that correctly represents the behaviors of the real world system. The generated model should implement the protocol execution in order to have the correct verification results.

Verification of a model is concerned with building the *model right*. A security protocol must satisfy some requirements in order to provide a secure communication between the principals. However, many protocols are shown to be flawed. Hence, a security protocol should be verified to check if it satisfies its requirements.

This section gives the validation and verification results for our Neuman-Stubblebine Repeated Authentication Protocol model obtained using UPPAAL.

Note that, for the initial part, the automata for the principals are named as $Initiator\_1$, $Responder\_1$, $Server$ and $Intruder\_1$; for the subsequent part, they are

named as $Initiator\_2$, $Responder\_2$ and $Intruder\_2$. For the complete model, these automata are merged as the $Initiator$, $Responder$, $Server$ and $Intruder$ automata.

5.5.1. Parameters and Configurations used in the Case Study

Verification of a specification can be performed by using either UPPAAL's graphical user interface or the stand-alone command line verifier. In this study, the verifier is executed using the command line, which is more appropriate for verifying large tasks. The verifier reads in the $.xml$ file (which is automatically generated by the tool during the graphical design) and checks whether it satisfies the query in the file given in a file with $.q$ extension. The following command form is used execute the verifier from the command line:

$$./verifyta\ [OPTIONS]\ <xml\_file><query\_file>$$

The options to specify may be related to state space representation, state space reduction, search order and trace options (Behrmann and David 2004). Since our model is large and needs large amount of memory, the following options to reduce memory consumption is used in this study (the non-listed properties are used with their default stand-alone command line verifier configurations):

- *State Space Representation:* Minimal Constraint Systems
  This representation uses less memory than DBMs.
- *State Space Reduction:* Aggressive Space Optimization
  This optimization may take more time but uses less memory by decreasing the number of states stored. ($-S2$ option)
- *Search Order:* Breadth First Search
  Depth first search cannot complete the verification of some of our queries since it runs out of memory where breadth first search generally finds out the diagnostics traces faster.
- *Trace Options:* Generate Some Trace
  The diagnostic traces of each property are generated and written in a $.xtr$ file which can be read by the simulator so that they can be viewed on the model. In this study, $-t0$ and $-t1$ options are used to generate some or shortest traces.

In addition, $-u$ option is used to see the number of states stored and explored during the verification process. To measure the time passed during the verification, the $time$ command in Linux is used which prints the execution time of a process. As an example, the following line can be executed to verify the queries in $initial.q$ on the model in $initial.xml$ and writes the diagnostic trace into the $traces-n.xtr$ file.

$$time\ ./verifyta-S2-t1\ -f\ traces\ initial.xml\ query1.q\ -u$$

The verification is performed on Ubuntu 9.10 Operating System on Intel Core2 Duo $P7350$, 2.00 GHz processor and $4GB$ of RAM.

5.5.2. Validation and Simulation of the Model

To ensure the correctness of our model, it is validated by checking whether the protocol works as specified. In a successful run of the protocol, the initiator authenticates itself to the responder at the end of the protocol execution, and both of these principal's $finish1$ variables should be set to 1. Query 1 is used to test whether the model is able to generate the correct run of the protocol.

Query 1: *Is such a state reachable where both the initiator and the responder finish the protocol execution?*

$$E\langle\rangle\ Initiator\_1.finish1\ \&\&\ Responder\_1.finish1$$

Query 1 is satisfied and the correct protocol execution is simulated successfully.

5.5.3. Verification of the Neuman-Stubblebine Initial Part

In this study, the correctness of an authentication protocol is aimed to be verified based on the goals of an authentication protocol. Two high level goals for an authentication protocol are listed as follows in (Woo and Lam 1994):

- *Authentication:* For each principal, after the successful run of the protocol, it should be assured that it is talking to the principal in its mind.
- *Key establishment:* A shared secret becomes available to the principals, for subsequent cryptographic use.

The possible attacks for the Neuman-Stubblebine protocol are analyzed by writing specifications derived from these authentication goals. In order to examine the protocol goals given above, the *correspondence* and the *secrecy* properties should be verified.

Correspondence means that the execution of different principals in an authentication protocol proceeds in a lock-stepped fashion. While the authenticating principal finishes its part of the protocol, the authenticated principal must have been present and participated in its part of the protocol.

Secrecy property specifies that a distributed session key cannot be discovered by the intruder. In the analysis of these goals, if an attack is found on a protocol, it is inferred that the protocol is incorrect since it does not satisfy the properties that it is intended for.

This section gives these specifications and the UPPAAL queries that we used to check whether our protocol model satisfies these properties.

Query 2: *Is such a state reachable where the responder has finished but the initiator has not finished the initial protocol execution?*

$$E\langle\rangle\ Responder\_1.finish1\ \&\&\ (!\,Initiator\_1.finish1)$$

Query 2 is related to the correspondence property. Here, it is used the fact that this property is not satisfied when the responder finishes the protocol execution although the initiator has not executed its part. In such a situation, one can say that an intruder has sent fake messages to the responder and attacked to the protocol. However, this property is satisfied. It means that the intruder caused the responder to finish its run by sending fake messages and we have found an attack on the protocol.

The diagnostic trace generated by UPPAAL (written in an .xtr file) is loaded into its simulator to be able to view the transitions and the operations performed. The transitions of the principal automata, the content of the msg variable and the operations to create the message are examined to find out the execution of the attack scenerio.

From the diagnostic trace, it is seen that this is the execution trace of the type flaw attack given in section 5.3. In this execution, after the responder sends the message to the server, the intruder (in Figure 5.14) extracts the information in messages 1 and 2 in the transitions from $I5$ to $I6$, and learns N_a, N_b, and $\{A, N_a, t_b\}_{K_{bs}}$(kept in its variable $block\_Kbs$). It skips the protocol message 3. To create a fake message, it takes $block\_Kbs$ into the $data2$ variable between the states $I14$, $I15$ and $I16$. Then, it selects

N_b as $param1$ and N_a as $param2$, to encrypt N_b with N_a. It composes this encrypted block with $block\_Kbs$ and sends it to the responder as the protocol message 4. In this attack, B accepts the nonce N_a as the key K_{ab}. As it is seen, a type flaw attack (substitution of a different type of message field) can be detected since the types of constituent parts for the encryption, decryption or message creation operations are not restricted. The intruder has the ability to make a composition of any of these variables that may be accepted by the responder as a correct message.

The next query is related to the secrecy property. The next query verifies the distribution of the key (at the end of the protocol execution, the secret key K_{ab} is generated by the server and distributed to the initiator and the responder) and the secrecy of this key by checking whether it can be learned by the intruder. The check is performed using the $data2$ variable in order to include all the decomposed pieces of the messages and their encryptions or decryptions.

Query 3: *The execution of the protocol run leads to the fact that, the secret key generated by the server is distributed to the protocol principals, and it cannot be learned by the intruder.*

$(Responder\_1.finish1 \;\&\&\; Initiator\_1.finish1) -\!\!\rightarrow \big((Responder\_1.Kab == Server.Kab)\&\&(Responder\_1.Kab == Initiator\_1.Kab)\&\&(Server.Kab != Intruder\_1.data2)\big)$

The property is satisfied, meaning that all the executions where the initiator and the responder finished the initial part lead to the equivalence of Kab s owned by them and this secret key cannot be obtained by the intruder. (Note that this property is true for the correct execution of the protocol in which both the initiator and the responder finishes their protocol execution.)

In the next queries, the aim is to find out whether the timing information can be used to analyze the attacks on a protocol. The timeout intervals - the time periods that a principal waits for a message - can be used for this purpose. If a message comes earlier than the required time to prepare a message (depending on the encryption/decryption times), then it can be said that the principal has received a fake message (Corin, et al. 2004, 2007).

In a normal run (see Figure 5.15.a), assuming the time to create or read a message is negligible, the timeout for the responder is:

$$Responder\_1.timeout \geq Server.t\_decrypt + 2 \times Server.t\_encrypt + Initiator\_1.t\_decrypt + Initiator\_1.t\_encrypt$$

1. A sends message 1 to B
2. B performs encryption, sends message 2 to S
3. S performs decryption, then two encryptions, sends message 3 to A
4. A performs a decryption and an encryption, sends message 4 to B

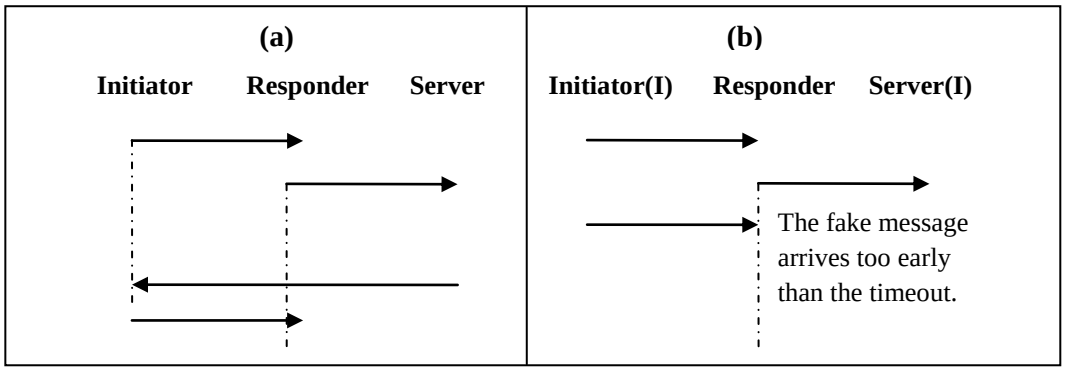

Figure 5.15. Normal (a) and Attacked (b) Message Flows of the Protocol

However, in a flawed run, the message can be received in a shorter time: $Responder\_1.timeout \geq Intruder\_1.t\_encrypt$ (illustrated in Figure 5.15.b, where the letter I in paranthesis indicate that the message comes from/goes to the intruder).

1. A sends message 1 to B (intruder reads the content)
2. B performs decryption, sends message 2 to S
3. Intruder captures and reads the message, performs an encryption, creates and sends a fake message 4 to B

In our model, it is assumed that the time needed for the encryption and decryption are the same for all principals. To measure the duration of time until the message is received, a local variable t_waited is used which is incremented at each time unit the responder waits (see Figure 5.8, state $B7$). Query 4 tests for a possible attack using the fact that if the message comes earlier than the required time for the cryptographic operations, then we can say that there is an attack on the protocol.

Query 4: Is such a state reachable where the responder has finished the protocol execution but the message has been received in a shorter time than the required time for the correct protocol execution?

$E\langle\rangle\ Responder\_1.finish1$ && $(Responder\_1.t\_waited < (Server.t\_decrypt + (2 \times Server.t\_encrypt) + Initiator\_1.t\_decrypt + Initiator\_1.t\_encrypt))$

This query is satisfied meaning that such a state is reachable. The diagnostic trace causing an earlier message is the execution trace of the attack detected in Query 2. Hence, it can be inferred that the attacks can also be detected by examining the quantitative timing information of the protocol.

In some protocols, a constraint on $B.timeout$ can be used to prevent a specific attack. For example, if the intruder needed more number of encryptions and decryptions than the normal run to perform an attack, then it would be possible to limit $B.timeout$ in a way that leaving no space for the executions of the intruder (such as the example given in (Corin, et al. 2007)). However, this is not the case in our protocol. The constraint $B.timeout \geq 2 \times t\_encrypt + 2 \times t\_decrypt$ does not prevent this attack since the intruder can wait for a while before sending the message.

In an authentication protocol, the timeout intervals can be examined for both the initiator and the responder. As we mentioned in section 5.4.3, only the waiting time for the responder automaton is included in our model since it is provides the detection the attack of the protocol and we have state space explosion problem when such transitions are included in both principals.

The timeout for the initiator can be analyzed by modifying the initiator automata in Figure 5.6 as in Figure 5.7 and removing the transitions counting $t\_waited1$ from the responder automata to avoid state space explosion. On that model, Query 5 can be used to test whether the initiator gets the protocol message 4 earlier than the required time for the creation of the message.

Query 5: Is such a state reachable where the initiator has finished the protocol execution but the message has been received in a shorter time than the required time for the correct protocol execution?

$E\langle\rangle\ Initiator\_1.finish1$ && $(Initiator\_1.t\_waited < (Responder\_1.t\_encrypt + Responder\_1.t\_gennonce + Server.t\_decrypt + (2 \times Server.t\_encrypt\ )))$

The query is not satisfied, hence it is not possible for the initiator to get the protocol message 4 earlier than the required time for the correct execution.

5.6. Modeling Neuman-Stubblebine Subsequent Part

Because of the fact that the intruder model increases the state space so enormously and it may cause state space explosion, the initial and subsequent authentication parts are analyzed individually. Up to now, the initial part of the protocol is modeled and verified. This section gives the model for the subsequent authentication part which involves only the initiator, responder and the intruder.

In the subsequent part, the knowledge acquired in the initial part should be included in the model as if the principals already know them. In the initial part, the initiator learns the shared key K_{ab}, the ticket $\{A, K_{ab}, t_b\}_{K_{bs}}$ to be used in the subsequent part. Hence, this information should be added to the initiator's knowledge base. Similarly, also the responder should know K_{ab} to communicate with the initiator in case the ticket check is successful.

In addition, the $plain$ and the key arrays should contain the values for the already encrypted blocks. Then, these arrays should contain the corresponding plaintext and the key for the ciphertext ticket to enable the responder to decrypt the ticket and check its correctness. In this second part, Init automaton is expanded with the initialization of these values as given in Figure 5.16. In the execution of the initial part, the block A, K_{ab}, t_b corresponds to 117, $K_{bs} = 12$, and the encrypted block $\{A, K_{ab}, t_b\}_{K_{bs}}$resides in the index 6. Hence, the 6^{th} index of $plain$ and key arrays should keep the corresponding values in order to make it decryptable by the responder.

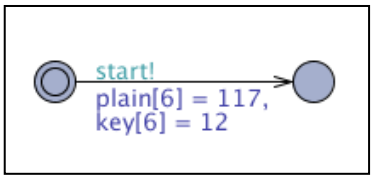

Figure 5.16. The Init Automaton for the Subsequent Part

The model includes the initiator and the responder both involving in all three steps of the subsequent part of the protocol.

1. $A \rightarrow B: \ N'_a, \{A, K_{ab}, t_b\}_{K_{bs}}$
2. $B \rightarrow A: \ N'_b, \{N'_a\}_{K_{ab}}$
3. $A \rightarrow B: \ \{N'_b\}_{K_{ab}}$

The automata for the subsequent part are constructed as the continuation of the initial part in order to make them easily combined. Figure 5.17 and Figure 5.18 give the initiator and the responder automata respectively.

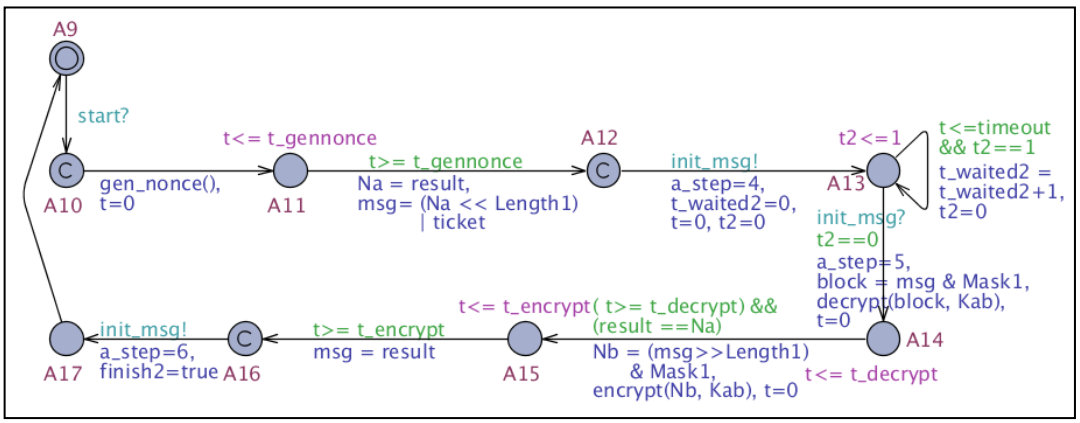

Figure 5.17. The Initiator Automata for the Subsequent Authentication Part

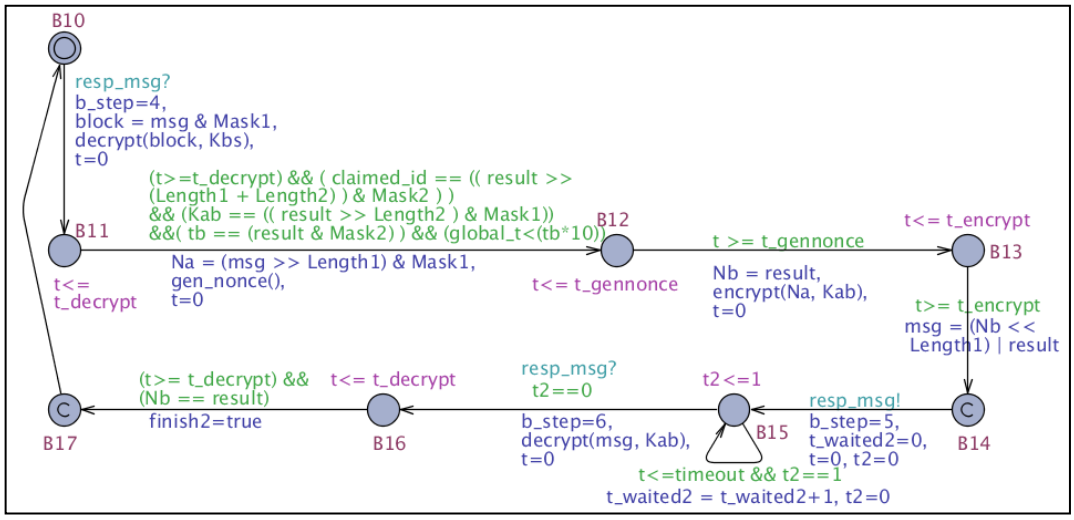

Figure 5.18. The Responder Automata for the Subsequent Authentication Part

The global clock variable $global\_t$ is employed in the model to keep the time elapsed from the beginning of the protocol execution, and used for testing the key expiration time. The ticket expires when $global\_t$ is greater than the expiration time. When the responder gets the first message, it checks the validity of the ticket. If it is still valid and the message content is correct, it continues the protocol execution. Note that

the expiration time is checked by comparing the $global\_t$ with $tb \times 10$ instead of tb, due to the design considerations, as explained in section 5.4.2.

The intruder model for the initial part is extended so that in addition to the protocol messages of this part, it can also read the messages of the subsequent part, decompose them, perform encryption/decryption and compose fake messages as given in Figure 5.19 (The dotted empty transition from state I1 to I3 is not employed in this subsequent model; but it is needed and used in the complete intruder model).

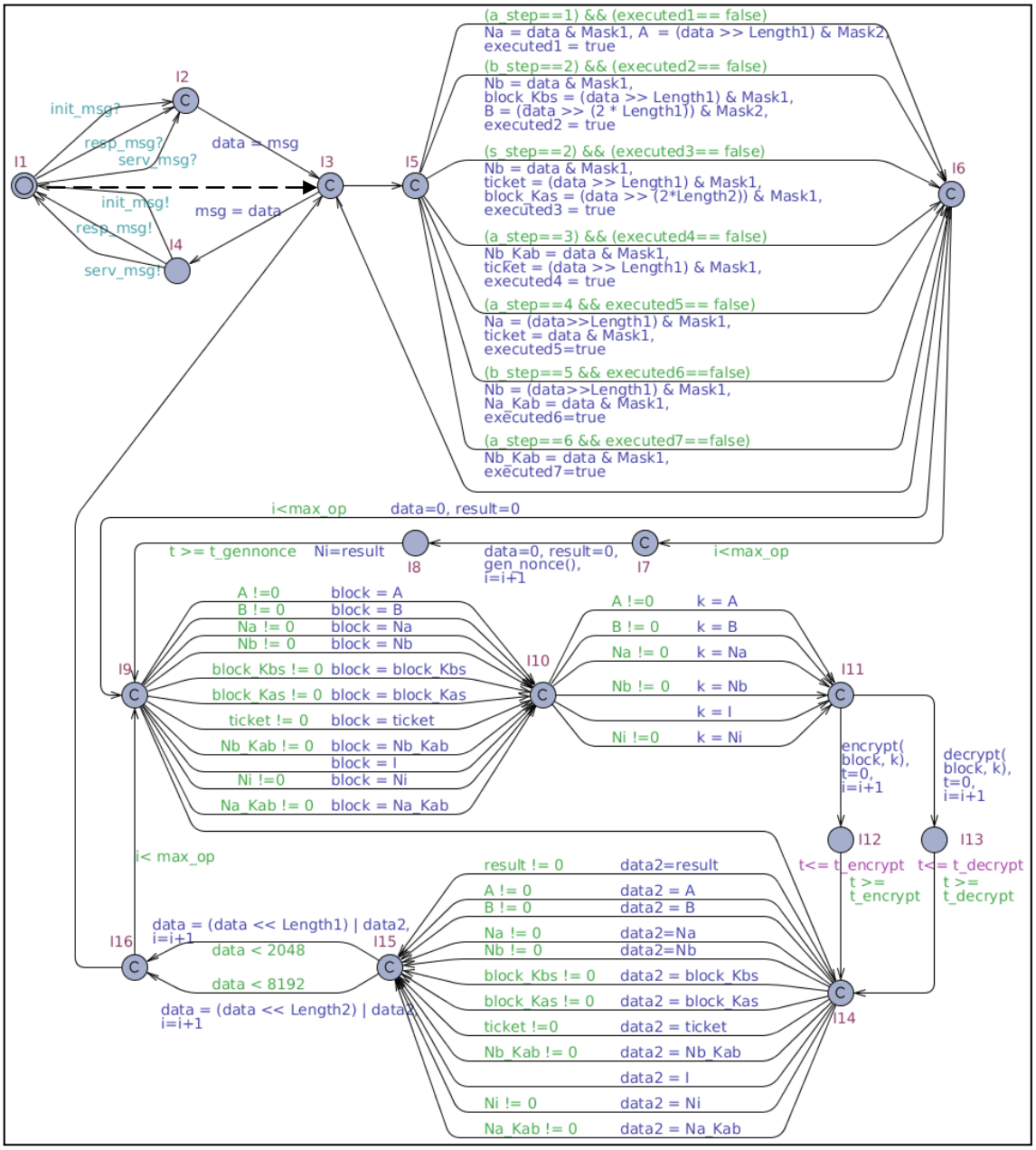

Figure 5.19. The Intruder Automata for the Neuman-Stubblebine Protocol

5.7. Validation and Verification of the Subsequent Part

The following query is used to validate the model by checking whether the principals can both finish the execution of the subsequent part. The property is satisfied and the correct protocol execution is simulated.

Query 6: *Is such a state reachable where both the initiator and the responder finish the protocol execution?*

$$E\langle\rangle\ Initiator\_2.finish2\ \&\&\ Responder\_2.finish2$$

Similar to the Query 2, the following query is related to the correspondence property and checks whether there is a run such that the responder finishes protocol execution although the initiator has not.

Query 7: *Is such a state reachable where the responder has finished but the initiator has not finished the initial protocol execution?*

$$E\langle\rangle\ Responder\_2.finish2\ \&\&\ (!\ Initiator\_2.finish2)$$

The property is not satisfied meaning that no such state is reachable. However, the protocol is exposed to a parallel session attack given in section 5.3 which may yield a protocol run not satisfying the correspondence property.

In this model, one automaton for the initiator and one automaton for the responder are used allowing them to execute just one protocol run at a time. Hence, the model cannot detect the parallel session attack that occurs when two protocol runs are executed concurrently and messages from one run are used to form fake messages in another run.

In this attack, the initiator sends the protocol message 1 to the responder and goes to state $A13$. The responder takes the message, sends the initiator message 2 and goes to state $B15$ where it waits for the protocol message3. However, in the parallel session attack, the intruder reads these messages, composes a new message as message1 of another protocol run and sends it to the responder which should be accepted in state $B10$. Since the responder is currently in state $B15$, it cannot accept this message1 of another run and the model cannot generate this flawed scenerio.

Query 8 checks the secrecy property for the subsequent part in a similar way as for Query 3. The query is satisfied meaning that in a correct protocol execution, the secret key shared by the responder and the initiator cannot be learned by the intruder.

Query 8: *The execution of the protocol run leads to the fact that, the secret key shared by the responder and the initiator cannot be learned by the intruder.*
$(Initiator\_2.finish2 \;\&\&\; Responder\_2.finish2) \; -\!-\!> \; ((Responder\_2.Kab == Initiator\_2.Kab)\&\&(Responder\_2.Kab != Intruder\_2.data2))$

As for the analysis of the timing of the message arrivals, the next queries check whether a message arrives earlier than expected, similar to queries 4 and 5.

Query 9: *Is such a state reachable where the responder has finished the protocol execution but the protocol message 3 has been received in a shorter time than the required time for the correct execution of the protocol?*
$E\langle\rangle\; Responder\_2.finish2 \;\&\&\; (Responder\_2.t\_waited2 < (Initiator\_2.t\_decrypt + Initiator\_2.t\_encrypt))$

Query 10: *Is such a state reachable where the initiator has finished the protocol execution but the protocol message 2 has been received in a shorter time than the required time for the correct execution of the protocol?*
$E\langle\rangle\; Initiator\_2.finish2 \;\&\&\; (Initiator\_2.t\_waited2 < (Responder\_2.t\_decrypt + Responder\_2.t\_gennonce + Responder\_2.t\_encrypt))$

These queries are not satisfied meaning that the messages do not arrive earlier than expected. As it is seen, although it may be possible to have premature messages in case of the parallel session attack, our model is unable to detect them since it cannot model the concurrent execution of more than one protocol executions.

5.8. Combining the Initial and Subsequent Parts

This section gives the complete automata for the initiator and the responder that model the execution of both initial and subsequent authentication parts. The intruder model for the complete protocol execution is same with the one given in Figure 5.19 with the additional empty transition from state $I1$ to $I3$. Figure 5.20 and Figure 5.21 give the complete protocol model for the initiator and the responder.

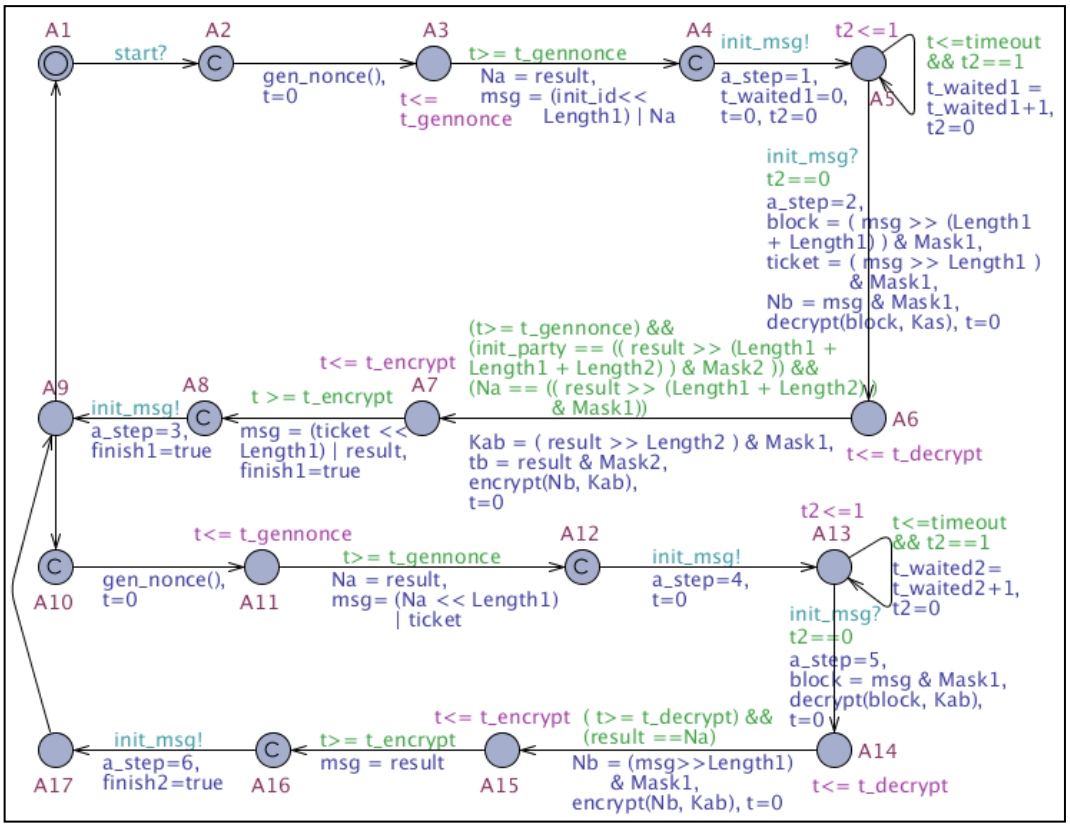

Figure 5.20. The Initiator Automata for the Neuman-Stubblebine Protocol

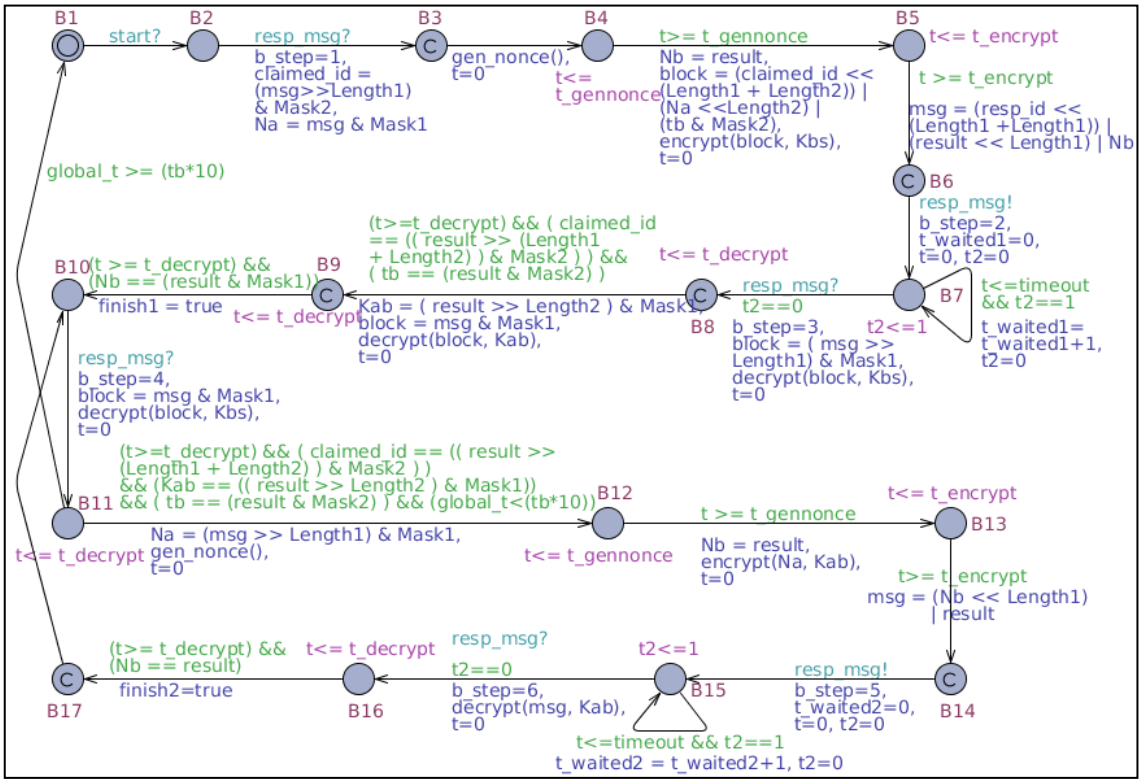

Figure 5.21. The Responder Automata for the Neuman-Stubblebine Protocol

In this complete model, the automata designed for initial and subsequent authentication are merged by involving the key expiration time tb. The global clock $global\_t$ is used to test the key expiration time, which is reset by the Init automaton so that $global\_t$ keeps the time elapsed from the beginning of the protocol execution. The execution expires when $global\_t > tb \times 10$, as given in Figure 5.21.

If the ticket expires, then the responder does not accept the protocol message 1 of the subsequent part and goes to the initial authentication to get a new session key. The initiator has two transitions after the execution of the initial part which may continue with the subsequent part or go back to the initial part to get a new session key and a ticket, depending on the responder's behavior.

The protocol execution can be executed more than once by using the Init automata emitting $start!$ signal in a loop and reseting the clock $global\_t$. So, it is possible to simulate the model continuing by the second iteration of the protocol, starting from the initial authentication part. Our limitation here to simulate it once is related to the limitation of the size of encryption/decryption arrays and the nonce values can be generated (see section 6.2.2), which are restricted to have smaller state space and to be able to verify our queries.

Table 5.2 lists the summary of the verification results for the queries including the number of states stored, number of states explored, the user and the system time for the queries performed in our case study (using random seed values).

Table 5.2. Verification Results for the Queries

Query	Model	Seed	States stored	States explored	Real time	Satisfied
1	Initial	1279609757	444648	651457	0m9.859s	Yes
2	Initial	1279609786	444648	651457	0m11.549s	Yes
3	Initial	1279609964	9045526	13472396	2m51.765s	Yes
4	Initial	1279610160	444648	651457	0m11.670s	Yes
5	Initial (*)	1279610225	1828010	2978888	0m31.286s	No
6	Subs.	1279610723	15472	24182	0m0.360s	Yes
7	Subs.	1279610746	12396981	20603774	4m1.857s	No
8	Subs.	1279611000	16404642	24710183	5m29.892s	Yes
9	Subs.	1279611343	12396981	20603774	4m2.420s	No
10	Subs.	1279611634	12396981	20603774	4m3.139s	No
1,2,3,4, 5,6,7, 8,9,10	Complete	out of memory				

The system marked with (*) contains the part of timed automata for the initiator timeout given in Figure 5.7, instead of the responder timeout part, since the query checks the timing for the initiator. Query 10 also checks the timing for the initiator on the subsequent model which does not have a state space explosion problem and includes the timing for both principals.

As it is seen from the table, the verifier goes out of memory in all queries during the verification of the complete model. This is caused by the state space explosion problem which is explained in the next chapter.

CHAPTER 6

ANALYSIS OF THE CASE STUDY: TIMED AUTOMATA AS A VERIFICATION TOOL FOR SECURITY PROTOCOLS

6.1. Benefits of the Model

Timed automata formalism provides an easily understandable model for the verification of concurrent systems. The state transitions give good insight about the real system so that the analysis can be performed in an easier way. Our protocol model mimics the protocol execution and allows studying on some example scenarios and counter examples by simulating and analyzing the model visually.

Timed automata modeling is suitable for the systems that can be viewed as a parallel composition of processes. To model a security protocol, each principal can be modeled as an automaton which represents the behavior of the principal who involves in protocol steps by sending and receiving messages, generating nonces, performing encryption and decryption, reading and checking message contents. The communication between the principals can also be easily represented by using the synchronization channels between automata. The emitting and receiving signals with a global message variable successfully model the message transfer.

C-like data structures and functions supported by UPPAAL are very useful for modeling cryptology. They provide the cryptographic operations to be held in the local functions of each principal which is close to the real world. In addition, these variables allow us to model the knowledge bases of the principals so that they know their ids, newly generated nonces, message contents…etc.

One of the most important advantages of timed automata is its ability to model time-sensitive systems. Since the correct functioning of the security protocols depend on some timing relationships between the events, timing information is important in their analysis. In our model, the network delays are considered to be negligible and the timing information for the time consuming operations such as nonce generation,

message encryption and decryption, the timeout intervals for the messages, and the expiration time of the session key, are included.

Automatic verification of the model is another advantage since it needs less or no human intervention. Once the required properties are specified, the verification is performed automatically by the model checking tools.

6.2. Challenges of the Model

6.2.1. State Space Explosion Problem

The most challenging problem of automatic verification and model checking is the usage of large amount of time and memory. Because of the fact that model checking performs reachability analysis to verify properties by exploring all possible states, it has to keep the clock values and the control structure of the automata. This results in the usage of huge amount of memory and it is called the *state space explosion* problem. This problem occurs on the systems with many components having transitions in parallel, since the number of states in a transition system grows exponentially with the number of the components.

In a large system such as the protocol in our case study, the verifier goes out of memory because of the state space explosion. It is a serious problem; because, the verification process cannot be completed whether an erroneous state that does not satisfy the specified property has not been found before explosion. Such a situation is illustrated in Figure 6.1. The exploration starts from the initial states and continues as the expanding circles. After the state space explosion, it goes out of memory and crashes. In that case, the erroneous states may not be detected.

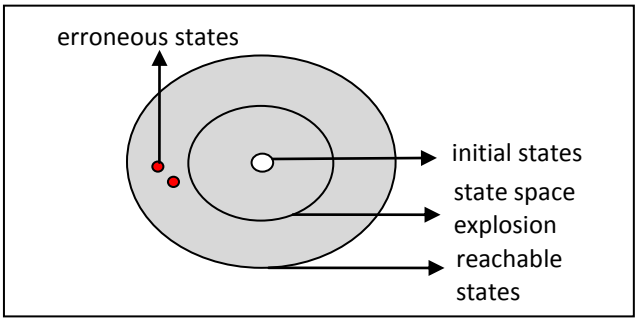

Figure 6.1. Reachability Analysis and State Space Explosion

This problem is also a current focus in the literature and some methods are proposed to overcome this problem. These can be classified as the methods to **(i)** reduce the number of states to explore, **(ii)** reduce the memory requirements needed for storing explored states, **(iii)** use parallelism or distributed environment, or **(iv)** exploring only part of the state space (Pelánek 2009). Some of these methods are provided or proposed as a future work for UPPAAL tool, and can be examined using the documentations in (Amnell, et al. 2001, Behrmann, et al. 2002, 2006). This section focuses on the modifications on our model to overcome this situation.

There are some studies in which some verification results cannot be obtained due to the fact that UPPAAL ran out of memory (Harrison, et al. 2007, Huber and Schoeberl 2009, Heidarian, et al. 2009). Also, in our study, state space explosion problem arises in some cases. For some queries, the verifier runs out of memory and crashes. It is not a solution to increase memory since UPPAAL is a 32-bit process which means that it cannot address more than 4GB of memory. Hence, some restrictions and limitations are applied in our model to reduce the state space:

- *Separate initial and the subsequent authentication parts:*

  The state space explosion problem is the reason why the initial and the subsequent authentication parts are modeled and verified separately. Because, the number of states is much larger in the complete protocol model that also employs some more transitions to ensure the continuity of the protocol execution. Since the queries cannot be checked on this model, the verification is performed for each part individually.
- *Simple Modeling and Eliminating the Redundant States:*

  The states that do not exhibit interesting behavior are eliminated and the unnecessary behaviors are not employed in the model since it may be very consumptive. In addition, all possible operations are performed in the updates of transitions to reduce the number of transitions and states.
- *Use of Committed States:*

  The use of committed states restricts the nondeterminism in the model. This kind of states are used in our model to guide the state space exploration and avoid the unnecessary interleavings of independent transitions, which can be considered as a simple form of partial order reduction. These states are not stored in the passed list and the interleavings of any state with a committed location is not explored.

This reduces the state space so much that changing even one state from committed to a not committed state may result in explosion.

- *Limit the Number of Operations:*

  The intruder model allows the attacker to perform unbounded number of encryption/decryption operations and composing new messages, resulting in a huge state space. To prevent the execution of unbounded number of these operations, the number of operations that the intruder can perform is limited by using the variable max_op in a way that the value is not so small to by-pass a possible attack.

- *Keep the number of clocks and variables as low as possible:*

  A symbolic state of a timed automaton includes the location vector, clock zone and variable valuations. So, reducing the number of variables provides us considerable savings. In our model, it is tried to use constant variables whose values do not change during the execution (such as the principal ids, initial knowledge of shared keys, etc.) and bounded integer variables which can be represented using less number of bits (such as the nonces and index variables used for encryption and decryption)

  For example, defining $Length1 = 4$ and $Length2 = 2$ instead of defining different length variables as $Nonce\_Length$, Key_Length, $Index\_Length$, $Agent\_Length$ and $Time\_Length$, saves us a few transitions from state to which might have ended up in exponential growth in possible next transitions and states. As it is seen in Figure 6.2, the first automaton with more number of length variables has five transitions where the other one has only two transitions.

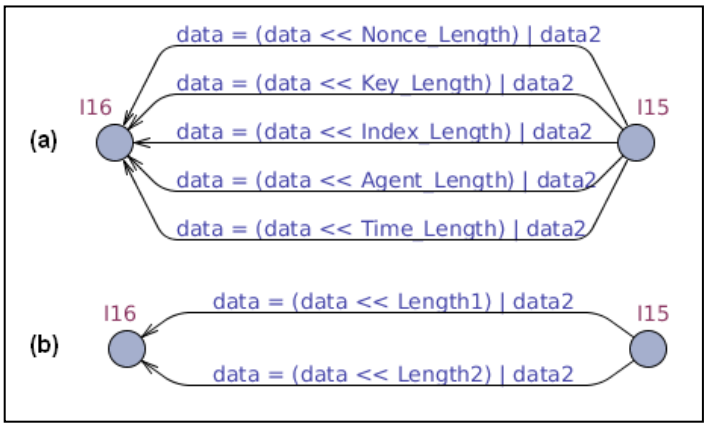

Figure 6.2. Reduced Number of Transitions

- *Adding Guards:*

  Some more guards are especially employed in the intruder model to reduce the number of available interleavings and nondeterminism without lessening the power of intruder, which caused an exhausting modeling phase.

  For example the parts of intruder automata in Figure 5.11, Figure 5.12 and Figure 5.13 involve some guards that prevent to take unnecessary transitions (It is unnecessary to try to extract a message where the intruder cannot learn anything new or to use an uninitialized variable. Hence, it is pointless to have these transitions as available transitions and increase the state space).
- *Make use of the UPPAAL verifier options:*

  The minimal constraint graphs and aggressive state optimization that decreases the number of stored states are used to reduce the memory consumption although it may increase time consumption which is less restrictive in our study.

These reductions provides considerably time and memory savings. However, the problem still exists for the verification of the complete protocol execution involving both the initial and the subsequent parts.

6.2.2. Collision of Variable Values

In a model, if it is possible to find an attack, then it should also be a possible attack on the real world. However; some flaws, although which is not an attack in fact, can be found because of some weaknesses or deficiencies of the model. In earlier phases of our study, such a problem had occured which is caused by the possibility of having same values for different data types in the model.

For example, the values of the principal ids, nonces, the index values representing encrypted blocks could coincide, resulting in the responder accepting wrong messages as they were correct since the values seem to be correct. An example situation is given in Figure 6.3.

In this execution, the responder B accepts an incorrect message because of the fact that the values of the message components seem to be true. In this example, the identity variables have the values $A = 1$, $B = 2$; and the $nonce$ is initially 0. The agents increment the value of this nonce variable and they get $Na = 1$, $Nb = 2$. The index values for the $plain$ and key arrays start from 0. In the first step of protocol execution,

A sends the message including its identity and nonce. B creates the protocol message2 (encrypts the block A, N_a, t_b with K_{bs} filling in the $0th$ index), to send it to the server. The intruder does not transmit it, and it also omits the third step. It encrypts B's identity with A (which is meaningless in a real execution); filling in the $1st$ index ($plain$ and key arrays become as in the figure). It sends a fake message including only A's identity. In real execution, this fake message is not accepted by B. However, in this deficient model, it may be identified as the correct protocol message 4. B reads the empty (zero) part of the message as $\{A, K_{ab}, t_b\}_{K_{bs}} = 0$, and $\{N_b\}_{K_{ab}}$=1, which are possible to be decrypted because of the collided values.

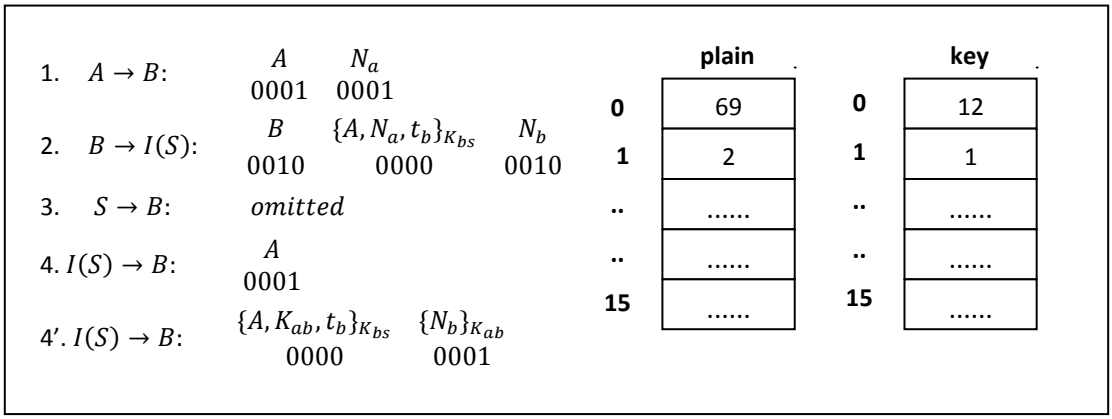

Figure 6.3. A Deficient Run Caused by the Collisions on the Variable Values

To prevent the problem, the integer values to be used for each type are predefined. By using the variables values given in Table 5.1, such number collisions do not occur. One disadvantage of this solution is the limitation on the numbers to be used since we use only 4 bits for the values of a component. This also limits the number of protocol executions since the nonce values to be generated and the number encryptions that can be performed are limited. One way to overcome this problem may be to increase the number of bits to represent these cryptology modeling variables which is in trade-off with state space explosion.

6.3. Possible Extensions

6.3.1. Retransmissions

The model can be further extended by modeling the retransmissions which involve resending of messages that may be damaged or lost. A message can be retransmitted if the response message does not arrive in a specific timeout interval.

To model retransmissions, an extra transition can be added from the waiting state to the sending state as it is seen in Figure 6.4 and Figure 6.5.

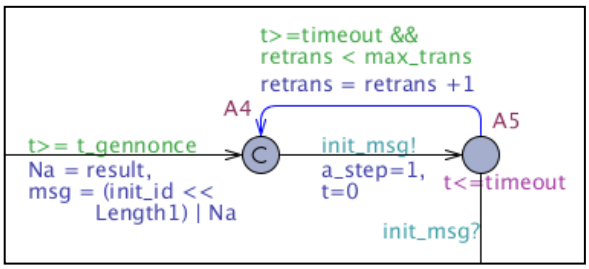

Figure 6.4. Extension for the Initiator Automaton for Retransmission

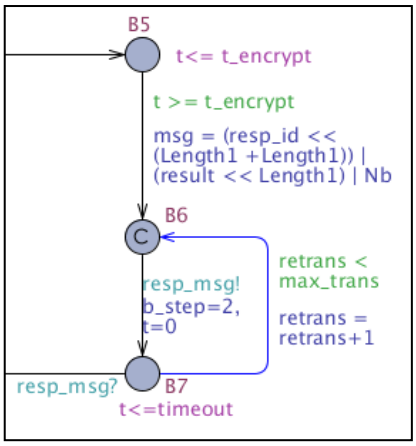

Figure 6.5. Extension for the Responder Automaton for Retransmission

The transitions from A5 to A4 and from B7 to B6 model retransmissions that make the principals resend the message. These transitions are allowed when the timeout is reached and the total number of allowed retransmissions is not exceeded. This model is closer to the real protocol execution; however, it increases the state space considerably.

6.3.2. Parallel Sessions

The subsequent part of Neuman-Stubblebine authentication protocol is exposed to a parallel session attack given in section 5.3.

The parallel session attack occurs when two protocol runs are executed concurrently and messages from one run are used to form fake messages in another run. However, our model is able to execute just one protocol run at a time. Consequently, our automata model which has one automaton for the initiator and one automaton for the responder, allowing them to execute just one protocol run at a time cannot detect the parallel session attack.

One idea may be to extend the network of timed automata by using more than one instantiations of the principal automata templates in the system definition to model different runs of the protocol. This can be done in a way that the instantiations of a principal should share the same knowledge base in order to behave as a single principal involving in different protocol runs. Nevertheless, besides having a huge state space, this does not provide us a way to analyze unbounded number of parallel protocol runs. The model and the protocol analysis can be further extended so that it is possible to examine unbounded number of parallel sessions.

CHAPTER 7

CONCLUSIONS

In this thesis, the objective is to study the timed automata model which introduces quantitative time information into the real time system verification and utilize it for security protocols. After the analysis of the timed automata theory and its implementation, a case study is performed on a repeated authentication protocol, by directly modeling it with timed automata. The time needed for the cryptographic operations, message timeouts and key expiration time are also employed in the model.

In summary, the following steps are performed for modeling the protocol:

- Each principal is designed as an automaton whose transitions mimic the protocol execution, which together make up a network of timed automata.
- The communication of the principals is provided using the shared variables and the synchronization channels between the automata.
- Dolev-Yao intruder model is used which also models the network. It is concentrated on the modeling of the intruder in a way that it has all of the abilities of the Dolev-Yao model and does not have large numbers of variables and the state transitions.
- The details of the cryptographic operations (nonce generation, encryption and decryption) are abstracted away. Cryptology modeling and message creations are employed in the updates of the transitions.
- Some simplifications and limitations are applied on the model to avoid state space explosion.

Then, the model is aimed to be verified to find out the possible attacks.

- The properties to be checked are specified based on the goals of an authentication protocol
- The properties are written in the query language of UPPAAL which performs automatic verification.
- The execution of the diagnostic traces are examined using the visual simulator.

It is concluded that the timed automata model for an authentication protocol can be used to examine the predefined goals of a protocol. Model checking of Neuman-Stubblebine authentication protocol with timed automata is able to find the type flaw attack in the initial part of the protocol, by analyzing the correspondence property. In addition, this attack can also be detected by using the quantitative time information.

However, the parallel session attack in the subsequent part, which occurs when two protocol runs are executed concurrently, cannot be detected since our model allows the principals to execute just one protocol run at a time. The model can be further improved by employing the parallel execution of more than one protocol runs so that it can detect the parallel session attacks.

In addition, similar to the other model checking methods, timed automata model suffers from the state space explosion problem that occurs in large models. In our case study, although the state space is tried to be reduced by using the verifier options and some limitations are applied on the model, the complete protocol model including both the initial and subsequent authentication parts cannot be verified since the verifier goes out of memory. Hence, some work should be devoted to overcome the state space explosion problem.

REFERENCES

Abdulla P.A., Deneux J., Ouaknine J. and Worrell J. (2005). "Decidability and Complexity Results for Timed Automata via Channel Machines", *Proceedings of 32nd International Colloquium on Automata, Languages and Programming (ICALP)*: 1089-1101.

Adams S., Ouaknine J. and Worrell J. (2007). "Undecidability of Universality for Timed Automata with Minimal Resources", *Lecture Notes in Computer Science (LCNS) 4763/2007, Springer*: 25-37.

Alur R. (1999). "Timed Automata", *11th International Conference on Computer-Aided Verification, Lecture Notes in Computer Science (LCNS) 1633/1999, Springer*: 8-22

Alur R. and Henzinger T.A. (1989). "A Really Temporal Logic", *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (IEEE Computer Society Press)*: 164-169.

Alur R. and Henzinger T.A. (1990). "Real Time Logics: Complexity and Expressiveness", *Proc. of the Fifth Annual Symposium on Logic in Computer Science (IEEE Computer Society Press)*: 390-401.

Alur R. and Dill D. L. (1994). "A Theory of Timed Automata", *Theoretical Computer Science 126*: 183-235.

Alur R. and T.A. Henzinger. (1992). "Logics and Models of Real Time: A Survey", *Proceedings of the Real-Time: Theory in Practice, REX Workshop, Springer*: 74-106.

Alur R., Courcoubetis C. and Dill L. (1990). "Model-checking for Real-time Systems", *Proceedings of the 5th IEEE Symposium on Logic in Computer Science*: 414-425.

Alur R., Courcoubetis C. and Dill D. (1993). "Model-checking in Dense Real-time", *Information and Computation 104*: 2-34.

Alur R., Courcoubetis C., Henzinger T.A. and Ho P.H. (1993). "Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems", *Lecture Notes in Computer Science (LCNS) 736/1993, Springer*: 209-229.

Alur R., Fix L. and Henzinger T.A. (1999). "Event-clock Automata: A Determinizable Class of Timed Automata", *Theoretical Computer Science 211*: 253-273.

Amnell T. and Behrmann G. (2001). "UPPAAL – Now, Next, and Future", *Lecture Notes in Computer Science (LCNS) 2067/2001,Springer*: 99-124.

Asarin E., Caspi P. and Maler O. (2002). "Timed Regular Expressions", *Journal of the ACM 49*: 172-206.

Behrmann G. and Bengtsson J. (2002) "UPPAAL Implementation Secrets", *Lecture Notes in Computer Science (LCNS) 2469/2002,Springer*: 3-22.

Behrmann G. and David A. (2004). "A Tutorial on UPPAAL", *Lecture Notes in Computer Science (LCNS) 3185/2004,Springer*: 200-236.

Behrmann G.,David A, Larsen K. G., Hakansson J., Petterson P., Yi W. and Hendriks M. (2006). "UPPAAL 4.0.", *Proceedings of the 3rd International Conference on the Quantitative Evaluation of SysTems (QEST)*: 125-126.

Benerecetti M. and Cuomo N. (2009)."TPMC: A Model Checker For Time Sensitive Security Protocols", *Journal of Computers* 4: 366-377.

Bengtsson J. and Larsen K. (1996). "UPPAAL — A Tool Suite for Automatic Verification of Real-time Systems", *Lecture Notes in Computer Science (LCNS) 1066/1996, Springer*: 232-243.

Bengtsson, J. and Yi, W. (2004). "Timed automata: Semantics, Algorithms and Tools." *Lecture Notes in Computer Science (LCNS) 3098/2004, Springer*: 87-124.

Bengtsson J., David Griffioen W. O., Kristoffersen K. J., Larsen K. G., Larsson F., Pettersson P. and Yi W. (1996). "Verification of an Audio Protocol with Bus Collision using Uppaal", *Lecture Notes in Computer Science (LNCS) 1102/1996, Springer and Proceedings of the 8th International Conference on Computer Aided Verification (CAV'96)*: 244-256.

Bengtsson O., David Griffioen W.O., Kristoffersen K.J., Are K., Kristoffersen J., Larsen K. G., Larsson F., Pettersson P. and Yi W. (2002). "Automated Analysis of an Audio Control Protocol Using UPPAAL" *Journal of Logic and Algebraic Programming*: 52-53.

Bérard B., Petit A., Diekert V. and Gastin P. (1998). "Characterization of the Expressive Power of Silent Transitions in Timed Automata", *Fundamenta Informaticae 36, Citeseer*: 145-182.

Bordbar B. and Okano K. (2003). "Verification of Timeliness QoS Properties in Multimedia Systems." *Formal Methods and Software Engineering 2885, Springer*: 523-540.

Bosnacki, D. (1999). "Digitization of Timed Automata", *Proceedings of Formal Methods for Industrial Critical Systems (FMICS 99)*: 283-302.

Bouyer, P. (2004). "Forward Analysis of Updatable Timed Automata", *Formal Methods in System Design 24*: 281-320.

Bouyer, P. and Laroussine F. (2008). "Model Checking Timed Automata", *Modeling and Verification of Real-Time Systems, John Wiley & Sons, Ltd.*: 111-140.

Bouyer P. (2002). "Timed Automata May Cause Some Troubles", *Research Report LSV–02–9, LSV, ENS DE.*

Bouyer, P. (2003). "Untameable Timed Automata", *Lecture Notes in Computer Science (LNCS) 2607/2003, Springer and Proceedings of the 20th Annual Symposium on Theoretical Aspects of Computer Science*: 620-631.

Bouyer P., Dufourd C., Fleury E. and Petit A. (2000). "Are Timed Automata Updatable?", *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification, Springer*: 464-479.

Bouyer P., Dufourd C., Fleury E. and Petit A. (2004). "Updatable Timed Automata", *Theoretical Computer Science 321, Elsevier*: 291-345.

Bouyer P., Laroussinie F. and Reynier P. (2005). "Diagonal Constraints in Timed Automata: Forward Analysis of Timed Systems", *Lecture Notes in Computer Science (LCNS) 3829/2005, Springer*: 112-126.

Bouyer P., Haddad S. and Reynier P.A. (2009). "Undecidability Results for Timed Automata with Silent Transitions", *Fundamenta Informaticae 92*: 1-25.

Bowman H. and Gomez R. (2006). "Timelocks in Timed Automata", Concurrency Theory: Calculi an Automata for Modelling Untimed and Timed Concurrent Systems, *ISBN-13: 978-1852338954, Springer*: 347-376.

Bowman H., Faconti G., Katoen J.P., Latella D. and Massink M. (1998). "Automatic Verification of a Lip Synchronisation Algorithm using UPPAAL", *Proceedings of the 3rd International Workshop on Formal Methods for Industrial Critical Systems:* 97-124.

Bozga M., Daws C., Maler O., Olivero A., Tripakis S. and Yovine S. (1998). "Kronos: A Model-Checking Tool for Real-Time Systems", *Proceedings of the 10th International Conference on Computer Aided Verification (CAV '98), Springer*: 546-550.

Cerone, A. and Maggiolio-Schettini A. (1999). "Time-based Expressivity of Timed Petri Nets for System Specification", *Theoetical Computer Science 216, Elsevier*: 1-53.

Clark J., and Jacob J. (1997). A Survey of Authentication Protocol Literature Version 1.0.

Clarke E.M., Grumberg O. and Peled D.A. (1999). Model checking, *The MIT Press, ISBN-13: 978-0262032704*.

Corin R., Etalle S., Hartel P.H. and Mader A. (2007). "Timed Analysis of Security Protocols", *Journal of Computer Security 15*: 619-645.

Corin R., Etalle S., Hartel P.H. and Mader A. (2004). "Timed Model Checking of Security Protocols", *FMSE '04: Proceedings of the 2004 ACM Workshop on Formal Methods in Security Engineering*: 23-32.

Cremers, C. J. (2006). "Scyther – Semantics and Verification of Security Protocols", *Ph.D. Thesis, Technische Universiteit Eindhoven*.

D'Argenio P.R., Katoen J.P., Ruys T.C. and Tretmans G. J. (1997). "The Bounded Retransmission Protocol must be on time", *Lecture Notes in Computer Science (LCNS) 1217/1997, Springer*: 416-431.

D'Argenio P.R., Katoen J.P., Ruys T. and Tretmans J. (1996). "Modeling and Verifying a Bounded Retransmission Protocol", *University of Maribor*.

Dembinski P., Janowska A., Janowski P., Penczek W., Polrola A., Szreter M., Wozna B., Zbrzezny A. (2003). "Verics: A Tool for Verifying Timed Automata and Estelle Specifications", *9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*: 278-283.

Dolev, D. and Yao A.C. (1981). "On the Security of Public Key Protocols", *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science:* 350-357.

Emerson, E.A and Clarke, E.M. (1982). "Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons", *Science of Computer Programming 2*: 241-266.

Emerson E.A., Mok A.K., Sistla,A.P. and Srinivasan J. (1990). "Quantitative Temporal Reasoning", *Lecture Notes in Computer Science (LCNS) 531, Springer and Proceedings of the 2nd International Workshop on Computer Aided Verification (CAV'90)*: 136-145.

Finkel, O. (2006). "Undecidable Problems about Timed Automata", *Lecture Notes in Computer Science (LCNS) 4202/2006, Springer*: 187.

Furia C.A., Mandrioli D., Morzenti A. and Rossi M. (2010). "Modeling Time in Computing: A Taxonomy and a Comparative Survey", *ACM Computing Surveys (CSUR) 42*: 1-59.

Gupta V., Henzinger T.A. and Jagadeesan R. (1997). "Robust Timed Automata", *Lecture Notes in Computer Science (LCNS) 1201/1997, Springer*: 331-345.

Harel E. and Lichtenstein O. (1990). "Explicit Clock Temporal Logic", *Proceedings of the 5th Annual Symosium on Logic in Computer Science, IEEE Computer Society Press*: *402–413*.

Harrison M.D., Kray C., Sun Z., and Zhang H. (2007). "Factoring User Experience into the Design of Ambient and Mobile Systems", *Engineering Interactive Systems EIS 2007, Lecture Notes in Computer Science (LCNS) 4940/2008, Springer*: 243-259.

Havelund K., Larsen K. G. and Skou A. (1999). "Formal Verification of a Power Controller Using the Real-Time Model Checker", *Proceedings of the 5th International AMAST Workshop on Real-Time and Probabilistic Systems*: 277-298.

Havelund K., Skou A., Larsen K.G. and Lund K. (1997). "Formal Modeling and Analysis of an Audio/Video Protocol: An Industrial Case Study Using UPPAAL", *In Proceedings of the 18th IEEE Real-Time Systems Symposium*: 2-13.

Heidarian F., Schmaltz J. and Vaandrager F. (2009). "Analysis of a Clock Synchronization Protocol for Wireless Sensor Networks", *Lecture Notes in Computer Science (LCNS) 5850/2009, Springer and Proceedings of the 2nd World Congress on Formal Methods*: 516-531.

Henzinger T.A. and Jean-Francois R. (2000). "Robust Undecidability of Timed and Hybrid Systems", *Lecture Notes in Computer Science (LCNS) 1790/2000, Springer*: 145-159.

Henzinger T.A., Nicollin X., Sifakis J. and Yovine S. (1992). "Symbolic Model Checking for Real-time Systems", *Information and Computation 111*: 394-406.

Henzinger T.A., Manna Z. and Pnueli A. (1992). "What Good are Digital Clocks?", *Proceedings of the 19th International Colloquium on Automata, Languages and Programming*: 545-558.

Hessel A. and Pettersson P. (2006). "Model-Based Testing of a WAP Gateway: an Industrial Study", *Proceedings of the 11th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'06), and Lecture Notes in Computer Science (LCNS) 4346/2007, Springer*: 116-131.

Hoare, C.A.R. (1978). "Communicating Sequential Processes", *Communications of the ACM 21*: 666-677.

Huber B. and Schoeberl M. (2009). "Comparison of Implicit Path Enumeration and Model Checking Based WCET Analysis", *Proceedings of the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*.

Hwang T., Lee N. Y., Li C. M., Ko M.Y. and Chen Y.H. (1995). "Two Attacks on Neuman-Stubblebine Authentication Protocols", *Information Processing Letters 53*: 103-107.

Jakubowska G. and Penczek W. (2008). "Modelling and Checking Timed Authentication of Security Protocols", *Fundamenta Informaticae 79*: 363-378.

Jakubowska G., Penczek W. and Srebrny M. (2005). "Verifying Security Protocols with Timestamps via Translation to Timed Automata", *Proceedings of the International Workshop on Concurrency, Specification and Programming (CS&P'05), Warsaw University*: 100-115.

Jensen H.E., Larsen K. G., Ejersbo H., Kim J., Larsen G. and Skou A. (1996). "Modelling and Analysis of a Collision Avoidance Protocol using SPIN and UPPAAL", *DIMACS Series in Discrete Mathematics and Theoretical Computer Science 32*: 33-50.

Larsen K. G. and Pettersson P. (1997). "Uppaal – Status & Developments", *Lecture Notes in Computer Science (LCNS) 1254/1997, Springer, and Proceedings of the 9th International Conference on Computer Aided Verification*: 456-459.

Larsen K.G., Pettersson P. and Yi W. (1995). "Compositional and Symbolic Model-checking of Real-time Systems", *Proceedings of the 16th IEEE Real-Time Systems Symposium*: 76-87.

Lasota S. and Walukiewicz L. (2008). "Alternating Timed Automata", *ACM Transactions on Computational Logic (TOCL)*: 1-27.

Lindahl M., Pettersson P. and Yi W. (1998). "Formal Design and Analysis of a Gear Controller", *Proceedings of the 4th International Workshop on Tools and Algorithms for the Construction and Analysis of Systems , and Lecture Notes In Computer Science (LCNS) 1384/1998, Springer*: 281-297.

Lönn H. and Pettersson P. (1997). "Formal Verification of a TDMA Protocol Start-Up Mechanism." *In Pacific Rim International Symposium on Fault-Tolerant Systems, IEEE Computer Society*: 235-242.

Mukhopadhyay S. and Podelski A. (1999). "Beyond Region Graphs: Symbolic Forward Analysis of Timed Automata", *Lecture notes in Computer Science (LCNS) 1738/1999, Springer*: 232-244.

Neuman B.C. and Stubblebine S.G. (1993)."A note on the Use of Timestamps as Nonces", *SIGOPS Operating Systems Review 27, ACM*: 10-14.

Ostroff J.S. (1989). Temporal Logic for Real Time Systems, *Wiley Advanced Software Development Series, John Wiley & Sons, ISBN: 0-471-92402-4.*

Ouaknine J. and Worrell J. (2004). "On the Language Inclusion Problem for Timed Automata: Closing a Decidability Gap", *Proceedings of IEEE Symposium on Logic in Computer Science (LICS)*: 54-63.

Ouaknine J. and Worrell J. (2003). "Revisiting Digitization, Robustness, and Decidability for Timed Automata", *Proceedings of IEEE Symposium on Logic in Computer Science (LICS)*:198-207.

Ouaknine J. and Worrell J. (2003). "Universality and Language Inclusion for Open and Closed Timed Automata", *Lecture Notes in Computer Science (LCNS) 2623/2003, Springer*: 375-388.

Pelánek R. (2009). "Fighting State Space Explosion: Review and Evaluation", *Lecture Notes in Computer Science (LCNS) 5596/2009, Springer*: 37-52.

Penczek W. and Polrola A. (2006). Advances in Verification of Time Petri Nets and Timed Automata: A Temporal Logic Approach, *Springer-Verlag, ISBN: 978-3540328698*.

Pnueli A. (1977). "The Temporal Logic of Programs", *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*: 46-77.

Reed G.M. and Roscoe A.W. (1988). "A Timed Model for Communicating Sequential Processes", *Theoretical Computer Science 58, Elsevier*: 314-323.

Schonwalder K., Kehne A., Schonwalder J., Langendorfer H. and Braunschweig T. (1992). "A Nonce-Based Protocol for Multiple Authentications", *ACM SIGOPS Operating Systems Review 26*: 84 - 89.

Srba J. (2008). "Comparing the Expressiveness of Timed Automata and Timed Extensions of Petri Nets", *Proceedings of the 6th international Conference on Formal Modeling and Analysis of Timed Systems, and Lecture Notes In Computer Science (LCNS) 5215, Springer*: 15-32.

Suman P.V. and Pandya P. K. (2009). "Determinization and Expressiveness of Integer Reset Timed Automata with Silent Transitions", *Proceedings of the 3rd International Conference on Language and Automata Theory and Applications:* 728-739.

Suman P.V., Pandya P.K., Krishna S.N.and Manasa L. (2008). "Timed Automata with Integer Resets: Language Inclusion and Expressiveness", *Proceedings of the 6th international conference on Formal Modeling and Analysis of Timed Systems*: 78-92.

Tripakis, S. (2006). "Folk theorems on the Determinization and Minimization of Timed Automata", *Information Processing Letters 99, Elsevier*: 222-226.

Vardi M. and Wolper P. (1986). "An Automata-theoretic Approach to Automatic Program Verification", *Proceedings of the First IEEE Symposium on Logic in Computer Science*: 332-344.

Wang A.D. and Yi W. (2000). "Modelling and Analysis of a Commercial Field Bus Protocol", *Proceedings of 12th Euromicro Conference on Real-Time Systems,* IEEE: 165-172.

Woo T.Y.C. and Lam S.S. (1994). "Design, Verification and Implementation of an Authentication Protocol", *Proceedings of International Conference on Network Protocols*.

Yovine S. (1997). "KRONOS: A Verification Tool for Real-time Systems", *International Journal on Software Tools for Technology Transfer*, *Springer*: 123-133.