Study on Model Abstraction for Model Checking of Real-time Systems

January 2011

Takeshi NAGAOKA

Study on Model Abstraction for Model Checking of Real-time Systems

Submitted to
Graduate School of Information Science and Technology
Osaka University

January 2011

Takeshi NAGAOKA

Abstract

Model checking techniques are considered as promising techniques for information system verification due to their ability of exhaustive checking. In addition, diagnostic information of model checking, which is provided as a counter example, is thought to be useful for error correction. In recent years, model checking supports more complicated analysis such as verification of real-time requirements, performance evaluation for systems with random behavior, and so on. It is difficult, however, to apply such analysis into large systems because of the well-known state explosion problem.

Timed automata, which can model real-time behavior of systems, are used to verify real-time requirements. The timed automata are finite automata with real-time constraints described by real-valued variables called clocks. Therefore the timed automata theoretically have infinite state space. Almost all verification techniques for the timed automata use the fact that the infinite state space can be approximated to finite one. However, the size of the approximated state space increases exponentially with the number of clock variables. For verification of systems with random behavior, probabilistic model checking techniques are developed, where probabilistic models such as Markov chains, Markov decision processes are used. In these days, probabilistic timed automata, which can model real-time and probabilistic behavior of systems, are proposed.

To reduce the state space of timed automata, we propose an abstraction technique for timed automata. The proposed technique is based on a well-organized abstraction framework, CounterExample-Guided Abstraction Refinement (CEGAR), in which we use a counter example as a guide to refine an abstracted model. In this technique, we generate an abstract model by removing all of the clock variables from given timed automata. Then, we refine the abstract model repeatedly to generate an appropriate abstract model for checking a given property. Although, in general, the refinement operation is applied to an abstract model directly, the proposed technique modifies the original timed automata, and next generates refined abstract models from the modified automata. Experimental results show the abstraction algorithm can reduce the total memory consumption by at most 80 per-

cent compared to applying model checking without abstraction.

We extend the abstraction technique into abstraction for probabilistic timed automata. In the technique, we remove clock variables from given probabilistic timed automata as well as the original one. Then, we apply probabilistic model checking to the generated abstract model which is just a Markov decision process (MDP) with no time attributes. In general, probabilistic model checking does not produce concrete paths as a counter example which are required for abstraction refinement. Therefore, we also perform k-shortest paths search to obtain the concrete paths. Experimental results show that the proposed technique reduces state space of probabilistic timed automata compared to an existing approach. Also, the contribution of this study includes generation of concrete paths as a counter example while other related works cannot generate them.

In this study, we propose a QoS analysis technique of real-time distributed systems based on hybrid analysis of probabilistic model checking and simulation. For the Internet, system developers often have to estimate the QoS by simulation techniques or mathematical analysis. Probabilistic model checking can evaluate performance, dependability and stability of information processing systems with random behavior. We apply a hybrid analysis approach onto real-time distributed systems. In the hybrid analysis approach, we perform stepwise analysis using probabilistic models of target systems in different abstract levels. First, we create a probabilistic model with detailed behavior of the system (called detailed model), and apply simulation on the detailed model. Next, based on the simulation results, we create a probabilistic model in an abstract level (called simplified model). Then, we verify qualitative properties using the probabilistic model checking techniques. This prevents from state-explosion. We evaluate the validity of our approach by comparing to simulation results of NS-2 using a case study of a video data streaming system. The experiments show that the result of the proposed approach is very close to that of NS-2 simulation. The result encourages the approach is useful for the performance analysis on various domain.

Finally, we propose a stepwise method to verify consistency of timeliness QoS of component-based designed real-time systems. In the proposed method, the system components are designed using UML diagrams and are provided with the timeliness QoS annotated with OCL. The basis of this technique is to formally ensure that the required timeliness QoS is satisfied under the provided timeliness QoS, given the network property and the UML diagrams. In order to avoid the state-explosion problem during model checking, we separate the model checking problem into two steps. The first step checks the satisfiability using an abstract model of each of the components derived automatically from the provided QoS. The second step independently performs model checking for each of the components using a more detailed version of the behavioral model of a given component.

Such an approach reduces the number of total states to check. Furthermore, the approach can be extended into hierarchical design, which leads to good scalability.

List of Publications

Major Publications

[1-1] Takeshi Nagaoka, Eigo Nagai, Kozo Okano, and Shinji Kusumoto: "Stepwise Approach to Design of Real-Time Systems based UML/OCL with Formal Verification," International Journal of Informatics Society (IJIS), Informatics Society, Vol.1, No.2, pp37-44, Sep.2009.

[1-2] Takeshi Nagaoka, Akihiko Ito, Kozo Okano, and Shinji Kusumoto: "Qos Evaluation for Real-Time Distributed Systems Using the Probabilistic Model Checker Prism," In Proceedings of International Workshop on INformatics, IWIN 2009, pp.60-66, Sep.2009.

[1-3] Takeshi Nagaoka, Akihiko Ito, Kozo Okano, and Shinji Kusumoto: "Qualitative Analysis of Real-time Distributed Systems Considering Network Congestion by Probabilistic Model Checker PRISM," In Proceedings of International Workshop on Empirical Software Engineering in Practice 2009, IWESEP2009, Oct.2009.

[1-4] Takeshi Nagaoka, Kozo Okano, and Shinji Kusumoto: "An Abstraction Refinement Technique for Timed Automata Based on Counterexample-Guided Abstraction Refinement Loop," IEICE Transactions on Information and Systems, IEICE, Vol.E93-D, No.5, pp.994-1005, 2010.

[1-5] Takeshi Nagaoka, Akihiko Ito, Toshiaki Tanaka, Kozo Okano, and Shinji Kusumoto: "Reachability Analysis of Probabilistic Real-Time Systems Based on CEGAR for Timed Automata," In Proceedings of International Workshop on INformatics, IWIN 2010, pp.18-26, Sep.2010.

[1-6] Takeshi Nagaoka, Akihiko Ito, Toshiaki Tanaka, Kozo Okano, Shinji Kusumoto: "Reachability Analysis of Probabilistic Timed Automata Based on an Abstraction Refinement Technique," In Proceedings of International Workshop

on Empirical Software Engineering in Practice 2010, IWESEP2010, pp.33-38, Dec.2010.

[1-7] Takeshi Nagaoka, Akihiko Ito, Kozo Okano, and Shinji Kusumoto: “QoS Analysis of Real-time Distributed Systems Based on Hybrid Analysis of Probabilistic Model Checking Technique and Simulation,” IEICE Transactions on Information and Systems, accepted.

Acknowledgements

This work could be achieved owing to a great deal of help of many individuals.

First, I would like to thank my supervisor Professor Shinji Kusumoto of Osaka University, for his continuous support, encouragement and guidance of the work.

I would like to express my gratitude to Professor Toshimitsu Masuzawa, and Professor Katsuro Inoue for their valuable comments and helpful suggestions and questions on this thesis.

I would like to express my sincere gratitude to Associate Professor Kozo Okano of Osaka University for his adequate guidance, valuable suggestions and discussions throughout this work.

I'm grateful to Assistant Professor Yoshiki Higo of Osaka University for his helpful comments and suggestions.

Many of courses that I have taken during my graduate career have been helpful in preparing this thesis. I would especially like to acknowledge the guidance of Professor Ken-ichi Hagihara and Professor Yasushi Yagi.

Finally, I would like to thank the all members of Kusumoto Laboratory of Osaka University for their helpful advice.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Formal Verification of Information Systems

In recent years, though information systems play an important role in social activities, defects on the systems may cause serious loss to society. As the systems become larger and more complicated, it is difficult to assure the reliability of the systems. In order to develop reliable systems, it is important to verify designs of the systems at an early phase of their development. Such verification avoids us to regress the development steps.

Formal methods are considered as promising techniques for reliable systems development. The formal methods can automatically prove the correctness of the systems using formal descriptions of the system designs and mathematical theories in order to analyse the systems. Using the formal methods, we can also obtain reliable proofs based on the mathematical theories.

In general, the formal methods are classified into two kinds of approaches, model checking[1] and automated theorem proving[2]. In the model checking technique, information systems are modeled as finite state machines. Model checkers[3, 4, 5, 6] decide whether the model satisfies given requirements or not by searching state space exhaustively. On the other hand, in the automated theorem proving technique, systems and requirements are described in logical forms. Theorem provers[7, 8, 9] prove whether the system satisfies the requirements in mathematical ways. Usually, theorem provers only check if proofs, which are human constructed manually, are correct or not. Conventional theorem provers are not full automatic. The proofs are constructed interactively. A simple theory, such as inequalities on integer arithmetic, can be automatically proved. For example, $x > y \ implies \ x + 2y > 3y$, can be proved by most of automated theorem provers. However, some of them cannot automatically prove a tautology $x = y \ and \ \forall x.f(x) = g(x) \ imply \ f(x) = g(y)$. Thus, each theorem prover

only supports its own limited class of theories. Though some of modern theorem provers also support for engineers to construct their proofs and have facilities to enhance several non-standard logics, and domain specific axioms and inference rules, the essential of the theorem provers is not to provide proofs but to check proofs. In general, theorem proving approaches can deal with a wide range of information systems according to their basis theories and mathematic abilities of the users. In contrast, model checking approaches are easy to use but the target domain is limited. They are suitable to check behaviour of the systems rather than check correctness of calculation. Some methods are hybrid. For example, B Method[10] supports both of a theorem prover and a model checker in order to resolve "proof obligations" which are automatically derived from specifications described in B Method syntax. Although formal verification has merits with automated and reliable verification, there are several problems such as difficulty of formal descriptions, limitation of its scalability. In this study, we propose techniques to improve the limitation of scalability of model checking using model abstraction techniques.

1.2 Model Checking

A model checker checks if a given system modeled in a finite automaton satisfies given specifications by searching the finite transition system exhaustively. It sometimes has, however, limitation in scalability. In order to improve the scalability, a model abstraction technique is important[11, 12, 13, 14, 15].

For verification of real-time systems such as embedded systems, timed automata[16, 17, 18] are usually used. On the other hand, probabilistic model checking can evaluate performance, dependability and stability of information processing systems with random behaviors[19]. Probabilistic model checking handles several probabilistic models such as discrete-time Markov chains (DTMC), continuous-time Markov chains (CTMC), Markov decision processes (MDP), and so on. In recent years, probabilistic models with real-time behaviors, called probabilistic timed automata[20] are used to evaluate dependability of real-time systems based on the probabilistic model checking technique.

1.2.1 Models

This research mainly focuses on timed automata and probabilistic models including probabilistic timed automata as models for model checking.

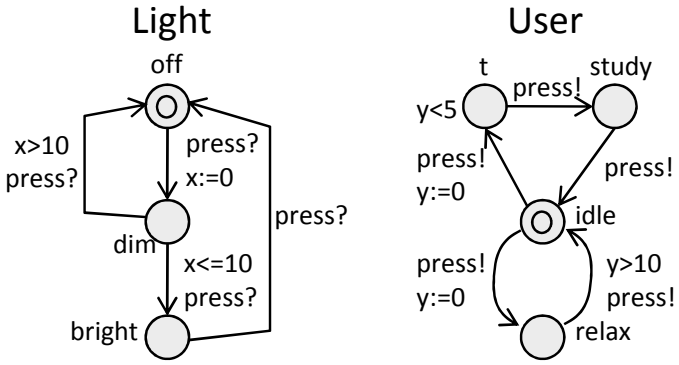


Figure 1.1: The Light User Model

Timed Automata

Timed automata are finite state automata with real-time constraints described with real-valued variables called clocks. In timed automata, real-valued clock constraints are assigned to their control state (called a location) and transitions. Therefore, they have infinite state space represented in a product of discrete state space made by locations and continuous state space made by clock variables. In traditional model checking for timed automata, using the property that we can treat the state space of clock variables as a finite set of regions, we can perform model checking on timed automata models. However, the size of such regions increases exponentially with the number of clock variables, Thus an abstraction technique is also needed. For verification of timed automata, several tools such as UPPAAL[5, 21, 22], KRONOS[23, 24] are developed.

Figure 1.1 is an example of timed automata, the Light User model[16]. The model is composed of two timed automata, a light model (left side of the figure) and a user model (right side of the model). Each model has one clock variable and communicates with the other model through the channel $press$.

Probabilistic Models

Probabilistic models[25] can describe behavior of stochastic systems such as network protocols which decide next behavior stochastically, randomised distributed algorithms, and so on. In the probabilistic models, transitions between states are labeled by the probability to fire the transitions.

In general, the probabilistic models are classified into discrete-time Markov chains (DTMCs), continuous-time Markov chains (CTMCs), and Markov decision

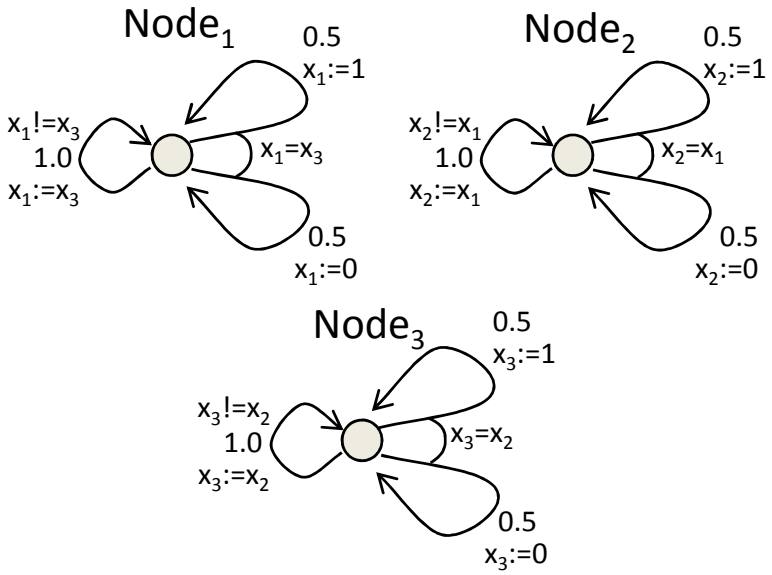


Figure 1.2: The Randomised Self-Stabilizing Algorithms Model

processes (MDPs). In the DTMCs, the label of probability is assigned discretely for each transition. In the CTMCs, transitions between states occur in a real time step while transitions of DTMCs occur in a discrete time step. MDPs can be seen as a generalisation of DTMCs. MDPs can describe both non-deterministic and probabilistic behavior.

Figure 1.2 shows an example of the DTMCs, the Randomised Self-Stabilizing Algorithms[26] model. In the model, each node has one integer variable x_i. The stable configurations are those where exactly one node has a token. In this model, the $node_i$ has a token if $x_i = x_{i-1}$ holds. Each node changes the value of x_i randomly, and this finally makes the configuration stable.

One of probabilistic model checkers PRISM[6, 27] can handle DTMCs, CTMCs, and MDPs.

Probabilistic Timed Automata

Probabilistic timed automata are kinds of timed automata extended with probabilistic behavior. Therefore, state space of probabilistic timed automata is same as that of timed automata. Since the probabilistic timed automata accept non-deterministic transitions, they are also considered as kinds of Markov decision processes. Using the probabilistic timed automata, several case studies were intro-

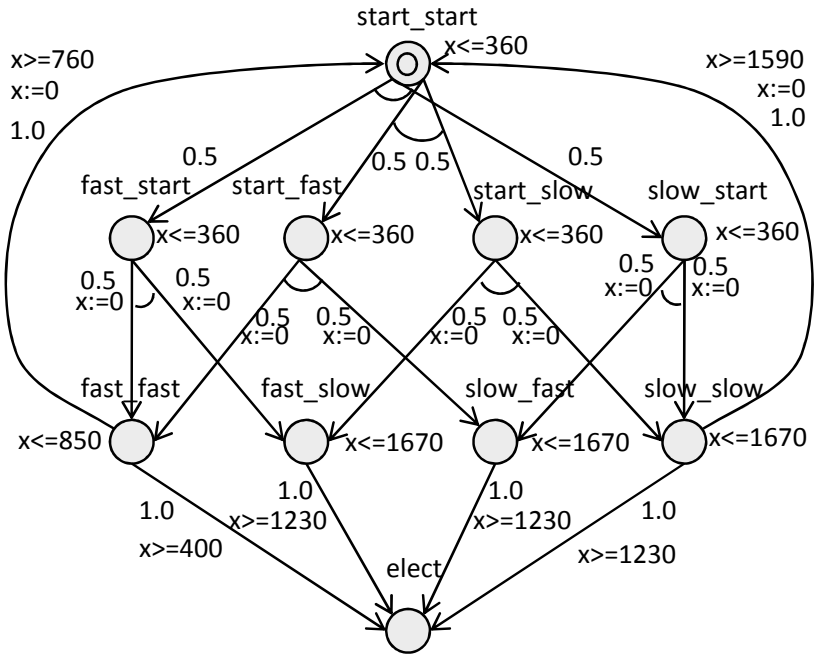


Figure 1.3: The FireWire Root Contention Protocol Model

duced. For example, Papers[28, 29] performed performance analysis of protocols for wireless networks.

Figure 1.3 shows an example of a probabilistic timed automaton, the FireWire root contention protocol model[29]. In the figure, transitions which belong to the same distribution are connected with a small arc at their source points. For example, at the location *start_start*, there are two probability distributions. In one of them, the control moves to the location *fast_start* with the probability 0.5 and to the location *slow_start* with 0.5 as well. In the other one, the control moves to the location *start_fast* with the probability 0.5 and to the location *start_slow* with 0.5 as well. These distributions are selected non-deterministically.

1.2.2 Properties to be Checked

If we model behavior of systems, we should formally describe properties to be checked. In formal verification, the properties are described with temporal logic[1] such as liner temporal logic (LTL)[30] and computational tree logic (CTL)[31]. The temporal logic can describe properties quantified in terms of time. In formulae

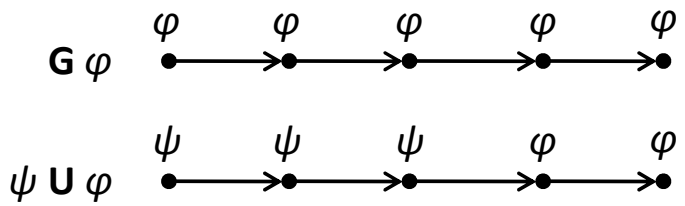


Figure 1.4: Examples of Semantics of LTL

described with temporal logic, we can use temporal operators such as **F** which means the property will hold in the future, **A** which means that the property always holds, and so on.

LTL

LTL can represent properties related to a path on a target model. For example, properties such as 'for some state on the path' or 'for every two consecutive states' can be expressed. LTL accepts temporal operators **F** (in the future), **G** (globally), **X** (next) and **U** (until) as well as general logical operators such as conjunction ($\wedge$), disjunction ($\vee$), implication ($\rightarrow$), and so on. For example, the LTL formula '**G** ϕ' means that the property ϕ holds at all states along the path, and 'ψ **U** ϕ' means that there is a state on the path at which ϕ holds, and ψ holds at every state before ϕ. Figure 1.4 shows examples of semantics of LTL. The dots in the figure show states. The states where some LTL formulae such as ϕ and ψ are associated show that the associated formulae hold at the states.

The model checker SPIN[3] supports LTL model checking. In the LTL model checking, the property described with LTL is translated into a Büchi automaton[32]. SPIN computes the synchronous product of this Büchi automaton and the automaton describing target system's behavior, and checks whether there exists the language accepted by the product automaton.

CTL

CTL can represent properties with tree like structures which have several branches. CTL accepts quantifiers over paths such as **A**, which means that the property holds for all paths starting from the current state, and **E**, which means that the there exists at least one path starting from the current state where the property holds, as well as temporal operators **F**, **G**, **X** and **U**. In the CTL formula, these quantifiers and temporal operators are combined. For example, the CTL property '**AG** ϕ'

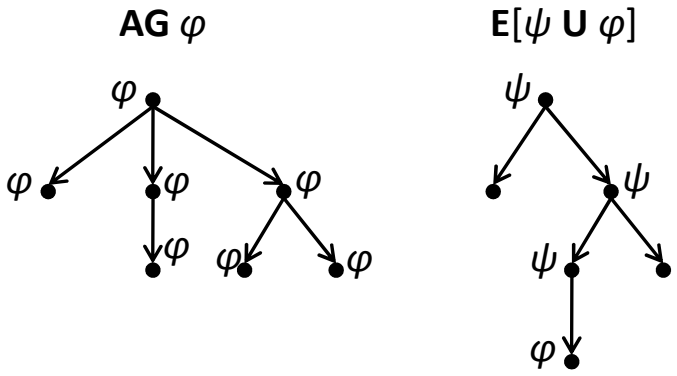


Figure 1.5: Examples of Semantics of CTL

means that for all paths starting from the current state the property ϕ holds at all states along the paths. Figure 1.5 shows examples of semantics of CTL.

The model checker NuSMV[4] supports CTL model checking. In the CTL model checking, the model checker inductively computes a set of states which satisfy the given CTL formula. If the set of states obtained by CTL model checking includes initial states, the model checker concludes that the model satisfies the given CTL property. If the model is finite, such computation converges within finite time. Therefore, the CTL model checking terminates.

There are several extensions of CTL to represent real-time properties, probabilistic properties, and so on.

Timed CTL (TCTL)[33, 34] extends CTL with clock variables. Therefore, TCTL accepts atomic formulae about clock variables, which are restricted by differential inequalities between two clock variables in general. TCTL is used to describe properties on timed automata and probabilistic timed automata.

Probabilistic CTL (PCTL)[35] extends CTL with probability. In the PCTL properties, the operator **P**, which indicates probability over an associated computation tree, is provided. For example, PCTL can describe the property of 'what is the probability to reach a given error state.' PCTL is used to describe properties on discrete-time Markov chains, Markov decision processes, and probabilistic timed automata. On the other hand, for continuous-timed Markov chains, continuous stochastic logic (CSL)[36], which is inspired by the logic CTL, PCTL, and TCTL, is mainly used.

Probabilistic timed CTL (PTCTL)[20] is proposed to combine the probabilistic operator of the PCTL and timing constraints of the TCTL. PTCTL is used to describe properties of probabilistic timed automata.

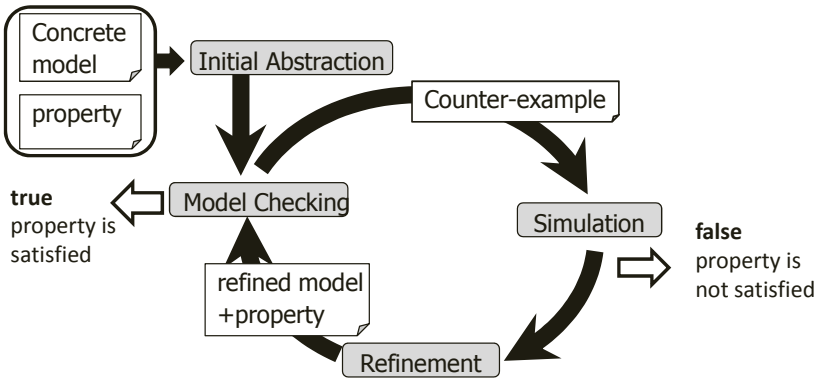


Figure 1.6: A General CEGAR Technique

1.3 Approaches to Avoid State Explosion

In model checking, explosion of state space or execution time is a major problem. In general, such problem is caused by increasing of size of target systems or complexity of properties to be checked. In recent years, there are several studies which tackle this problem.

1.3.1 Counterexample Guided Abstraction Refinement

Clarke *et al.* proposed an abstraction technique called CEGAR (CounterExample-Guided Abstraction Refinement)[11] shown in Fig.1.6. The technique is used for abstraction of finite models[11, 12], hybrid systems[13], timed automata[37, 38, 39], and other models. In the CEGAR technique, we use a counter example produced by a model checker as a guide to refine excessively abstracted models.

A general CEGAR technique consists of several steps. First, we abstract an original model and generate an initial abstract model. Next we perform model checking on the abstract model. In this step, if a counter example is detected by a model checker, we check the counter example on the original model. If we find that the counter example is spurious, we have to refine the abstract model. The last step is repeated until the valid output is obtained. In the CEGAR loop, an abstract model must satisfy the following property; if the abstract model satisfies a given specification, the concrete model also satisfies it.

1.3.2 Predicate Abstraction

Predicate abstraction[14, 15] abstracts state space using a given finite set of predicates. For the set of predicates, predicate abstraction defines an equivalence relation over a set of states with regard to the truth of every predicate. In the predicate abstraction, how to choose such a set of predicates is one of major problems. Some works use the framework of CEGAR, i.e. if abstraction generates spurious behavior, they refine an abstract model by adding other predicates into the set.

1.3.3 Partial Order Reduction

Partial order reduction[40, 41] mainly reduces state space search for a model described as a set of concurrent processes. An execution of such a concurrent model is represented as a sequence of events where events of concurrent processes are interleaved. Concrete orders of such events, however, have little effect on the truth of properties to be checked in general, if the events are independent each other. Partial order reduction defines partial orders over such interleaves and reduces state space search which is seems to be equivalent.

1.3.4 Symmetry Reduction

Symmetry Reduction[42, 43, 44] exploits symmetry of a concurrent process model. Then it reduce state space of the model into that of its quotient model, where symmetric states are identified. For model checking of timed automata[43] and probabilistic models[44], the symmetry reduction is adopted.

1.4 Research Overview

In this study, we propose techniques to avoid state explosion for model checking of real-time systems. First, we propose an abstraction refinement technique for timed automata using the CEGAR framework. Next, we extend the technique into abstraction for probabilistic timed automata. On the other hand, for model checking of distributed systems, we propose a hybrid approach of probabilistic model checking and simulation techniques.

1.4.1 Abstraction Refinement for Timed Automata based on CEGAR

Chapter 2 describes the abstraction refinement technique for timed automata using the CEGAR framework. The first step of the technique is abstraction, in which we delete all of time attributes from the given timed automaton. The obtained automaton is just a finite automaton preserving the transition relations of the

original timed automaton; therefore the obtained finite automaton is, in general, an over-approximation of the original one. We restrict the class of the verification properties into reachability; thus if an abstract model satisfies a given property then the concrete model also satisfies the property. The reachability analysis is the primitive procedure for safety checking. This means that model checking problems on several important properties could be reduced into the reachability analysis problem.

In general, CEGAR techniques[11, 12, 13, 37, 38, 39] directly transforms an abstract model using counter examples in the refinement step. Our proposed method, however, doesn't directly transform the abstract model. It first transforms the original model using counter examples and then it creates a new abstract model from it by removing clock attributes; thus our algorithm indirectly refines the abstract model. The algorithm transforms the original timed automaton by adding extra transitions and removing some transitions but it preserves the behavioral equivalence of the timed automaton and prevents the spurious counter examples. More concretely, it duplicates locations and transitions so that its abstract model can tell behavioral difference caused by clock values which affects the counter examples. Consequently the obtained new abstract model does not accept the spurious counter example.

Related works [37, 38, 39] have proposed CEGAR based abstraction techniques for timed automata. Although these techniques mainly refine the abstract models by adding clock variables which have been removed by initial abstraction, our refinement method modifies the original timed automata and produces the refined abstract model from the modified models, instead of adding clock variables.

1.4.2 Abstraction Refinement for Probabilistic Timed Automata based on CEGAR

Chapter 3 describes the extension of our CEGAR technique into abstraction for probabilistic timed automata. In this study, we propose a reachability analysis technique for probabilistic timed automata. The abstraction technique abstracts time attributes of probabilistic timed automata by applying our abstraction technique for timed automata. Then, we apply probabilistic model checking to the generated abstract model which is just a Markov decision process (MDP) with no time attributes. The probabilistic model checking algorithm calculates a summation of occurrence probability of all paths which reach to a target state for reachability analysis. For probabilistic timed automata, however, we have to consider required clock constraints for such paths, and choose the paths whose required constraints are compatible. Since our abstract model does not consider the clock constraints, we add a new flow where we check whether all paths used for probability calcu-

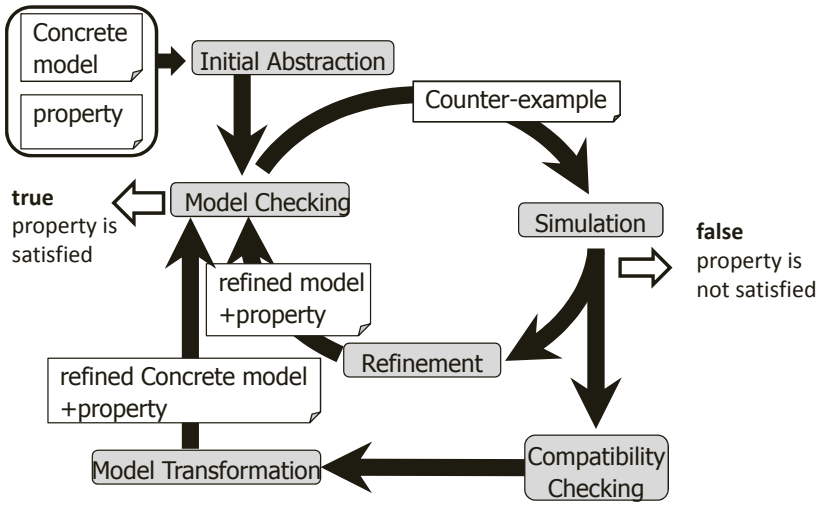


Figure 1.7: Our CEGAR Technique for Reachability Analysis of a Probabilistic Timed Automaton

lation are compatible. Also, if they are not compatible, we transform the model so that we do not accept such incompatible paths simultaneously. The proposed procedure for the probabilistic timed automata is shown in Fig.1.7.

Several papers including Paper[45] have proposed probabilistic model checking algorithms. These algorithms, however, don't provide counter examples when properties are not satisfied. Our proposed method provides a counter example as a set of paths based on k-shortest paths search. This is a major contribution of our method. The proposed method also performs model checking considering compatibility problem. Few approaches resolve the compatibility problem. Our approach also shows the efficiency via performing experiments.

1.4.3 Qualitative Analysis of Real-time Distributed Systems Using the Probabilistic Model Checker PRISM

Chapter 4 describes a qualitative analysis technique for Real-time Distributed Systems.

Probabilistic model checking can evaluate performance, dependability and stability of information processing systems with random behaviors[19]. PRISM[6] is one of the probabilistic model checkers. It handles automata with probabilities (discrete and continuous time Markov chains) and time elapse. Therefore, it is suitable for modeling the network systems. One of the approaches, which overcomes

drawbacks of simulation approach and model checking approach, seems to be a kind of a hybrid approach.

In order to find if the hybrid approach is applicable to real systems, we apply a hybrid analysis technique onto real-time distributed systems, which uses both of simulation and model checking techniques. In our approach, we perform a stepwise analysis using probabilistic models of target systems in different abstract levels. First, we create a probabilistic model with detailed behavior of the system (called detailed model), and apply simulation on the detailed model. Next, based on the simulation results, we create a probabilistic model in an abstract level (called simplified model). Then, we verify qualitative properties using the probabilistic model checking techniques.

As related works, several case studies are performed using PRISM[45, 28, 20]. For example, Paper[28] deals with a network protocol. Few works, however, concern the QoS analysis of the whole system. Papers[46] and [47] propose abstraction methods for probabilistic systems based on an abstraction refinement approach. Papers[48] and [49] propose verification approach based on the simulation technique. Paper[46] extracts a number of representative sample paths on a probabilistic model and decides if the model satisfies a given property using such paths. In Paper[49], they model a biomedical sensor network as timed automata, and use the simulation technique to adjust some parameters.

The contribution of this research includes that we present a technique to guarantee the QoS of real-time distributed systems. Our experimental results show the correctness of our detailed model at least for all of nine scenarios. Also, we can apply probabilistic model checking on the simplified model within realistic time without state explosion. It shows that the proposed method is useful to analyze the network performance. We believe that such analysis is useful for other kind of network analysis.

1.4.4 Formal Verification with a Stepwise Abstraction Approach for UML/OCL Based Design of Real-time Systems

Chapter 5 describes a technique to verify consistency of timeliness QoS of component-based design for real-time systems.

Nowadays, most real-time systems are designed with help of UML diagrams[50]. Especially components and their relation through signal communication can be represented in a class diagram of UML. In UML based design, such timeliness QoS can be annotated in OCL[51]. The annotation is associated to each of components as a provided QoS and also to a network link as a network property.

This study proposes a new method to verify consistency of timeliness QoS of component-based real-time systems. We assume that timeliness QoS is not only

given to a whole system (Required QoS) but also associated with each component of a given system (Provided QoS). Timeliness QoS is a time aspect of QoS (Quality of Service) features[52]. In this study, we treat jitter, latency and throughput as timeliness QoS.

The proposed method is revised version of paper [53]. The method in [53] uses Linear Programming (LP) for some of verification. The approach has a disadvantage that connection among components has to be acyclic, and it cannot be applied to hierarchical design. The proposed method uses abstract QoS automata instead of using LP; thus it improves the former disadvantage. The heart of the technique is formally to ensure that the required timeliness QoS is satisfied under the provided timeliness QoS, given network property and the class diagram.

In order to check the satisfiability, there are several approaches. Model checking is one of such approaches. Notion of Test Automata[54, 55] and its application is also useful. However, one of disadvantage of the method is the state-explosion problem. In order to avoid state-explosion while performing model checking, we separate the problem into two steps. The first step checks the satisfiability using abstract model of each of components derived automatically from the provided QoS. The second step performs model checking each of components independently using more detailed version of behavioral model of a component. Such an approach efficiently reduces the number of total states to check. Moreover the approach can be extended into hierarchical design; therefore it has good scalability.

Chapter 2

Abstraction Refinement for Timed Automata based on CEGAR

2.1 Introduction

In verification of real time systems, timed automata have widely been used[16, 17, 18], which can describe behavior of real-time systems. In timed automata, real-valued clock constraints are assigned to its control state (called a location). Therefore, it has an infinite state space represented in a product of discrete state space made by locations and continuous state space made by clock variables. In traditional model checking for a timed automaton, using the property that we can treat the state space of clock variables as a finite set of regions; we can perform model checking on timed automata models. However, the size of such regions increases exponentially with the number of clock variables; thus an abstraction technique is also needed. One of the approaches to avoid explosion of the state space is a model abstraction approach. Especially, the CEGAR (CounterExample-Guided Abstraction Refinement)[11] technique proposed by Clarke *et al.* is considered as the promising technique for model abstraction.

This chapter describes a CEGAR technique for timed automata. This technique removes all of the clock variables from the timed automata, which means that the obtained abstract model is just a finite automaton with no clock constraints. Therefore, applying model checking into the abstract model may generate spurious counter examples which do not occur on the original timed automata. If such spurious counter example is detected, we transform the transition relation on the original timed automata so that the model behaves correctly even if we don't consider the

clock constraints. Such transformation obviously represents the difference of behavior caused by the clock attributes. Therefore, the finite number of application of the refinement algorithm enables us to check the given property without the clock attributes.

In this chapter, we show concrete algorithm of abstraction refinement for timed automata, and prove its correctness. Also, experimental results indicate our abstraction can reduce state space of timed automata.

2.2 Preliminary

2.2.1 Clock and Zone

Let C be a finite set of clock variables which take non-negative real values ($\mathbb{R}_{\geq 0}$). A map $\nu : C \to \mathbb{R}_{\geq 0}$ is called a clock assignment. The set of all clock assignments is denoted by $\mathbb{R}^C_{\geq 0}$. For any $\nu \in \mathbb{R}^C_{\geq 0}$ and $d \in \mathbb{R}_{\geq 0}$ we use $(\nu + d)$ to denote the clock assignment defined as $(\nu + d)(x) = \nu(x) + d$ for all $x \in C$. Also, we use $r(\nu)$ to denote the clock assignment obtained from ν by resetting all of the clocks in $r \subseteq C$ to zero.

Definition 2.2.1 (Differential Inequalities on C)**.** *Syntax and semantics of a differential inequality E on a finite set C of clocks is given as follows:*
$E ::= x - y \sim a \mid x \sim a$,
where $x, y \in C$, a is a literal of a real number constant, and $\sim \in \{\leq, \geq, <, >\}$. Semantics of a differential inequality is the same as the ordinal inequality.

Definition 2.2.2 (Clock Constraints on C)**.** *A set of clock constraints $c(C)$ on a finite set C of clocks is defined as follows:*
A differential inequality in_1 on C is an element of $c(C)$.
Let in_1 and in_2 be elements of $c(C)$, $in_1 \wedge in_2$ is also an element of $c(C)$.

A zone $D \in c(C)$ is described as a product of finite differential inequalities on clock set C, which represents a set of clock assignments that satisfy all the inequalities.

2.2.2 Timed Automaton

Definition 2.2.3 (Timed Automaton)**.** *A timed automaton $\mathscr{A}$ is a 6-tuple (A, L, l_0, C, I, T), where*
A: a finite set of actions;
L: a finite set of locations;
$l_0 \in L$: an initial location;

C: a finite set of clocks;
$I \subset (L \to c(C))$: a mapping from locations to clock constraints, called a location invariant; and
$T \subset L \times A \times c(C) \times \mathscr{R} \times L$, where $c(C)$ is a clock constraint, called guards;
$\mathscr{R} = 2^C$: a set of clocks to reset.

A transition $t = (l_1, a, g, r, l_2) \in T$ is denoted by $l_1 \xrightarrow{a,g,r} l_2$. A map $\nu : C \to \mathbb{R}_{\geq 0}$ is called a clock assignment. We define $(\nu + d)(x) = \nu(x) + d$ for $d \in \mathbb{R}_{\geq 0}$. $r(\nu) = \nu[x \mapsto 0], x \in r$, where $\nu[x \mapsto 0]$ means the valuation that maps x into zero, is also defined for $r \in 2^C$.

Definition 2.2.4 (Semantics of a Timed Automaton)**.** *Given a timed automaton $\mathscr{A} = (A, L, l_0, C, I, T)$, let $S \subseteq L \times \mathbb{R}^C_{\geq 0}$ be a set of whole states of $\mathscr{A}$. The initial state of $\mathscr{A}$ shall be given as $(l_0, 0^C) \in S$.*
For a transition $l_1 \xrightarrow{a,g,r} l_2 \ (\in T)$, the following two transitions are semantically defined. The former one is called an action transition, while the latter one is called a delay transition.

$$\frac{l_1 \xrightarrow{a,g,r} l_2, g(\nu), I(l_2)(r(\nu))}{(l_1, \nu) \overset{a}{\Rightarrow} (l_2, r(\nu))}, \qquad \frac{\forall d' \leq d \quad I(l_1)(\nu + d')}{(l_1, \nu) \overset{d}{\Rightarrow} (l_1, \nu + d)}$$

Definition 2.2.5 (A Semantic Model of a Timed Automaton)**.** *For timed automaton $\mathscr{A} = (A, L, l_0, C, I, T)$, an infinite transition system is defined according to the semantics of $\mathscr{A}$, where the model begins with the initial state.*

2.2.3 Region Automaton

For a given timed automaton $\mathscr{A}$, we can introduce a corresponding clock region $CR(\mathscr{A})$[56, 33]. In general, a clock region divides a $|C|$-dimensional Euclidean space into finite points, segments, and faces. By $[u]$, an element (a region) in $CR(\mathscr{A})$ is denoted. For $[u] \in CR(\mathscr{A})$, $g([u])$ and $I([u])$ represent that any point in $[u]$ satisfies a guard g and invariant I, respectively. Also by $r([u])$, applying clock resetting r onto $[u]$ is denoted, where $r([u]) = [u][x \mapsto 0]$, and $x \in r$.

Definition 2.2.6 (Region Automaton)**.** *A region automaton $\mathscr{A}_r = (A, L_r, l_{r\,0}, T_r)$ of a given timed automaton $\mathscr{A} = (A, L, l_0, C, I, T)$ is defined as follows.*
$L_r \subset L \times CR(\mathscr{A})$,
$l_{r\,0} = (l_0, [0^C])$, where $[0^C]$ satisfies $I(l_0)$,
$T_r \subset L_r \times A \times L_r$,
T_r consists of the transitions which satisfy the following property.

$$(l, [u]) \overset{a}{\Rightarrow} (l', [v]) \iff \exists d \in \mathbb{R}_{\geq 0}.\ (l, u) \overset{d}{\Rightarrow} (l, u') \in \mathscr{T}(\mathscr{A}) \\ \wedge (l, u') \overset{a}{\Rightarrow} (l', v) \in \mathscr{T}(\mathscr{A}) \ \wedge \ u \in [u] \wedge v \in [v].$$

There is bi-simulation equivalence between a timed automaton $\mathscr{A}$ and its region automaton $\mathscr{A}_r$ [1].

2.2.4 Zone Graph

In [16], a state space of timed automata, which has infinite semantic states, is represented as a finite state transition system called a zone graph. In this study, we treat a zone D as a set of clock assignments $\nu \in \mathbb{R}^C_{\geq 0}$ (For a zone D, $\nu \in D$ means the assignment ν satisfies all the inequalities in D). In addition to this, using a location l and a zone D, we describe a set of semantic states as $(l, D) = \{(l, \nu) \mid \nu \in D\}$. Also, for an initial location l_0, a set of initial states is denoted by $(l_0, D_0) = \{(l_0, 0^C + d) | (l_0, 0^C) \stackrel{d}{\Rightarrow} (l_0, 0^C + d) \in \Rightarrow\}$.

Paper[16] also gives operation functions on zones, such as up, and and so on, which represent elapsing time, intersection of time spaces and so on, respectively. For a given zone D, there is a minimal set of differential inequalities which is enough to represent D [16]. We use $Ineqset(D)$ to denote such a minimal set for D. $Ineqset(D)$ can be obtained by reduction operations on zones. A set of every state which satisfies an invariant $I(l)$ of location l is denoted by $(l, D_{I(l)})$ $(= \{(l, \nu) | I(l)(\nu)\})$.

When we create a zone graph from a timed automaton, we perform zone normalization called k-normalization[16], where $k : C \to \mathbb{N}$ is a clock ceiling, to prevent zones from increasing infinitely. The clock ceiling k is given by the maximal clock constants appearing in the automaton. In k-normalization, we represent zones that may contain arbitrarily large constants as a single representative zone. The details are given as follows; we remove the constraints of the form $x < m$, $x \leq m$, $x - y < m$, $x - y \leq m$ from the given zone, and also we replace the constraints of the form $x > m$, $x \geq m$, $x - y > m$, $x - y \geq m$ with $x > k(x)$, $x \geq k(x)$, $x - y > k(x)$, $x - y \geq k(x)$, where $x, y \in C$ and $m > k(x)$, respectively.

2.2.5 DBM (Difference Bound Matrix)

In [16, 57], a data structure DBM is introduced to represent a convex space in $|C|$ -dimensional Euclidean space, where C is a set of clock variables. DBM is a set of differential inequalities on two clock variables, and represents a state space which satisfies all inequalities over it (the state space is called a zone). DBM represents these set of inequalities as a $|C_0| \times |C_0|$ matrix, where $C_0 = C \cup \{ \mathbf{0} \}$, and $\mathbf{0}$ is a special variable which means a constant value 0. The (i, j)-th entry $(D_{i\,j})$ of the matrix means a differential inequality of $x_i - x_j$ for $x_i, x_j \in C_0$. Suppose there is an inequality $x_i - x_j \preceq n$ for $\preceq \in \{ <, \leq \}$, the (i, j)-th entry

$D_{i\,j}$ is represented by $(n, \preceq)$. Also, when $x_i - x_j$ is unbounded, the entry $D_{i\,j}$ is represented by ∞. In addition, the upper bound and lower bound of x_i itself are indicated by $D_{0,i}$ and $D_{i,0}$ respectively.

As an example of DBM, let's consider a zone which satisfies following constraint.

$$x - \mathbf{0} < 20 \wedge y - \mathbf{0} \leq 20 \wedge x - y \leq -10 \wedge y - x \leq 10 \wedge \mathbf{0} - z < 5.$$

When we represent this zone as DBM, variables $\mathbf{0},\ x,\ y,\ z$ are numbered with $0,\ 1,\ 2,\ 3$ respectively in the matrix. DBM which represents the zone of the constraint is given by (2.1).

$$D = \begin{pmatrix} (0,\leq) & (0,\leq) & (0,\leq) & (5,<) \\ (20,<) & (0,\leq) & (-10,\leq) & \infty \\ (20,\leq) & (10,\leq) & (0,\leq) & \infty \\ \infty & \infty & \infty & (0,\leq) \end{pmatrix}. \tag{2.1}$$

DBM is also represented as a set of some elements in the clock region $CR(\mathscr{A})$. Therefore a state set of states of a region automaton $\mathscr{A}_r = (L_r, l_{r\,0}, T_r, A)$, can be represented in $(l, D) = \{(l, [u]) \mid [u] \in D\}$ using the corresponding DBM D. Paper[16] gives operation functions on DBM, such as up, and and other functions, which represent elapsing time, intersection of time spaces and so on, respectively. There is a minimum set of differential inequalities which can represents DBM D [16]. Such a set is denoted by $c(D)$. $c(D)$ can be obtained by reduction operations on DBM. A set of every region which satisfies an invariant $I(l)$ of location l is denoted by (l, D_{Inv}).

2.2.6 Temporal Logic

Here, we give formal descriptions for temporal logic LTL and CTL.

LTL

LTL represents properties related to a path on a target model. The syntax of LTL is given as follows.

Definition 2.2.7 (Liner Temporal Logic)**.** *Syntax of LTL formulae is given as follows,*

$$\begin{aligned} \phi \ :=\ & true \mid false \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \\ & X\phi \mid F\phi \mid G\phi \mid \phi U \phi \mid \phi R \phi, \end{aligned}$$

where p is an arbitrary atomic proposition.

Also, we can interpret the semantics of LTL as follows.

Definition 2.2.8 (Semantics of LTL)**.** *For a transition system $TS = (S, R, L)$ where S is a set of states, $R : S \times S$ is transition relation on S, and $L : S \to 2^{AP}$ is function labeling states with atomic propositions, and for a path π on TS, the semantics of LTL is given as follows.*

$$\begin{aligned}
\pi \models p &\iff p \in L(\pi(0)) \\
\pi \models \neg\phi &\iff \textit{not } \pi \models \phi \\
\pi \models \phi_1 \wedge \phi_2 &\iff \pi \models \phi_1 \textit{ and } \pi \models \phi_2 \\
\pi \models \phi_1 \vee \phi_2 &\iff \pi \models \phi_1 \textit{ or } \pi \models \phi_2 \\
\pi \models \phi_1 \to \phi_2 &\iff \pi \models \phi_1 \textit{ implies } \pi \models \phi_2 \\
\pi \models F[\phi] &\iff \exists i.\pi^i \models \phi \\
\pi \models G[\phi] &\iff \forall i.\pi^i \models \phi \\
\pi \models X\phi &\iff \pi^1 \models \phi \\
\pi \models [\phi_1 U \phi_2] &\iff \exists i.(\pi^i \models \phi_2 \wedge \forall j.(0 \leq j < i \to \pi^j \models \phi_1)) \\
\pi \models [\phi_1 R \phi_2] &\iff \forall i.(\pi^i \models \phi_2 \vee \exists j.(0 \leq j < i \to \pi^j \models \phi_1)),
\end{aligned}$$

where p is an arbitrary atomic proposition, ϕ, ϕ_1 and ϕ_2 are arbitrary LTL formulae, $\pi(i)$ is i-th state in the path π, and π^i is the suffix of π after the i-th state.

CTL

CTL represents properties with tree like structures which have several branches. The syntax of CTL is given as follows.

Definition 2.2.9 (Computational Tree Logic)**.** *Syntax of CTL formulae is given as follows,*

$$\begin{aligned}
\phi \;:=\; & true \mid false \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \to \phi \mid \\
& AX\phi \mid EX\phi \mid AF\phi \mid EF\phi \mid AG\phi \mid EG\phi \mid A[\phi U \phi] \mid E[\phi U \phi],
\end{aligned}$$

where p is an arbitrary atomic proposition.

Definition 2.2.10 (Semantics of CTL)**.** *For a transition system $TS = (S, R, L)$ where S is a set of states, $R : S \times S$ is transition relation on S, and $L : S \to 2^{AP}$ is function labeling states with atomic propositions, and for a state $s \in S$, the*

semantics of CTL is given as follows.

$$
\begin{aligned}
s \models p &\iff p \in L(s) \\
s \models \neg\phi &\iff \textit{not } s \models \phi \\
s \models \phi_1 \wedge \phi_2 &\iff s \models \phi_1 \textit{ and } s \models \phi_2 \\
s \models \phi_1 \vee \phi_2 &\iff s \models \phi_1 \textit{ or } s \models \phi_2 \\
s \models \phi_1 \rightarrow \phi_2 &\iff s \models \phi_1 \textit{ implies } s \models \phi_2 \\
s \models AF[\phi] &\iff \textit{for all paths } (s_0, s_1, \ldots), \\
&\quad \exists i.(i \geq 0 \wedge s_i \models \phi) \\
s \models EF[\phi] &\iff \textit{for some paths } (s_0, s_1, \ldots), \\
&\quad \exists i.(i \geq 0 \wedge s_i \models \phi) \\
s \models AG[\phi] &\iff \textit{for all paths } (s_0, s_1, \ldots), \\
&\quad \forall i.(i \geq 0 \wedge s_i \models \phi) \\
s \models EG[\phi] &\iff \textit{for some paths } (s_0, s_1, \ldots), \\
&\quad \forall i.(i \geq 0 \wedge s_i \models \phi) \\
s \models AX\phi &\iff \textit{for all states } t \textit{ such that} (s,t) \in R, t \models \phi \\
s \models EX\phi &\iff \textit{for some states } t \textit{ such that} (s,t) \in R, t \models \phi \\
s \models A[\phi_1 U \phi_2] &\iff \textit{for all paths } (s_0, s_1, \ldots), \\
&\quad \exists i.(i \geq 0 \wedge s_i \models \phi_2 \wedge \forall j.(0 \leq j < i \rightarrow s_j \models \phi_1)) \\
s \models E[\phi_1 U \phi_2] &\iff \textit{for some paths } (s_0, s_1, \ldots), \\
&\quad \exists i.(i \geq 0 \wedge s_i \models \phi_2 \wedge \forall j.(0 \leq j < i \rightarrow s_j \models \phi_1))
\end{aligned}
$$

where p is an arbitrary atomic proposition, ϕ, ϕ_1 and ϕ_2 are arbitrary CTL formulae.

The CTL formulae $AF\phi$ and $EF\phi$ can be replaced by $A[true\, U\phi]$ and $E[true\, U\phi]$, respectively. Also $AG\phi$ and $EG\phi$ can be replaced by $\neg EF\neg\phi$ and $\neg AF\neg\phi$, respectively.

2.2.7 CounterExample-Guided Abstraction Refinement

Since model abstraction sometimes over-approximates an original model, we may obtain spurious CEs which are infeasible on the original model. Paper [11] gives an abstraction refinement framework called CEGAR (CounterExample-Guided Abstraction Refinement) (Fig.1.6).

In the algorithm, at the first step (called Initial Abstraction), it generates an initial abstract model. Next, it performs model checking on the abstract model. In this step, if the model checker reports that the model satisfies a given specification, we can conclude that the original model also satisfies the specification, because the abstract model is an over-approximation of the original model. If the model checker reports that the model does not satisfy the specification, however, we have to check whether the CE detected is spurious or not in the next step (called Simulation). In the Simulation step, if we find that the CE is valid, we stop the loop. Otherwise, we have to refine the abstract model to eliminate the spurious CE, and repeat these steps until valid output is obtained.

2.3 Proposed Algorithm

Our proposed algorithm generates an abstract model $\hat{M}$ from a given timed automaton $\mathscr{A}$, and performs model checking on $\hat{M}$. $\hat{M}$ is in fact a finite automaton. If a counter example $\hat{\rho}$ (represented as a sequence of states and labels on $\hat{M}$) is detected by model checking, we check whether $\hat{\rho}$ is feasible on the concrete model $\mathscr{T}(\mathscr{A})$ or not at the simulation step (In this study, for an abstract model $\hat{M}$ obtained from a timed automaton $\mathscr{A}$, we call the semantic model $\mathscr{T}(\mathscr{A})$ a concrete model of $\hat{M}$). In this step, we obtain a set Π of sequences of transitions on $\mathscr{A}$ corresponding to $\hat{\rho}$, and check whether each path in Π is feasible on $\mathscr{T}(\mathscr{A})$ or not. If every path in Π is infeasible, the next step shall refine the model so that the counter example $\hat{\rho}$ becomes infeasible. Our algorithm does not directly refine $\hat{M}$ but it modifies $\mathscr{A}$ and then obtains a new abstract model from the modified timed automaton $\mathscr{A}$.

The proposed algorithm checks a property $\mathtt{AG} \bigwedge_{e \in E} \neg e$, where E ($\subset L$) of a timed automaton $\mathscr{A}$ is a set of error locations of the target system. The property means there is no path to locations in E from the initial state. Please note that any counter example of such a property can be represented in a finite length of sequence without infinite loops. Therefore, hereafter, we assume that counter examples are finite sequences.

In model checking techniques, several properties presented in CTL[31], LTL[30], and others would be checked in general. The typical properties, however, are safety and progress. The reachability analysis is the primitive procedure for safety checking, thus model checking problems on several important properties represented in CTL could be reduced into the reachability analysis problem. Therefore, the reachability analysis is an important problem.

2.3.1 Abstract Model

The proposed method abstracts a given timed automata $\mathscr{A} = (A, L, l_0, C, I, T)$ by removing clock variables from $\mathscr{A}$. Therefore, the obtained abstracted model $\hat{M}$ will be $(\hat{S}, \hat{s}_0, \hat{\Rightarrow})$, where $\hat{S} = L$, $\hat{s}_0 = l_0$.

Here, we define the abstraction function $h : S \to \hat{S}$ which is a mapping from S to $\hat{S}$.

Definition 2.3.1 (Abstraction Function h)**.** *For a timed automaton $\mathscr{A}$ and its semantic model $\mathscr{T}(\mathscr{A}) = (S, s_0, \Rightarrow)$, an abstraction function $h : S \to \hat{S}$ is defined as follows:*

$$h((l, \nu)) = l.$$

The inverse function $h^{-1} : \hat{S} \to 2^S$ of h is also defined as $h^{-1}(\hat{s}) = (l, D_{I(l)})$ where $\hat{s} = l$.

The abstraction function should be defined for each iteration of the refinement, because both the concrete model and abstract models are deformed. Let $\mathscr{A}_i$ and $\hat{M}_i$ be a timed automaton and an abstract model of i-th iteration, respectively. The abstraction function h_i for the i-th loop is defined in the similar way as Definition 2.3.1.

Symbols decorated with '$\hat{}$' represent those of an abstract model (i.e. $\hat{S}$ represents a state set of an abstract model). Definition 2.3.2 gives an abstract model $\hat{M}$ of a given timed automaton $\mathscr{A}$ using the abstraction function h defined in Definition 2.3.1.

Definition 2.3.2 (Abstract Model)**.** *An abstract model $\hat{M} = (\hat{S}, \hat{s}_0, \hat{\Rightarrow})$ of a given timed automaton $\mathscr{A} = (A, L, l_0, C, I, T)$ and its semantic model $\mathscr{T}(\mathscr{A}) = (S, s_0, \Rightarrow)$ is defined as follows:*

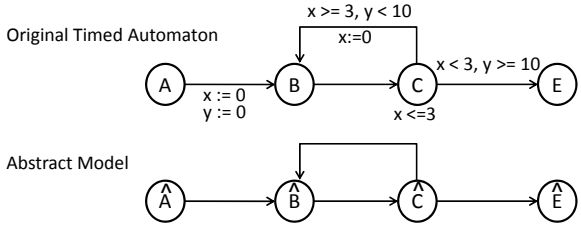
- $\hat{S} = L$*,*
- $\hat{s_0} = h(s_0)$*, and*
- $\hat{\Rightarrow} = \{(h(s_1), a, h(s_2)) \mid s_1 \overset{a}{\Rightarrow} s_2)\}$.

The i-th iteration of the refinement loop generates the i-th abstract model $\hat{M}_i = (\hat{S}_i, \hat{s}_{i,0}, \hat{\Rightarrow}_i)$ from the i-th timed automaton $\mathscr{A}_i = (A_i, L_i, l_{i,0}, C_i, I_i, T_i)$ by Definition 2.3.2.

Definition 2.3.3 (Abstract Counter Example)**.** *A counter example on $\hat{M} = (\hat{S}, \hat{s}_0, \hat{\Rightarrow})$ is a sequence of states of $\hat{S}$ and labels. An abstract counter example $\hat{\rho}$ of length n is represented in $\hat{\rho} = \langle \hat{s}_0 \overset{a_1}{\to} \hat{s}_1 \overset{a_2}{\to} \hat{s}_2 \overset{a_3}{\to} \cdots \overset{a_{n-1}}{\to} \hat{s}_{n-1} \overset{a_n}{\to} \hat{s}_n \rangle$. A set P of run*

Algorithm 2.1 Abstraction($\mathscr{A}$)

1: /* $\hat{M} = (\hat{S}, \hat{s}_0, \hat{\Rightarrow})$ */
2: $\hat{S} := L$
3: $\hat{s}_0 := l_0$
4: $\hat{\Rightarrow} := \emptyset$
5: **foreach** $(l_1, a, g, r, l_2) \in T$ **do**
6: $\quad \hat{\Rightarrow} := \hat{\Rightarrow} \cup \{l_1, a, l_2)\}$
7: **end for**
8: **return** $\hat{M}$



The location E is an error location.

Figure 2.1: Examples of a Timed Automaton and its Abstract Model

sequences on $\mathscr{T}(\mathscr{A})$ obtained by concretizing a counter example $\hat{\rho}$ is also defined as follows using the inverse function h^{-1}:

$$P = \{\langle s_0 \xrightarrow{d_0} s_0' \xrightarrow{a_1} s_1 \xrightarrow{d_1} s_1' \xrightarrow{a_2} s_2 \xrightarrow{d_2} \cdots \xrightarrow{a_n} s_n\rangle \mid \bigwedge_{i=0}^{n-1} (s_i \in h^{-1}(\hat{s}_i) \ \wedge d_i \in \mathbb{R}_{\geq 0} \wedge s_i \xrightarrow{d_i} s_i' \ \wedge s_i' \xrightarrow{a_i} s_{i+1})\}.$$

If P has at least one element, we find that the counter example $\hat{\rho}$ is feasible on the original timed automaton. Otherwise we find that $\hat{\rho}$ is spurious.

2.3.2 Initial Abstraction

Initial Abstraction generates an abstract model $\hat{M}_0$ shown in Sec.2.3.1 from a timed automaton $\mathscr{A}_0$. Figure 2.1 represents an example of a timed automaton and its abstract model obtained by applying Initial Abstraction to the timed automaton.

Algorithm 2.1 shows the algorithm of Initial Abstraction.

2.3.3 Simulation

For an abstract counter example $\hat{\rho} = \langle \hat{s}_0 \xrightarrow{a_1} \hat{s}_1 \xrightarrow{a_2} \hat{s}_2 \xrightarrow{a_3} \cdots \xrightarrow{a_n} \hat{s}_n\rangle$, Simulation checks if a set P of the corresponding run sequences on the concrete model is

empty or not.

It is difficult to obtain P directly on the semantic model, because P may have infinite of sequences. Therefore, in the algorithm, first we obtain a set Π of sequences of transitions on $\mathscr{A}$ corresponding to $\hat{\rho}$. Then, we check if each element in Π is feasible on the $\mathscr{T}(\mathscr{A})$ or not. If there is an element $\pi \in \Pi$ that is feasible on $\mathscr{T}(\mathscr{A})$, we can conclude that $\hat{\rho}$ is a feasible counter example. On the other hand, if all the elements in Π are infeasible on $\mathscr{T}(\mathscr{A})$, we can conclude that $\hat{\rho}$ is spurious. The set Π of sequences of transitions on $\mathscr{A}$ corresponding to $\hat{\rho} = \langle \hat{s}_0 \overset{a_1}{\to} \hat{s}_1 \overset{a_2}{\to} \hat{s}_2 \overset{a_3}{\to} \cdots \overset{a_n}{\to} \hat{s}_n \rangle$ is given as follows;

$$\Pi = \{\langle l_0 \overset{a_1,g_1,r_1}{\longrightarrow} l_1 \overset{a_2,g_2,r_2}{\longrightarrow} l_2 \overset{a_3,g_3,r_3}{\longrightarrow} \cdots \overset{a_n,g_n,r_n}{\longrightarrow} l_n \rangle \mid \bigwedge_{i=0}^{n} \hat{s}_i = l_i \wedge \bigwedge_{i=1}^{n} (l_{i-1}, a_i, g_i, r_i, l_i) \in T\}.$$

Π may have a finite number of sequences corresponding to $\hat{\rho}$ because there might be several transitions in $\mathscr{A}$ corresponding to the transition $\hat{s}_{i-1} \overset{a_i}{\Rightarrow} \hat{s}_i$ even if $\hat{s}_i$ always corresponds to the single location l_i.

Whether $\pi \in \Pi$ is feasible on the $\mathscr{A}$ is determined by calculating a reachable state set on $\mathscr{A}$ along with π. In this process, when the reachable state set becomes empty, we can conclude that π is infeasible.

In the Definition 2.3.4, we define the successor state set to be reachable by one action transition followed by arbitrary delay transitions.

Definition 2.3.4 (Successor State Set)**.** *Given a state set (l_1, D_1) on $\mathscr{T}(\mathscr{A})$ for a timed automaton $\mathscr{A}$ and a transition $e = (l_1, a, g, r, l_2)$, a successor state set $succ((l_1, D_1), e)$ from (l_1, D_1) through the transition e is defined as follows;*

$$succ((l_1, D_1), e) = \{(l_2, r(\nu)+d) \mid \nu \in D_1 \wedge d \in \mathbb{R}_{\geq 0} \wedge (l_1, \nu) \overset{a}{\Rightarrow} (l_2, r(\nu)) \wedge (l_2, r(\nu)) \overset{d}{\Rightarrow} (l_2, r(\nu) + d) \ \}.$$

Lemma 2.3.1. *Given a state set (l_1, D_1) on $\mathscr{T}(\mathscr{A})$ of a timed automaton $\mathscr{A}$ and a transition $e = (l_1, a, g, r, l_2)$, a reachable state set $succ((l_1, D_1), e)$ $(= (l_2, D_2))$ satisfies the following property;*

$$\forall \nu \in D_2, \forall d \in \mathbb{R}_{\geq 0}.\ (l_2, \nu) \overset{d}{\Rightarrow} (l_2, \nu + d) \text{ implies } \nu + d \in D_2\ .$$

Lemma 2.3.1 is proved by Definition 2.3.4 obviously.

From Definition 2.3.4, a k-th reachable state set from the initial state set (l_0, D_0) is obtained by applying $succ$ function k times like $succ(succ(succ(\ldots succ((l_0,$

Algorithm 2.2 Simulation($\mathscr{A}, \pi$)

```
1: /* π = (l_0 --a_1,g_1,r_1--> l_1 --a_2,g_2,r_2--> ... --a_n,g_n,r_n--> l_n(l_n = e)) */
2: D_0 := {0^C}
3: D_0 := up(D_0)                                   /* Any elapsing time */
4: D_0 := and(D, I(l_0))                            /* Add Invariant of l_0 */
5: succ_list_0 := (l_0, D_0)
6: for i := 1 to n do
7:     succ_list_i := Succ(A, succ_list_{i-1}, (l_{i-1}, a_i, g_i, r_i, l_i))
8:     if succ_list_i = ∅ then
9:         return succ_list
10:    end if
11: end for
12: return NULL                                     /* The counterexample can be reproduced */
```

Algorithm 2.3 Succ($\mathscr{A}, R, e$)

```
1: /* To obtain a reachable zone by the given action transition */
2: /* R = (l_1, D), e = (l_1, a, g, r, l_2)) */
3: D := and(D, g)                                   /* add guards of transitions */
4: D := reset(D, r)                                 /* reset the clocks */
5: D := and(D, I(l_2))                              /* add Invariant of l_2 */
6: /* To obtain a reachable zone by delay transitions */
7: D := up(D)                                       /* Any elapsing time */
8: D := and(D, I(l_2))                              /* add Invariant of l_2 */
9: return (l_2, D)
```

$D_0), e_0) \dots), e_{k-2}), e_{k-1})$. In the rest of this thesis, by $succ^k(\pi)$ the k-th reachable state set for π is denoted.

For the sequence π of the length n, π is feasible if $succ^n(\pi) \neq \emptyset$, and π is infeasible if there exists $1 \leq k \leq n$ such that $succ^k(\pi)$ equals $\emptyset$.

Algorithm 2.2 represents our Simulation algorithm whose inputs are a timed automaton $\mathscr{A}$ and an element π in Π. It checks whether π is feasible on $\mathscr{T}(\mathscr{A})$ or not. The algorithm outputs $NULL$ if π is feasible, and otherwise the list of reachable state set $succ\_list$ along with π, which is used in the Refinement step. Algorithm 2.3 shows the algorithm to obtain the successor state set from a given state set and an action transition.

The functions $and : c(C) \times c(C) \rightarrow c(C)$, $up : c(C) \rightarrow c(C)$, and $reset : c(C) \times 2^C \rightarrow c(C)$ used in the algorithm are defined as follows. $and(D, g) = \{\nu | \nu \in D \wedge g(\nu)\}$, $up(D) = \{\nu' | \nu \in D \wedge d \in \mathbb{R}_{\geq 0} \wedge \nu \stackrel{d}{\Rightarrow} \nu'\}$, $reset(D, r) = \{r(\nu) | \nu \in D\}$.

Figure 2.2 represents a Simulation process in which an abstract counter exam-

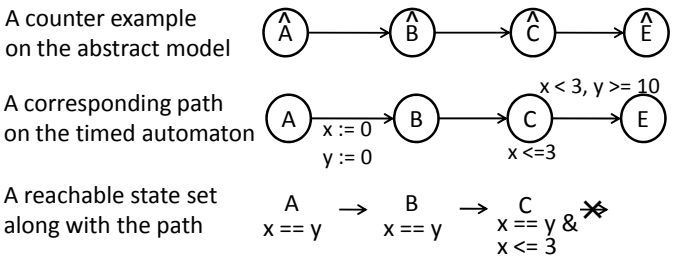


Figure 2.2: A Simulation Process

ple on the abstract model of Fig.2.1 is checked. As shown in the figure, a reachable state set is represented as a product of a location and a zone. Since any reachable states on the location C don't satisfy the guard condition $x \leq 3 \wedge y \geq 10$ for the transition from C to E, they cannot reach to the error location E. Therefore, we can conclude the counter example is spurious.

2.3.4 Abstraction Refinement

In the example of Fig.2.2, from any state on the location C that is reachable from the initial state, the control cannot move to E due to the guard condition. On its abstract model, however, $\hat{E}$ is reachable because we do not consider the clock constraints on it. This is the cause of the spurious counter example. Generally in such a case, we have to refine the abstract model by dividing the abstract state $\hat{C}$ so that the state set which is reachable from the initial state and the state set which is able to move to D become disjoint. Dividing a state space of a timed automaton usually needs a subtraction operation on zones. However, zones are not closed under the subtraction operation[57]; therefore, applying such an approach is difficult. In our approach, we transform the transition relation on the timed automaton preserving its equivalence so that the model behaves correctly even if we don't consider the clock constraints.

Algorithm 2.4 represents our abstraction refinement algorithm. In the algorithm, we first apply equivalent transformation to the original model ($l3$-$l15$). Next, we generate the refined abstract model by removing clock variables from the transformed model ($l16$). The transformation algorithm on the original model is composed of three steps; Duplication of Locations, Duplication of Transitions, and Removal of Transitions. Since we have to preserve model equivalence, we impose the restriction for applying removal of transitions.

These transformation steps add or remove some locations and transitions. Therefore, at each step, we have to construct a timed automaton for the step with new

Algorithm 2.4 Refinement($\mathscr{A}_i, \pi, succ\_list$)

/* $\pi = \langle l_0 \xrightarrow{a_1,g_1,r_1} l_1 \xrightarrow{a_2,g_2,r_2} \cdots \xrightarrow{a_n,g_n,r_n} l_n(l_n = e)\rangle$ */
/* $succ\_list = \langle (l_0, D_0), (l_1, D_1), \cdots, (l_k, D_k)\rangle$,
where (l_j, D_j) represents the j-th reachable state set along with π, and l_k is the last location reachable from the initial state. */
$\mathscr{A}_{i+1} := \mathscr{A}_i$
for $j := succ\_list.length$ downto 1 **do**
  $e_j := (l_{j-1}, a_{j-1}, g_{j-1}, r_{j-1}, l_j)$
  $\mathscr{A}_{i+1} := Duplication(\mathscr{A}_{i+1}, succ\_list_j, e_j)$
    /* Duplication of the Location and Transitions */
  if $IsRemovable(\mathscr{A}_{i+1}, succ\_list_j, e_j)$ **then**
    $\mathscr{A}_{i+1} := RemoveTransition(\mathscr{A}_{i+1}, e_j)$ /* Removal of Transitions */
    break
  else if $j = 1$ **then**
    $\mathscr{A}_{i+1} := DuplicateInitialLocation(\mathscr{A}_{i+1}, (l_0, D_0))$
      /* Duplicate the initial location and transitions from the initial location */
  end if
end for
$\hat{M}_{i+1} := Abstraction(\mathscr{A}_{i+1}, h)$
return $\hat{M}_{i+1}$

Algorithm 2.5 DuplicateInitialLocation($\mathscr{A}, R_0$)

/* $R_0 = (l_0, D_0)$ */
$l'_0 := newLoc()$ /* Generate a new location */
$I(l'_0) := Ineqset(D_0)$ /* A set of inequalities representing D_0 */
$L := L \cup \{l'_0\}$
foreach $(l_0, a, g, r, l) \in T$ **do**
  $(l, D) := Succ(\mathscr{A}, (l_0, D_0), (l_0, a, g, r, l))$
  if $(l, D) \neq \emptyset$ **then**
    /* Duplicate transitions only feasible from (l_0, D_0) */
    $l' := DuplicateOf(\mathscr{A}, (l, D))$
    if $l' = \perp$ **then**
      $T := T \cup \{(l'_0, a, g, r, l)\}$ /* Case (2.3) of Def.2.3.9 */
    else
      $T := T \cup \{(l'_0, a, g, r, l')\}$ /* Case (2.4) of Def.2.3.9 */
    end if
  end if
end for
$l_0 := l'_0$
return $\mathscr{A}$

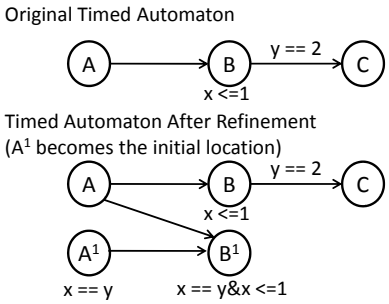


Figure 2.3: An Example of the Algorithm Reaches to the Initial Location

locations and transitions as well as invariants. We can discuss equivalence on each of the new timed automaton and the original timed automaton.

The algorithm starts with the reachable last location along with π, and if it cannot apply Removal of Transitions, it traces back π applying the refinement algorithm until being able to apply the removal operation. Finally, if the algorithm reaches to the initial location, it duplicates (l_0, D_0) as $\hat{l}'_0$. Figure 2.3 shows an example where the algorithm reaches the initial location. Here, at the location A, the algorithm cannot remove a transition from A to B. For such a case, it duplicates the initial location A. We let the new duplicated location A^1 be the initial location.

Duplication of Locations

In Definitions 2.3.5, 2.3.6 and 2.3.7, we give some definitions related to the duplicated locations.

Definition 2.3.5 (A Parent of a Location)**.** *Let l' be a duplication of the location l, and we call l the parent of l'. The function $parent : L \to L$ is defined as follows;*

$$parent(l) = \begin{cases} l\text{'s parent}, & \text{if } l \text{ has the parent} \\ \bot, & \text{if } l \text{ has no parent.} \end{cases}$$

Definition 2.3.6 (A Root of a Location)**.** *A root of a location is the eldest ancestor parent of the location. The function $root : L \to L$ is defined as follows;*

$$root(l) = \begin{cases} l, & \text{if } parent(l) = \bot \\ root(parent(l)), & \text{if } parent(l) \neq \bot. \end{cases}$$

Algorithm 2.6 DuplicateOf($\mathscr{A}, R$)

```
1: /* R = (l, D) */
2: l' := ⊥
3: D_norm := norm_k(D)                    /* k-normalize the zone D */
4: foreach l_1 ∈ L such that root(l_1) = root(l) do
5:    if D_{I(l_1)} = D_norm then
6:       l' := l_1
7:       break
8:    end if
9: end for
10: return l'
```

In this study, we use the function $root$ to decide whether given locations are duplicated from the same original location. For example, for locations l_1 and l_2 with $root(l_1) = root(l_2)$, we regard l_1 and l_2 are derived from the same original location.

Definition 2.3.7. *Let $\mathscr{T}(\mathscr{A}) = (S, s_0, \Rightarrow)$ be a semantic model of a timed automaton $\mathscr{A} = (A, L, l_0, C, I, T)$. For a location $l \in L$ and a zone $D \in c(C)$, the function $duplicateof : 2^S \times (L \to c(C)) \to L$ is defined as follows;*

$$duplicateof((l, D), I) = \begin{cases} l', & \text{if there exists a location } l' \in L \\ & \quad \text{such that } root(l') = root(l) \\ & \qquad \wedge D_{I(l')} = D \\ \bot, & \text{otherwise.} \end{cases}$$

For a given state set (l, D), $duplicateof$ returns the duplicated location l' of l such that the invariant $I(l')$ corresponds to D. In the definition, $D_{I(l')} = D$ means the equivalence of a zone corresponding to $I(l')$ and D (i.e. $\nu \in D_{I(l')} \iff \nu \in D$).

The Algorithm 2.6 implements the function $duplicateof$ defined in Definition 2.3.7. The function $norm_k$ is the k-normalization function defined in [16]. At the line 4 of the algorithm, it checks whether the given zone D is equivalent to $I(l')$ though D is normalized to D_{norm}. If this condition is satisfied for l', we can find that the duplicated location based on the state set (l, D) is already generated, and is l'.

Definition 2.3.8 (Duplication of Locations)**.** *Given a timed automaton $\mathscr{A} = (A, L, l_0, C, I, T)$ and a path π corresponding to the spurious counter example, suppose that we apply refinement to the k-th location l_k and transition e_k of π. And let $succ^k(\pi)$ equal (l_k, D_k). In this case, we generate a new location l'_k as a duplicate*

of l_k only if $duplicateof((l_k, D_k), I) = \bot$. Also, we impose a location invariant on l'_k as $I(l'_k) = Ineqset(D_k)$.

Invariant $I(l')$ for the duplicated location l' is stronger than that ($I(l)$) of the original location l. Therefore, for zone representation of them, $D_{I(l')} \subset D_{I(l)}$ holds.

For semantic states $s = (l, \nu)$ and $s' = (l', \nu')$, we consider s' is the duplicate of s if $l = parent(l')$ and $\nu' = \nu$ hold.

Duplication of Transitions

Suppose that we apply refinement to the k-th location l_k and transition e_k of a path π corresponding to the spurious counter example. In Duplication of Transition, we duplicate the transition e_k and also transitions which are feasible on the reachable states on l_k. In Def.2.3.9, we define the transitions to duplicate.

Definition 2.3.9 (Duplication of Transitions)**.** *For a timed automaton $\mathscr{A} = (A, L, l_0, C, I, T)$ and a path π corresponding to the spurious counter example, suppose that we apply refinement to the k-th location l_k and transition $e_k = (l_{k-1}, a_k, g_k, r_k, l_k)$ of π. And also let l'_k be the duplicate of the k-th reachable state set $succ^k(\pi)$ $(= (l_k, D_k))$. Then l'_k equals $duplicateof((l_k, D_k), I)$ $(\neq \bot)$. In Duplicate of Transitions, we duplicate the following transitions;*

- *a duplicate of e_k*
$$(l_{k-1}, a_k, g_k, r_k, l'_k) \tag{2.2}$$
- *duplicates of action transitions $e = (l_k, a, g, r, l) \in T$ which are feasible from (l_k, D_k)*
  - *In the case when a duplicate location corresponding to $succ((l_k, D_k), e)$ has not been generated;*
$$\{\ (l'_k, a, g, r, l)\ \mid\ e = (l_k, a, g, r, l) \in T\ \wedge\ (l, D) = succ((l_k, D_k), e) \neq \emptyset \wedge duplicateof((l, D), I) = \bot\ \} \tag{2.3}$$
  - *In the case when a duplicate location corresponding to $succ((l_k, D_k), e)$ has been generated;*
$$\{\ (l'_k, a, g, r, l')\ \mid\ e = (l_k, a, g, r, l) \in T\ \wedge\ (l, D) = succ((l_k, D_k), e) \neq \emptyset \wedge l' = duplicateof((l, D), I) \neq \bot\} \tag{2.4}$$

Algorithm 2.7 Duplication($\mathscr{A}, R_k, e_k$)

```
1: /* R_k = (l_k, D_k), e_k = (l_{k-1}, a_k, g_k, r_k, l_k) */
2: l'_k := DuplicateOf(A, (l_k, D_k))
3: if l'_k = ⊥ then
4:    /* Duplicate a Location */
5:    l'_k := newLoc()                          /* Generate a new location */
6:    L := L ∪ {l'_k}
7:    D_norm := norm_k(D_k)                     /* k-normalize the zone D_k */
8:    I( l'_k ) := Ineqset(D_norm)              /* A set of inequalities representing D_k */
9: end if
10:
11: /* Duplicate transitions */
12: T := T ∪ {(l_{k-1}, a_k, g_k, r_k, l'_k)}   /* Case (2.2) of Def.2.3.9 */
13: foreach (l_k, a, g, r, l) ∈ T do
14:    (l, D) := Succ(A, (l_k, D_k), (l_k, a, g, r, l))
15:    if (l, D) ≠ ∅ then
16:                                  /* Duplicate transitions only feasible from (l_k, D_k) */
17:       l' := DuplicateOf(A, (l, D))
18:       if l' = ⊥ then
19:          T := T ∪ {(l'_k, a, g, r, l)}      /* Case (2.3) of Def.2.3.9 */
20:       else
21:          T := T ∪ {(l'_k, a, g, r, l')}     /* Case (2.4) of Def.2.3.9 */
22:       end if
23:    end if
24: end for
25: return A
```

Expression (2.2) duplicates a transition e_k (a transition from l_{k-1} to l_k). The duplicated one is a transition from l_{k-1} to l'_k.

Expressions (2.3) and (2.4) duplicate a transition e such that its starting location is l_k and can fire from $succ^k(\pi)$. The duplicated one is a transition from l'_k. Expression (2.4) is for the case that there exists a duplicated location l' such that l' is derived from (l, D) and (l, D) is reachable from $succ^k(\pi)$ via e. For such a case, it duplicates a transition to l' instead of l.

Algorithm 2.7 shows both of the duplication algorithms for locations and transitions, because duplication algorithm for transitions depends on the information of that for locations.

Here, we give a lemma about transitions duplicated in the algorithm.

Lemma 2.3.2. *Given a timed automaton $\mathscr{A}$ and a path π corresponding to the spurious counter example, suppose that we apply refinement to the k-th location l_k and transition e_k of π, and let $(l_k, D_k) = succ^k(\pi)$. The semantic model $\mathscr{T}(\mathscr{A}) =$*

Algorithm 2.8 RemoveTransition($\mathscr{A}, e$)

1: /* e is a transition to remove */
2: $T := T \setminus \{e\}$
3: **return** $\mathscr{A}$

Algorithm 2.9 IsRemovable($\mathscr{A}, R, e$)

1: /* $R = (l', D'), e = (l, a, g, r, l')$ */
2: $R' := Succ(\mathscr{A}, (l, D_{I(l)}), e)$ /* obtain the whole states reachable from l */
3: **return** $(R = R')$

$(S, s_0, \Rightarrow)$ *of* $\mathscr{A}$ *satisfies a following property;*

> *There exists a transition* $(s_1, a, s_2) \in \Rightarrow$ *such that* $s_1 \in (l_k, D_k)$*, if and only if there exists a duplicated transition* $(s'_1, a, s'_2) \in \Rightarrow$ *such that* s'_1 *is the duplicate of* s_1*, and* $s'_2 = s_2$ *or* s'_2 *is the duplicate of* s_2*.*

Proof. From the Expressions (2.3) and (2.4) in the Definition 2.3.9, we duplicate all the transitions which are feasible from the states in (l_k, D_k). Also, for transitions duplicated in the step, the transformed automaton has the original transitions on which the duplication is based. Therefore, $\mathscr{T}(\mathscr{A})$ satisfies the given property. □

Removal of Transitions

Let $e_k = (l_{k-1}, a_k, g_k, r_k, l_k)$ be the k-th action transition in a path π, and $(l_k, D_k) = succ^k(\pi)$ be the k-th reachable state set on π. Here, we also assume that a duplicated location $l'_k = duplicateof((l_k, D_k), I)$ and a duplicated transition $e'_k = (l_{k-1}, a_k, g_k, r_k, l'_k)$ are generated by the latest application of the duplication operations. Here, let us consider that a semantic state $s = (l_k, \nu)$ and its duplicated state $s' = (l'_k, \nu)$ are essentially equivalent (all the enable transitions from the states are equivalent) for all $\nu \in D_k$. Then, if the successor state set $succ((l_{k-1}, D_{I(l_{k-1})}), e_k)$ is equal to (l_k, D_k), it seems that we can substitute e'_k for e_k because $succ((l_{k-1}, D_{I(l_{k-1})}), e'_k)$ is equal to (l'_k, D_k). From this fact we can conclude that even if we remove the transition e_k, the equivalence among the semantic models is preserved. The equivalence is proved in Sec.2.4.

We define the condition to remove transitions in Definition 2.3.10. The function $isRemovable$ in the definition is implemented as represented in Algorithm 2.9.

Definition 2.3.10 (Removable Transitions)**.** *Let* $\mathscr{T}(\mathscr{A}) = (S, s_0, \Rightarrow)$ *be a semantic model of a timed automaton* $\mathscr{A} = (A, L, l_0, C, I, T)$*. For a transition*

$e = (l_1, a, g, r, l_2) \in T$ *and a state set* $(l_2, D_2) \subset S$*, the function* $isRemovable$: $T \times 2^S \times (L \to c(C)) \to$ *bool is defined as follows;*

$$isRemovable(e, (l_2, D_2), I) = \begin{cases} \mathbf{true}, & \textit{if } succ((l_1, D_{I(l_1)}), e) = (l_2, D_2) \\ \mathbf{false}, & \textit{otherwise}. \end{cases}$$

In our approach, as represented in Algorithm.2.4, we finish the refinement operation when $isRemovable(e_k, succ^k(\pi), I)$ becomes true for the first time. Though we are able to produce a finer model if we continue the refinement operation, we finish the operation at this point to produce a coarser model that is enough to remove the spurious counter example.

We define transitions to remove in Definition 2.3.11, and in Algorithm 2.8 the algorithm for removal of transitions is represented.

Definition 2.3.11 (Removal of Transitions)**.** *For a path* π *on a timed automaton* $\mathscr{A} = (A, L, l_0, C, I, T)$ *corresponding to the spurious counter example, let* m *be a minimum positive integer which satisfies* $succ^m(\pi) = \emptyset$*. For a maximum positive integer* $k \leq m$ *which satisfies* $isRemovable(e_k, succ^k(\pi), I) = true$*, we remove the* k*-th transition* e_k *from* T *in the removal of transitions, if there exists such an integer* k*.*

In the rest of this thesis, we describe $isRemovable(e_k, succ^k(\pi), I)$ as $isRemovable(\pi, k, I)$ simply.

Lemma 2.3.3. *If a transition was removed by Removal of Transitions, the same transition will not be generated by Duplication of Transition.*

Proof. Without loss of generality, we assume that a transition $e = (l_1, a, g, r, l_2)$ is removed in the i-th loop.

Here, there is (l_2, D_2) such that $isRemovable(e, (l_2, D_2), I_j)$ is true, and $succ((l_1, D_{I_j(l_1)}), e)$ is equal to (l_2, D_2) by Definition 2.3.11. By Sec.2.3.4, there is a duplicated location l_2' such that $l_2' = duplicateof((l_2, D_2), I_j))$ and a duplicated transition (l_1, a, g, r, l_2'). $D_2 \subset D_{I_i(l_2)}$ also holds.

We have to consider three cases that the transition e is regenerated in $j(> i)$-th loop. The three cases are corresponding to Expressions (2.2), (2.3), and (2.4) of Definition 2.3.9. Let assume that a sequence which is corresponding to a counter example detected in the j-th loop, is π_j.

Case Expression (2.2): In such a case, the $(k-1)$-th location of π_j is l_1. Also the k-th transition is $e_k = (l_1, a, g, r, l_k)(root(l_k) = root(l_2))$, where l_2 equals $duplicateof(succ(\pi_j, k)), I_j)$. However, a set of every reachable state from l_1 via e_k is (l_k, D_2). From the fact $D_2 \subset D_{I_j(l_2)} (= D_{I_i(l_2)}))$, $l_2 = duplicateof(succ(\pi, k)), I_j)$ does not hold.

Case Expression (2.3) and (2.4): Let us assume that the k-th location of π_j is l_k and $succ(\pi_j, k)$ is equal to (l_k, D_k). Then for a transition $e' = (l_k, a, g, r, l) \in T_j(root(l) = root(l_2))$, It should hold that $l_1 = duplicateof\,(l_k, D_k)$, and $duplicateof\,(succ(l_k, D_k), e'), I_j) = l_2$ or $\perp$. From the facts $duplicateof\,(succ(l_k, D_k), e'), I_j) = l'_2$ and $l'_2 \neq l_2$ hold, which contradicts.

Thus, we can conclude that for any case of Definition 2.3.9, the transition e will not be duplicated. Therefore, the lemma is proved. □

2.3.5 Example

Figure 2.4 shows a process of Refinement for a counter example in Fig.2.2. The last states reachable from the initial location through the path is represented as $(C, x \leq 3 \wedge x == y)$. The algorithm duplicates the location C and obtains a duplicated location C^1. Then it duplicates transitions by Definition 2.3.9. This process is shown in Fig.2.4 (a)-(c). The transition from C to B can fire in the state $(C, x \leq 3 \wedge x == y)$. Therefore it generates a transition from C^1 (Fig.2.4 (c)), however it does not duplicate the transition from C to E, because it cannot fire in the state $(C, x \leq 3 \wedge x == y)$.

At this point, if we could remove a transition from B to C, we would have removed the counter example. However, we cannot remove the transition, because the state set of C reachable from any state which satisfies the invariant of B is $(C, x \leq 3)$ and it does not equal $(C, x \leq 3 \wedge x == y)$ which is the reachable state set of C through the path. Instead of this, we perform the algorithm to location B backwards the path (Fig.2.4 (d)-(g)).

Figure 2.4 (f) shows the process of Duplication of Transitions from B to C. Here, the state set of C reachable from the state set $(B, x == y)$, which is the reachable state set on B via the path, is $(C, x \leq 3 \wedge x == y)$. The corresponding location for $(C, x \leq 3 \wedge x == y)$ is already generated as location C^1, hence $duplicateof\,((C, x \leq 3 \wedge x == y), I) = C^1$. Therefore the transition from B to C is duplicated as a transition from B^1 to C^1.

Figure 2.4 (g) shows the process of removal of a transition. A state set of B reachable from $(A, true)$ is $(B, x == y)$, which is equal to a state set of B reachable from the initial state set $(A, x == y)$. Therefore a transition e_1 from A to B is removable, because $isRemovable(e_1, (B, x == y), I)$ is true.

Figure 2.5 (a) and (b) show the obtained timed automata applying second and third refinements, respectively. In the second iteration, a spurious counter example $\langle \hat{A} \xrightarrow{\tau} \hat{B}^1 \xrightarrow{\tau} \hat{C}^1 \xrightarrow{\tau} \hat{B} \xrightarrow{\tau} \hat{C} \xrightarrow{\tau} \hat{E} \rangle$ is detected for the abstract model for the model in Fig.2.4 (g) (We assume that unlabelled transition is labelled by τ). For such a spurious counter example, we can obtain a timed automaton in Fig.2.5 (b).

In a similar way, the third iteration detects counter example $\langle \hat{A} \xrightarrow{\tau} \hat{B}^1 \xrightarrow{\tau}$

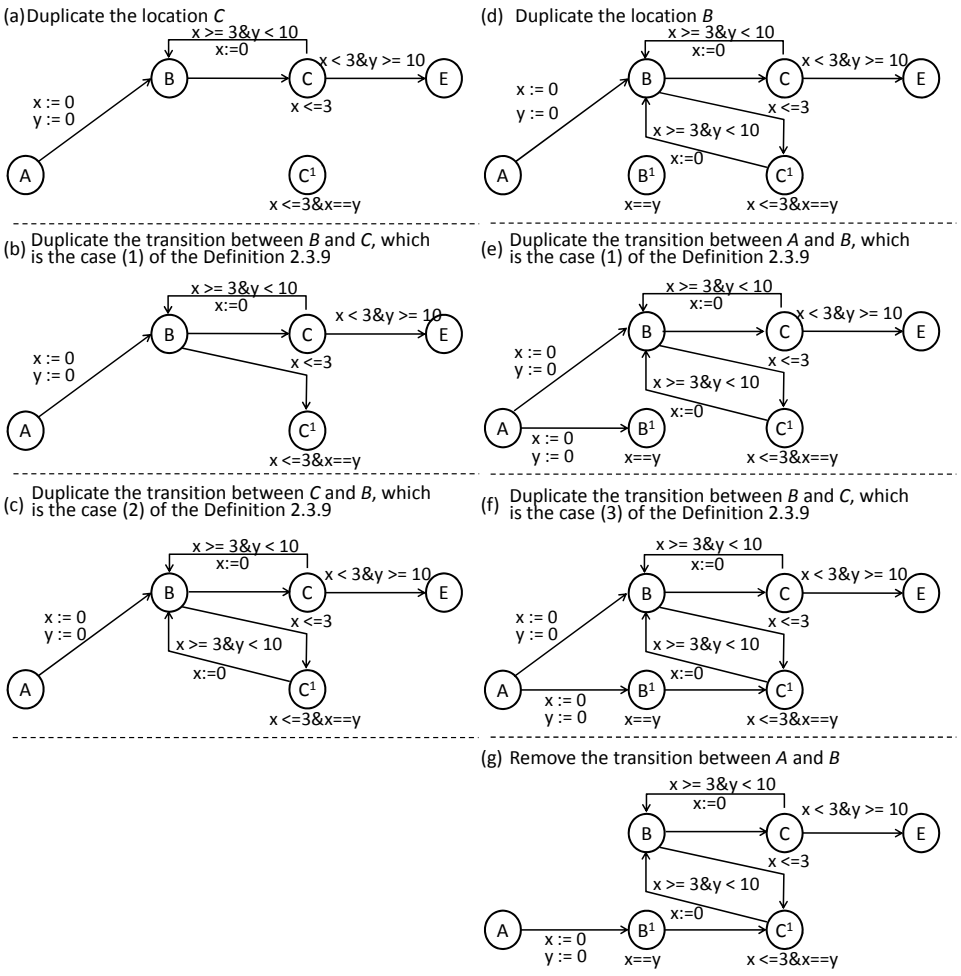


Figure 2.4: The Refinement Process for the Path in Fig.2.2

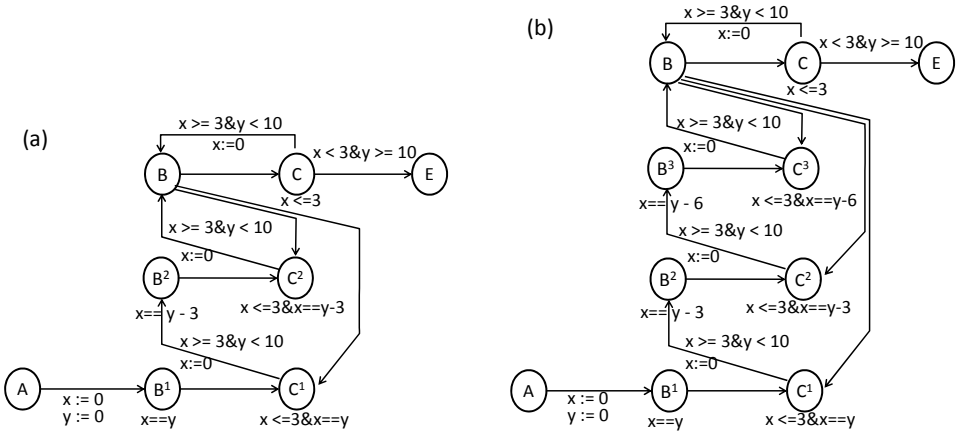


Figure 2.5: The Timed Automata After the Second ((a) of the Figure) and Third ((b) of the Figure) Refinement Steps Respectively

$\hat{C}^1 \xrightarrow{\tau} \hat{B}^2 \xrightarrow{\tau} \hat{C}^2 \xrightarrow{\tau} \hat{B} \xrightarrow{\tau} \hat{C} \xrightarrow{\tau} \hat{E}\rangle$, which is a true counter example. As a result, the algorithm terminates with the report of the counter example.

2.4 Correctness Proof

Paper [12] gives a theorem on a conservative class of abstractions which attempts to preserve semantics of automata against state reductions under the condition that it checks only a property $\mathtt{AG}\ p$ for a proposition p.

From the theorem, we can derive the following theorem.

Theorem 2.4.1. *Given a timed automaton $\mathscr{A} = (A, L, l_0, C, I, T)$, $E \subset L$ be a set of error location and $\hat{M}$ be the abstract model. The following statement always holds.*

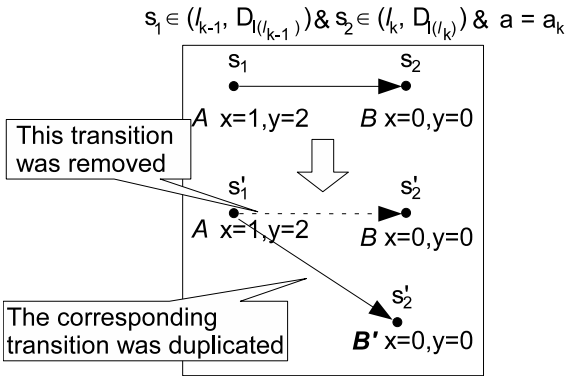
$$\hat{M} \models \mathtt{AG} \bigwedge_{e \in E} \neg \hat{e} \Rightarrow \mathscr{A} \models \mathtt{AG} \bigwedge_{e \in E} \neg e \tag{2.5}$$

Proof. Let the semantic model of $\mathscr{A}$ be $\mathscr{T}(\mathscr{A}) = (S, s_0, \Rightarrow)$. For a proposition p, if an abstraction function h satisfies the following for every $s \in S$:

$$h(s) \models p \Rightarrow s \models p \tag{2.6}$$

then $\hat{M} \models \mathtt{AG}\, p \Rightarrow M \models \mathtt{AG}\, p$ holds by Theorem 1 in Paper [12].

Here we assume that p equals $\bigwedge_{e \in E} \neg e$. From the Definition 2.3.1, $h(l, \nu)$ equals l. Therefore, $h((l, \nu)) \models \bigwedge_{e \in E} e \Rightarrow (l, \nu) \models \bigwedge_{e \in E} e$ obviously holds. As a result, the abstraction function h satisfies the statement 2.6; Theorem 2.4.1 is proved. □



The transitions at the top are those of the semantic model of Fig.2.1 and at the bottom are those of Fig.2.4 (g).

Figure 2.6: An Example of the Case When $s_1 \stackrel{a}{\Rightarrow} s_2 \not\in \Rightarrow'$ in the Proof (i) of Lemma 2.4.1

Next, we prove that the transformation on a timed automaton is equivalent transformation, and also that the abstract model generated by the refinement algorithm is the refined model of the former abstract model.

We prove the equivalence of the transformation by proving that the semantic model of the timed automaton after the transformation is bi-simulation equivalent to the original one.

Lemma 2.4.1. *Let $\mathscr{A}'$ be a timed automaton model obtained by applying refinement algorithm into a timed automaton $\mathscr{A}$. Then the semantic models of them are bi-simulation equivalent to each other.*

Proof. For semantic models $\mathscr{T}(\mathscr{A}) = (S, s_0, \Rightarrow)$ and $\mathscr{T}(\mathscr{A}') = (S', s'_0, \Rightarrow')$, we define a relation $R \subseteq S \times S'$ as follows, and prove that R is a bi-simulation relation.

$$R = \{(s, s') \mid s \in S \wedge s' \in S' \wedge ((s = s') \vee (s' \text{ is the duplicate of } s))\}$$

Let us denote a path on the $\mathscr{A}$ corresponding to the spurious counter example used in the refinement step by $\pi = \langle l_0 \stackrel{a_1,g_1,r_1}{\longrightarrow} l_1 \stackrel{a_2,g_2,r_2}{\longrightarrow} \cdots \stackrel{a_n,g_n,r_n}{\longrightarrow} l_n \rangle$, and denote the i-th transition $(l_{i-1}, a_i, g_i, r_i, l_i)$ on π by e_i.

Let m be the minimum positive integer which satisfies $succ^m(\pi) = \emptyset$, and k be the maximum integer within $1 \leq k \leq m$ which satisfies $isRemovable(\pi, k, I) =$

$true$. If there is no such an integer k, we let k be 0. We are going to prove that R is a bi-simulation relation by proving the following properties (i) to $(\mathtt{iv})$.

$(\mathtt{i})$ For all $(s_1, s_1') \in R$ and $a \in A$, if there exists an action transition $s_1 \stackrel{a}{\Rightarrow} s_2$, there exists a corresponding transition $s_1' \stackrel{a}{\Rightarrow}' s_2'$ and $(s_2, s_2') \in R$ holds.

First, we consider the case when s_1 equals s_1' holds.

If $s_1 \stackrel{a}{\Rightarrow}' s_2$ holds, then the property is obviously satisfied.

In the case when $s_1 \stackrel{a}{\Rightarrow}' s_2$ does not hold, (See Fig.2.6) that is when the transition $s_1 \stackrel{a}{\Rightarrow} s_2$ is related to the transition e_k, then $k \neq 0$, $s_1 \in (l_{k-1}, D_{I(l_{k-1})})$, $s_2 \in (l_k, D_{I(l_k)})$ and $a = a_k$ hold. In addition to this, $isRemovable(\pi, k, I)$ is true due to Definition 2.3.11. Also, if $isRemovable(\pi, k, I)$ is true, any successor state from s_1 through e_k is included in (l_k, D_k) due to Definition 2.3.10, and this implies $s_2 \in (l_k, D_k)$. Also, there is a duplicated location corresponding to (l_k, D_k), and therefore, s_2 has its duplication s'. In addition, by the Duplication of Transitions, the transition $s_1 \stackrel{a}{\Rightarrow}' s_2'$ is generated as the duplicate of $s_1 \stackrel{a}{\Rightarrow} s_2$, and $(s_2, s_2') \in R$ holds.

In the case that holds $s_1 \neq s_1'$, i.e. s_1' is the duplicate of s_1, by Lemma 2.3.2 there always exists $s_1' \stackrel{a}{\Rightarrow}' s_2'$ as the duplicate transition of $s_1 \stackrel{a}{\Rightarrow} s_2$. In this case, s_2' is s_2 itself or the duplicate of s_2. Thus, $(s_2, s_2') \in R$ holds.

$(\mathtt{ii})$ For every $(s_1, s_1') \in R$ and $d \in \mathbb{R}_{\geq 0}$, if there exists a delay transition $s_1 \stackrel{d}{\Rightarrow} s_2$, there exists a corresponding transition $s_1' \stackrel{d}{\Rightarrow}' s_2'$ and $(s_2, s_2') \in R$ holds.

In the case $s_1 = s_1'$, it obviously holds.

In the case $s_1 \neq s_1'$, i.e. s_1' is the duplicate of s_1, let s_1 and s_1' be (l, ν) and (l', ν) respectively, which satisfies $l \neq l'$ and $l = parent(l')$. Also, l' is the duplicated location based on a state set (l, D) ($\nu \in D$) which is obtained by the $succ$ operation. According to Lemma 2.3.1, a state set obtained by the $succ$ operation closes under delay transitions. Here, although the invariants on l and l' are different, l' is the duplicated location of l. Therefore, the invariant of l' is stronger than that of l and this implies $D \subset D_{I(l)}$. From this fact, we find that for all ν and d , $(l, \nu) \stackrel{d}{\Rightarrow} (l, \nu + d)$ implies $(l, \nu + d) \in (l, D)$. Similarly for such ν and d, $(l', \nu + d) \in (l', D)$ holds. Therefore, $(l', \nu) \stackrel{d}{\Rightarrow}' (l', \nu + d)$ and $((l, \nu + d), (l', \nu + d)) \in R$ holds.

$(\mathtt{iii})$ For every $(s_1, s_1') \in R$ and $a \in A$, if there exists an action transition $s_1' \stackrel{a}{\Rightarrow}' s_2'$, there exists a corresponding transition $s_1 \stackrel{a}{\Rightarrow} s_2$ and $(s_2, s_2') \in R$ holds.

If $s'_1 \stackrel{a}{\Rightarrow}' s'_2$ is not a duplicated transition, $s'_1 \stackrel{a}{\Rightarrow} s'_2$ holds obviously.

Otherwise, $s'_1 \stackrel{a}{\Rightarrow} s'_2$ does not hold. As implied in Lemma 2.3.2, however, a duplicated transition always has the original one, and therefore, there exists the original transition $s_1 \stackrel{a}{\Rightarrow} s_2$, and $(s_2, s'_2) \in R$.

(iv) For every $(s_1, s'_1) \in R$ and $d \in \mathbb{R}_{\geq 0}$, if there exists a delay transition $s'_1 \stackrel{d}{\Rightarrow}' s'_2$, there exists a corresponding transition $s_1 \stackrel{d}{\Rightarrow} s_2$ and $(s_2, s'_2) \in R$ holds.

This is proved by Lemma 2.3.1 in a similar manner of the proof of (ii).

From the proof of (i), (ii), (iii) and (iv), R is proved to be a bi-simulation relation. Thus, $\mathscr{T}(\mathscr{A}')$ is bi-simulation equivalent to $\mathscr{T}(\mathscr{A})$. □

Next, we show that an abstract model generated by applying our algorithm will be a refined model of the previous one.

Lemma 2.4.2. *Let $\hat{M}'$ be a refined abstract model from $\hat{M}$ by our proposing technique. $\hat{M}'$ is a refined model of $\hat{M}$.*

We prove the lemma by prove that $\hat{M}$ simulates $\hat{M}'$.

Let $\hat{M}$ and $\hat{M}'$ equal $(\hat{S}, \hat{s}_0, \hat{\Rightarrow})$, and $(\hat{S}', \hat{s}'_0, \hat{\Rightarrow}')$, respectively. Then we can define a relation $\hat{R} \subseteq \hat{S}' \times \hat{S}$ as follows and we have only to prove that $\hat{R}$ is a simulation relation.

$$\hat{R} = \{(\hat{s}', \hat{s}) \mid \hat{s}' \in \hat{S}' \wedge \hat{s} \in \hat{S} \wedge (\hat{s}' = \hat{s}) \vee (\hat{s} = parent(\hat{s}'))\}$$

We omit the detailed proof because it is similar to the case (iii) in the proof of Lemma 2.4.1.

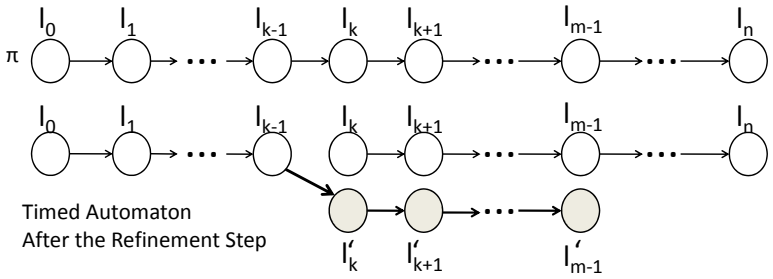
Next, we show that our refinement removes $\hat{\rho}$ for a counter example $\hat{\rho} = \langle \hat{s}_0 \stackrel{a_1}{\rightarrow} \hat{s}_1 \stackrel{a_2}{\rightarrow} \cdots \stackrel{a_n}{\rightarrow} \hat{s}_n \rangle$ on an abstract model $\hat{M}_i$ of i-th iteration.

Here, we show the simple case only. However, other cases are also discussed in a similar way. We assume that path set Π corresponding to $\hat{\rho}$ on timed automaton $\mathscr{A}_i$, has a single element $\pi = \langle l_0 \stackrel{a_1, g_1, r_1}{\longrightarrow} l_1 \stackrel{a_2, g_2, r_2}{\longrightarrow} \cdots \stackrel{a_n, g_n, r_n}{\longrightarrow} l_n \rangle$.

Let integers k and b be the integers defined by Definition 2.3.11. The algorithm generates duplicated locations $l'_k, l'_{k+1}, \cdots l'_{m-1}$ for the location l_k to l_{m-1}, if these locations have not been generated, and also, it removes the transition $(l_{k-1}, a_k, g_k, r_k, l_k)$ as shown in Fig.2.7. It also duplicates related transitions as shown in Fig.2.7.

These transitions are duplicated using Expression (2.4) in Definition 2.3.9. Each transition from l_{j-1} to l_j is duplicated as a transition from l'_{j-1} to l'_j, where $k \leq j \leq m-1$.

For abstract model $\hat{M}_{i+1}$ of a timed automaton $\mathscr{A}_{i+1}$ which is obtained after the i-th iteration, we can reach a state $\hat{s}'_{m-1} (= l'_{m-1})$ through a path $\hat{\rho}$. However, we



The hashed locations, and bold transitions mean duplicated locations and duplicated transitions respectively.

Figure 2.7: An Overview of the Timed Automaton After the Refinement Step with the Path π

cannot reach a state $\hat{s}_m$ even considering duplicated transitions. Thus, the sprious counter example is removed.

Theorem 2.4.2 (Correctness)**.** *Our abstraction refinement algorithm refines abstract models correctly.*

Proof. By Lemma 2.4.1, abstract models before and after the refinement step are bi-simulation equivalent. Also, by Lemma 2.4.2, an original abstract model simulates its refined abstract model. These lemmas imply our abstraction refinement algorithm refines abstract models correctly. □

Theorem 2.4.3 (Termination)**.** *The proposed CEGAR algorithm terminates.*

Proof. The proposed algorithm consists of three operations, Duplication of Location, Duplication of Transition, and Removal of Transition.

For a given finite counter example without loop, at least one of the three operations is executed at each iteration. Therefore, our goal is to show that the numbers of duplicated locations, duplicated transitions, and removal of transition are finite. And also we have to show that the algorithm never repeats the process of regeneration of a transition which is once removed.

Duplication of Location duplicates the location for a given location $l \in L$ and zone $D \in c(C)$. In general, a zone on C is finite[16] under the k-normalization. Therefore, the duplicated location also be finite.

Duplication of Transition duplicates a transition. The number of locations including the duplicated ones is finite. And also the number of application of Dupli-

cation of Transition is limited to the number of locations. Therefore the number of duplicated transition is also finite.

The transition set which will be removed is subset of whole transitions; it is finite. Lemma 2.3.3 states that the algorithm never repeats the process of the regeneration. Thus, the algorithm terminates. □

2.5 Experiment

This section shows experimental results of applying the proposed abstraction to the example of "Gearbox Controller"[58], and also evaluates effectiveness of the abstraction mainly from the view points of space consumption.

2.5.1 goals of the Experiments

In this experiment, we evaluated the performance of our proposed approach with regard to execution time, memory consumption, We compared performance of model checking using our abstraction technique with that of model checking by UPPAAL[5] without any abstraction.

2.5.2 Example

The model of Gearbox Controller is composed of five processes (Gear Controller, Interface, Gearbox, Clutch, and Engine), and has five clock variables. In paper[58], 14 requirement specifications to verify are defined. In this experiment, we verified five specifications of them (specifications (7) to (11) in [58]).

2.5.3 Procedure of the Experiments

Since our abstraction technique cannot accept timed automata described as concurrent processes, we generated a parallel composition of the concurrent processes and applied abstraction to the composition. When we performed model checking without abstraction, we used two types of models. One is the model described as concurrent processes (original model), and the other is the parallel composition of them.

The experiments were performed under Intel Core2 Duo 1.8 GHz, 2GB RAM, and Ubuntu 10.04 (64bit). The version of UPPAAL was 4.0.11.

2.5.4 Results of Experiments

Table 2.1 shows results of experiments. The columns of 'original' and 'composition' represent results of performing model checking to the original model and

Table 2.1: The Experimental Results

	original		composition		abstraction			
spec	time(s)	mem(MB)	time(s)	mem(MB)	time(s)	mem(MB)	loop	dup
(7)	0.10	1.62	0.80	18.92	471.91	10.18	104	538
(8)	0.10	1.62	0.80	35.43	470.48	9.77	96	392
(9)	0.10	1.62	0.80	19.06	514.89	10.05	112	485
(10)	0.10	1.62	0.80	18.90	503.53	9.86	105	435
(11)	0.10	1.62	0.80	46.13	987.63	10.17	175	521

parallel composition of it, respectively, and they are the result without abstraction. Also, the columns of 'abstraction' represent results of applying abstraction. The columns of 'time' represent total execution time (sec), and those of 'mem' represent the maximum memory consumption during model checking (MB). The columns of 'loop', 'dup' represent a count of loop, and a number of duplicated location, respectively. The column of 'spec' means a specification verified in its experiment.

The results show the proposed abstraction can reduce the memory consumption from applying model checking without abstraction in all cases. Especially, For verification of the specification (11), the abstraction reduces the memory consumption about 80 percent.

2.5.5 Discussion

The experimental results show that our model abstraction technique reduces the memory consumption about 50 percent to 80 percent. From this results, we conclude that our abstraction can reduce state space of a timed automaton. On the other hand, the execution time increases drastically. This is because our abstraction requires more than 100 times of iteration to generate the appropriate abstract model. Also, overheads of file I/O cause the increase of execution time since our model checker and CEGAR tool in the prototype send and receive information about abstract models through files for each iteration.

Unfortunately, results of 'original' show better performance than those of 'abstraction.' Comparison of the results of 'original' and 'composition' indicates that parallel composition causes the increase of memory consumption. Therefore, memory consumption of 'abstraction,' which requires a parallel composed model, also increased in this experiment. For a model described as concurrent processes, UPPAAL can use several state space reduction techniques such as the partial order reduction[41] and symmetry reduction[43] techniques. Because of such tech-

niques, the results of 'original' show better performance.

2.5.6 Complexity

Here, the computational complexity and space complexity of our algorithm (Initial Abstraction, Simulation and Refinement) are given.

For $\mathscr{A} = (L, l_0, T, I, C, A)$, let $n = |C|$.

Initial Abstraction From the Algorithm 2.1, the computational complexity is $O(|L| + |T|)$ and the space complexity is also $O(|L| + |T|)$

Simulation Let a length of counter example be l. In the algorithm $Succ$ shown in Algorithm 2.3, operational functions up and and are used. The computational complexity of up and and are $O(n)$ and $O(n^2)$, respectively. Therefore, the computational complexity of $Succ$ is $O(n^2)$. Also, because a new DBM is generated in the algorithm $Reach$, the space complexity is $O(n^2)$. Because the algorithm $Reach$ is called l times in Simulation, its computational complexity and space complexity is $O(l \times n^2)$ and $O(l \times n^2)$, respectively.

Abstraction Refinement The computational complexity of the algorithms $DuplicateState$, $DuplicateTransition$, $RemoveTransition$ is $O(n^3)$, $O(|T|)$, $O(n^2)$. Therefore that of the algorithm Refinement becomes $O(n^3 + |T|)$. Also, the space complexity of these algorithms is $O(n^2)$, $O(|T|)$, and $O(1)$. Thus, that of the algorithm Refinement is $O(n^2 + |T|)$.

2.6 Summary

This study proposes a model abstraction technique for timed automata based on the CEGAR algorithm. In general, most CEGAR based algorithms obtain refined abstract models from the previous abstract models by modifying some transformations. In our algorithm, however, the refined model is obtained indirectly; we transform the original timed automaton preserving the equivalence and from it we generate an abstract model by eliminating clock attributes.

This study gives a formal description and correctness proof of our algorithms.

Chapter 3

Abstraction Refinement for Probabilistic Timed Automata based on CEGAR

3.1 Introduction

This chapter describes an extension of our CEGAR technique described in Chapter 2 into abstraction for probabilistic timed automata.

Probabilistic timed automata are kinds of timed automata extended with probabilistic behavior. This means that the probabilistic timed automata have probabilistic distributions instead of discrete transitions. Therefore, we can extend our abstraction technique into abstraction for probabilistic timed automata by replacing the operations on discrete transitions used in the original algorithm with those on probabilistic distributions.

The abstraction technique abstracts time attributes of probabilistic timed automata by applying our abstraction technique for timed automata. Then, we apply probabilistic model checking to the generated abstract model which is just a Markov decision process (MDP) with no time attributes. The probabilistic model checking algorithm calculates a summation of occurrence probability of all paths which reach to a target state for reachability analysis. For probabilistic timed automata, however, we have to consider required clock constraints for such paths, and choose the paths whose required constraints are compatible. Since our abstract model does not consider the clock constraints, we add a new flow where we check whether all paths used for probability calculation are compatible. Also, if they are not compatible, we transform the model so that we do not accept such incompatible paths simultaneously.

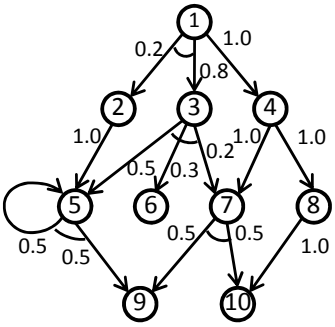


Figure 3.1: An Example of an MDP

In this research, we show the reachability analysis algorithm for probabilistic timed automata, and prove its correctness.

3.2 Preliminary

3.2.1 Probability Distribution

A discrete probability distribution on a finite set Q is given as the function $\mu : Q \to [0, 1]$ such that $\Sigma_{q \in Q} \mu(q) = 1$. Also, $support(\mu)$ is a subset of Q such that $\forall q \in support(\mu).\mu(q) > 0$ holds.

3.2.2 Markov Decision Process

A Markov decision process (MDP)[59] is a Markov chain with non-deterministic choices.

Definition 3.2.1 (Markov Decision Process)**.** *A Markov decision process MDP is 3-tuple $(S, s_0, Steps)$, where*
S : a finite set of states;
$s_0 \in S$: an initial state; and
$Steps \subseteq S \times A \times Dist(S)$: a probabilistic transition relation where $Dist(S)$ is a probability distribution over S.

A set of infinite paths starting from the state s is denoted by $Path_{ful}(s)$. We denote a probability measure over a set Ω of paths by $Prob(\Omega)$.

In our reachability analysis procedure, we transform a given PTA into a finite MDP, and perform probabilistic verification based on the Value Iteration[60] technique.

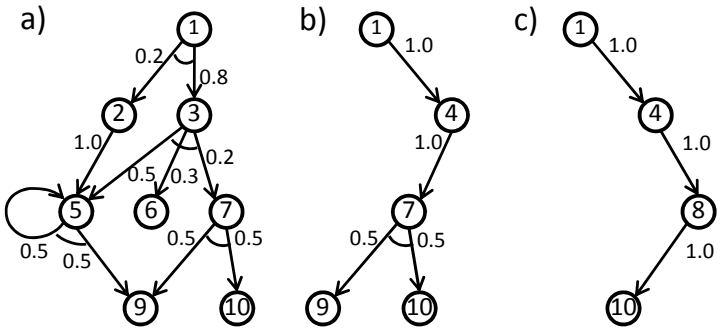


Figure 3.2: Examples of Adversaries

Figure 3.1 shows an example of an MDP. In the figure, probability distributions are associated with transitions. In the figure, transitions which belong to the same distribution are connected with a small arc at their source points. The MDP has several non-deterministic choices at the state 1 and 4. For example, at the state 1, we have two choices; $1)$ the control moves to the state 2 with the probability 0.2 and to the state 3 with the probability 0.8, $2)$ the control moves to the state 4 with the probability 1.0.

Adversary

An MDP has non-deterministic transitions called action. To resolve the non-determinism, an adversary is used. The adversary requires a finite path on an MDP, and decides a transition to be chosen at the next step. For any adversary A and state s, we let $Path^{A}_{ful}(s)$ denote the subset of $Path_{ful}(s)$ induced by A. For a given MDP MDP, we denote the set of all adversaries on MDP by Adb_{MDP}.

Figure 3.2 shows examples of resolving the non-determinism of the MDP shown in Fig.3.1 by some adversaries. Figure 3.2. $a)$ is the case where we choose the action which moves to the state 2 or state 3 at the initial state 1. On the other hand, $b)$ and $c)$ are the cases where we choose the action which moves to the state 4 at the initial state 1. In the case of $b)$, we choose the action which moves to the state 7 when we move from the state 1 to state 4. Also, in the case of $c)$, we choose the action which moves to the state 8 in the same trace.

Here, if we want to obtain the reachability probability from the state 1 to the state 10, under the adversary of $a)$, we can obtain the probability 0.08 ($= 0.8 \times 0.2 \times 0.5$), which is the minimum reachability probability. On the other hand, under the adversary of $c)$, we can obtain the probability 1.0 ($= 1.0 \times 1.0 \times 1.0$), which is the maximum reachability probability.

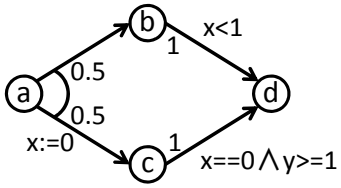


Figure 3.3: An Example of a PTA

Value Iteration

A representative technique of model checking for an MDP is Value Iteration[60]. The Value Iteration technique can obtain both of maximum and minimum probabilities of reachability and safety properties, respectively. At each state, Value Iteration can select an appropriate action according to the property to be checked. Therefore, the technique can produce the adversary as well as the probability.

3.2.3 Probabilistic Timed Automaton

A PTA is a kind of a timed automaton extended with probabilistic behavior. Therefore, using the PTA, we can evaluate quantitative properties such as performance of information systems based on the probabilistic model checking technique. In the PTA, a set of probabilistic distributions is used instead of a set T of discrete transitions on the timed automaton.

Definition 3.2.2 (Probabilistic Timed Automaton)**.** *A probabilistic timed automaton PTA is a 6-tuple $(A, L, l_0, C, I, prob)$, where*
A: a finite set of actions;
L: a finite set of locations;
$l_0 \in L$: an initial location;
C: a finite set of clocks;
$I \subset (L \to c(C))$: a location invariant; and
$prob \subseteq L \times A \times c(C) \times Dist(2^C \times L)$: a finite set of probabilistic transition relations, where $c(C)$ represents a guard condition, and $Dist(2^C \times L)$ represents a finite set of probability distributions p. The Distribution $p(r, l) \in Dist(2^C \times L)$ represents the probability of resetting clock variables in r and also moving to the location l;

Figure 3.3 shows an example of a PTA. In the figure, from the location a, it moves to the location b with the probability 0.5 and also moves to the location c letting the value of the clock x reset to zero with the probability 0.5. Both of the

arcs starting location a are connected with a small arc at their source points, which represents that they belong to the same probability distribution.

Definition 3.2.3 (Transitions of a Probabilistic Timed Automaton)**.** *For $PTA = (A, L, l_0, C, I, prov)$, 6-tuple (l, a, g, p, r, l') represents a transition generated by a probabilistic distribution $(l, a, g, p) \in prob$ such that $p(r, l') > 0$.*

By $l \overset{a,g,p,r}{\rightarrow} l'$, we denote the transition (l, a, g, p, r, l').

Definition 3.2.4 (Semantics of a Probabilistic Timed Automaton)**.** *Semantics of a probabilistic timed automaton $PTA = (A, L, l_0, C, I, prob)$ is given as a timed probabilistic system $TPS_{PTA} = (S, s_0, TSteps)$ where,*

- $S \subseteq L \times \mathbb{R}^C$*;*
- $s_0 = (l_0, 0^C)$*; and*
- $TSteps \subseteq S \times A \cup \mathbb{R}_{\geq 0} \times Dist(S)$ *is composed of action transitions and delay transitions.*

  a) *action transition*
  if $a \in A$ and there exists $(l, a, g, p) \in prob$ such that $g(\nu)$ and $I(l')(r(\nu))$ for all $(r, l') \in support(p)$, $((l, \nu), a, \mu) \in TSteps$ where for all $(l', \nu') \in S$
  $$\mu(l', \nu') = \sum_{r \subseteq C \wedge \nu' = r(\nu)} p(r, l').$$

  b) *delay transition*
  if $d \in \mathbb{R}_{\geq 0}$, and for all $d' \leq d$, $I(l)(\nu + d')$, $((l, \nu), d, \mu) \in TSteps$ where $\mu(l, \nu + d) = 1$.

The concrete delay in the delay transition can be decided non-deterministically on the semantics of a probabilistic timed automaton as well as those of a timed automaton.

In this study, using a location l and a zone D, we describe a set of semantic states as $(l, D) = \{(l, \nu) \mid \nu \in D\}$.

A probabilistic timed automaton is said to be well-formed if a probabilistic edge can be taken whenever it is enabled[20]. Formally, a probabilistic timed automaton $PTA = (A, L, l_0, C, I, prob)$ is well-formed if

$$\forall (l, g, p) \in prob.\ \forall \nu \in \mathbb{R}^C_{\geq 0}.\ (g(\nu)) \ \rightarrow\ \forall (r, l) \in support(p).\ I(l)(r(\nu)).$$

In this study, we assume that a given PTA is well-formed.

Definition 3.2.5 (Path on a Timed Probabilistic System)**.** *A path ω with length of n on a timed probabilistic system $TPS_{PTA} = (S, s_0, TSteps)$ is denoted as follows.*

$$\omega = (l_0, \nu_0) \overset{d_0,\mu_0}{\longrightarrow} (l_1, \nu_1) \overset{d_1,\mu_1}{\longrightarrow} \dots \overset{d_{n-1},\mu_{n-1}}{\longrightarrow} (l_n, \nu_n)$$

, where $(l_0, \nu_0) = s_0$, $(l_i, \nu_i) \in S$ for $0 \leq i \leq n$ and $((l_i, \nu_i), d_i, \mu) \in TSteps \wedge ((l_i, \nu_i + d_i), 0, \mu_i) \in TSteps \wedge (l_{i+1}, \nu_{i+1}) \in support(\mu_i)$ for $0 \leq i \leq n-1$.

For model checking of a probabilistic timed automaton, we extract a number of paths and calculate a summation of their occurrence probabilities in order to check the probability of satisfying a given property. The important point is that we have to choose a set of paths which are compatible with respect to time elapsing.

Lemma 3.2.1 (Compatibility of two paths)**.** *If two paths $\omega^\alpha = (l_0^\alpha, \nu_0^\alpha) \overset{d_0^\alpha,\mu_0^\alpha}{\longrightarrow} (l_1^\alpha, \nu_1^\alpha) \overset{d_1^\alpha,\mu_1^\alpha}{\longrightarrow} \dots \overset{d_{n-1}^\alpha,\mu_{n-1}^\alpha}{\longrightarrow} (l_n^\alpha, \nu_n^\alpha)$ and $\omega^\beta = (l_0^\beta, \nu_0^\beta) \overset{d_0^\beta,\mu_0^\beta}{\longrightarrow} (l_1^\beta, \nu_1^\beta) \overset{d_1^\beta,\mu_1^\beta}{\longrightarrow} \dots \overset{d_{m-1}^\beta,\mu_{m-1}^\beta}{\longrightarrow} (l_m^\beta, \nu_m^\beta)$ on a timed probabilistic system TPS_{PTA} satisfy the following predicate $isCompatible$, then ω^α and ω^β are said to be compatible.*

$$isCompatible(\omega^\alpha, \omega^\beta) = \begin{cases} \textit{true}, & \textit{if } \forall i < min(n,m).\ l_i^\alpha = l_i^\beta \wedge d_i^\alpha = d_i^\beta \\ & \textit{or there exists } i < min(n,m) \textit{ such that} \\ & l_i^\alpha \neq l_i^\beta \wedge d_i^\alpha = d_i^\beta \wedge \\ & \forall j < i.\ l_j^\alpha = l_j^\beta \wedge d_j^\alpha = d_j^\beta \\ \textit{false}, & \textit{otherwise}. \end{cases}$$

Above predicate $isCompatible$ stands for that two paths are compatible if and only if one path is a prefix of the other, or same amount of delay is executed in both paths at the branching point of them.

Lemma 3.2.2 (Compatibility of a set of paths)**.** *If a set Ω of paths on a timed probabilistic system TPS_{PTA} satisfies the following predicate $isCompatible$, then all*

of the paths over Ω are said to be compatible.

$$isCompatible(\Omega) = \begin{cases} \textit{true}, & \textit{if } \forall i \leq min(\Omega) \bigwedge\limits_{\substack{\omega^\alpha, \omega^\beta \in \Omega \\ \wedge \omega^\alpha \neq \omega^\beta}} (l_i^\alpha = l_i^\beta \wedge d_i^\alpha = d_i^\beta) \\ & \quad \textit{or there exists } i \leq min(\Omega) \textit{ such that} \\ & \quad \bigwedge\limits_{\substack{\omega^\alpha \omega^\beta \in \Omega \\ \wedge \omega^\alpha \neq \omega^\beta}} (l_i^\alpha \neq l_i^\beta \wedge d_i^\alpha = d_i^\beta \wedge \bigwedge\limits_{j \leq i} (l_j^\alpha = l_j^\beta \wedge d_j^\alpha = d_j^\beta)), \\ & \quad \textit{and also} \bigwedge\limits_{\substack{\Omega' \in 2^\Omega \wedge \\ \Omega' \neq \Omega \wedge |\Omega'| \leq 2}} isCompatible(\Omega') \\ \textit{false}, & \textit{otherwise}. \end{cases}$$

In Lemma3.2.2, we give the predicate $isCompatible$ for a set Ω of paths on a timed probabilistic system. In the lemma, we let paths in Ω be compatible if there is no contradiction with respect to time elapsing at the branching point of all the paths in Ω, and also if the compatibility is kept for every subset of Ω which contains more than two paths.

Next, we give a simple example of a pair of paths which does not satisfy the compatibility. In Fig.3.3, paths from the location a to d are given as $\omega^\alpha = (a, x = 0 \wedge y = 0) \xrightarrow{0,0.5} (b, x = 0 \wedge y = 0) \xrightarrow{0,1.0} (d, x = 0 \wedge y = 0)$ which reaches to d through b, and $\omega^\beta = (a, x = 0 \wedge y = 0) \xrightarrow{1,0.5} (c, x = 0 \wedge y = 1) \xrightarrow{0,1.0} (d, x = 0 \wedge y = 1)$ which reaches to d through c. In the path ω^α, we are required to let delay at the location a be less than one unit of time because of the guarded condition $x < 1$ of the transition between b and d. On the other hand, in the path ω^β, we are required to let delay at a be grater than or equal to one unit of time because of the condition $x == 0 \wedge y \geq 1$ of the transition between c and d. Like the paths ω^α and ω^β, if the required conditions of time elapsing at the branching point are contradict, we cannot use such paths simultaneously in the probability calculation.

3.2.4 Probabilistic CTL

Definition 3.2.6 (Probabilistic Computational Tree Logic)**.** *Syntax of PCTL formulae is given as follows,*

$$\phi \quad := \quad true \mid false \mid p \mid \neg\phi \mid \phi \vee \phi \mid P_{\sim\lambda}[\phi \, U \, \phi] \mid P_{\sim\lambda}[\phi \, R \, \phi],$$

where p is an arbitrary atomic proposition, $\sim \in \{\leq, <, >, \geq\}$, and $\lambda \in [0, 1]$.

Definition 3.2.7 (Semantics of PCTL). *For a probabilistic transition system $PTS = (S, Steps, L)$ where S is a set of states, $Steps : S \to [0,1]$ is transition relation on S, and $L : S \to 2^{AP}$ is function labeling states with atomic propositions, and for a state $s \in S$, the semantics of PCTL is given as follows.*

$$\begin{aligned} s \models p &\iff p \in L(s) \\ s \models \neg\phi &\iff \textit{not } s \models \phi \\ s \models \phi_1 \vee \phi_2 &\iff s \models \phi_1 \textit{ or } s \models \phi_2 \\ s \models P_{\sim\lambda}\Phi &\iff p_s^A(\Phi) \sim \lambda \textit{ for all } A \in Adv_{PTS}, \end{aligned}$$

where p is an arbitrary atomic proposition, ϕ, ϕ_1 and ϕ_2 are arbitrary PCTL formulae, $p_s^A(\Phi) = Prob\{\omega | \omega \in Path_{full}^A(s) \wedge \omega \models \Phi\}$, and for any path $\omega \in Path_{full}^A(s)$

$$\begin{aligned} \omega \models [\phi_1 \, U \, \phi_2] &\iff \exists i.(\pi^i \models \phi_2 \wedge \forall j.(0 \le j < i \to \pi^j \models \phi_1)) \\ \omega \models [\phi_1 \, R \, \phi_2] &\iff \forall i.(\pi^i \models \phi_2 \vee \exists j.(0 \le j < i \to \pi^j \models \phi_1)), \end{aligned}$$

where $Prob$ represents the probability for a given set of paths and π^i is the suffix of π after the i-th state.

3.3 Proposed Approach

In this section, we will present our abstraction refinement technique for a probabilistic timed automaton. In the technique, we use the abstraction refinement technique for a timed automaton proposed in Chapter 2. Though the probability calculated on the abstract model may be spurious because the abstract model has no time attributes, the finite number of applications of the refinement algorithm enables us to obtain correct results on the abstract model. In addition, we resolve the compatibility problem shown in Sec.3.2.3 by performing a backward simulation technique and generating additional location to distinguish the required condition for every incompatible path. Figure 1.7 shows our abstraction refinement framework. As shown in the figure, we add another flow where we resolve the compatibility problem.

Our abstraction requires a probabilistic timed automaton PTA and a property to be checked as its inputs. The property is limited by the PCTL formula $P_{<p}[\text{true } \mathbf{U}\ err]$. The formula represents a property that the probability of reaching to states where err (which means an error condition in general) is satisfied, is less than p.

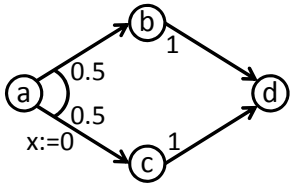


Figure 3.4: An Initial Abstract Model

3.3.1 Initial Abstraction

The initial abstraction removes all the clock attributes from a given probabilistic timed automaton as well as the technique in Chapter 2. The generated abstract model over-approximates the original probabilistic timed automaton. Also, the abstract model is just an MDP without time attributes. We associate information of guard conditions with labels of transitions on the abstract model.

Definition 3.3.1 (Abstract Model)**.** *For a given probabilistic timed automaton* $PTA = (A, L, l_0, C, I, prob)$*, a Markov decision process* $\hat{MDP}_{PTA} = (\hat{S}, \hat{s}_0, \hat{Steps})$ *is produced as its abstract model, where*

- $\hat{S} = L$
- $\hat{s}_0 = l_0$
- $\hat{Steps} = \{\ (s, a\_g, p) \mid (s, a, g, p) \in prob\ \}$

Figure 3.4 shows an initial abstract model for the PTA shown in Fig.3.3 As shown in the figure, the abstract model is just an MDP where all of the clock constraints are removed though we keep a set of clock reset as a label of transitions.

3.3.2 Model Checking

In model checking, we apply Value Iteration[60] into the markov decision process obtained by abstraction and calculate a maximum reachability probability. Also, it decides an action to be chosen at every state as an adversary. If the obtained probability is less than p, we can terminate the CEGAR loop and conclude that the property is satisfied.

Although Value Iteration can calculate a maximum reachability probability, it cannot produce concrete paths used for the probability calculation. To obtain the concrete paths, we use an approach proposed in Paper[61] which can produce

counter example paths for PCTL formulas. The approach translates a probabilistic automaton into a weighted digraph. And we can obtain at most k paths by performing k-shortest paths search on the graph.

Definition 3.3.2 (Path on the Abstract Model)**.** *A path $\hat{\rho}$ on an abstract model $\hat{MDP}_{PTA} = (\hat{S}, \hat{s}_0, \hat{Steps})$ for $PTA = (A, L, l_0, C, I, prob)$ is given as follows,*

$$\hat{\rho} = \hat{s}_0 \xrightarrow{a_0\_g_0, p_0, r_0} \hat{s}_1 \xrightarrow{a_1\_g_1, p_1, r_1} \cdots \xrightarrow{a_{n-1}\_g_{n-1}, p_{n-1}, r_{n-1}} \hat{s}_n$$

, where $\hat{s}_i \in \hat{S}$ for $0 \leq i \leq n$ and $(\hat{s}_i, a_i\_g_i, p_i) \in \hat{Steps} \wedge (r_i, \hat{s}_{i+1}) \in support(p_i)$ for $0 \leq i \leq n-1$.

As defined in Def. 3.3.2, we associate a set r of clock reset with a path on an abstract model in order to show the difference of r over the probabilistic distribution p.

For the abstract model shown in Fig.3.4, Value Iteration outputs 1.0 as the probability that it reaches to the location d from the location a. On the other hand, k-shortest paths search ($k \geq 2$) detects two paths $\hat{\rho}^{\alpha} = a \xrightarrow{\tau, 0.5, \{\}} b \xrightarrow{\tau, 1.0, \{\}} d$ and $\hat{\rho}^{\beta} = a \xrightarrow{\tau, 0.5, \{x:=0\}} c \xrightarrow{\tau, 1.0, \{\}} d$, where τ represents a label for transitions with no label in the figure.

3.3.3 Simulation

Simulation checks whether all the paths obtained by k-shortest paths search are feasible or not on the original probabilistic timed automaton. For the path $\hat{\rho} = \hat{s}_0 \xrightarrow{a_0\_g_0, p_0, r_0} \hat{s}_1 \xrightarrow{a_1\_g_1, p_1, r_1} \cdots \xrightarrow{a_{n-1}\_g_{n-1}, p_{n-1}, r_{n-1}} \hat{s}_n$ on the abstract model, the corresponding path on the original probabilistic timed automaton is given as $\rho = l_0 \xrightarrow{a_0, g_0, p_0, r_0} l_1 \xrightarrow{a_1, g_1, p_1, r_1} \cdots \xrightarrow{a_{n-1}, g_{n-1}, p_{n-1}, r_{n-1}} l_n$, where $l_i = \hat{s}_i$. Since the path on the abstract model has information of guard and reset clocks as its label, we can decide the corresponding path with one-to-one relation.

We use the simulation algorithm shown in Chapter 2 where we use some operations of DBM (Difference Bound Matrix)[16] to obtain zones which are reachable from the initial state. If there is at least one path which is infeasible on the original PTA, we proceed to the abstraction refinement step.

Figure 3.5 shows the simulation results for two paths $\hat{\rho}^{\alpha}$ and $\hat{\rho}^{\beta}$. Simulation concludes that the two paths are feasible on the original PTA.

3.3.4 Abstraction Refinement

In this step, we refine the abstract model so that the given spurious counter example also becomes infeasible on the refined abstract model. We can use the

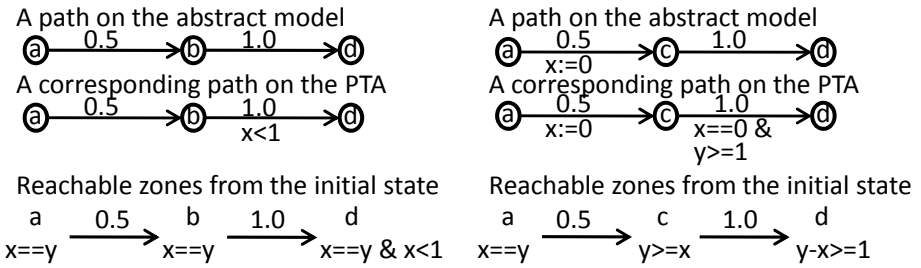


Figure 3.5: Simulation Results for a Set of Paths

algorithm shown in Chapter 2. Since the refinement algorithm performs some operations on transitions of a timed automaton, we replace such operations by those on probability distributions of a probabilistic timed automaton.

3.3.5 Compatibility Checking

When all the paths obtained by k-shortest paths search are feasible and a summation of occurrence probabilities of them is greater than p, we also have to check whether all the paths are compatible or not.

First, we define a predecessor state set by one discrete transition and one delay transition.

Definition 3.3.3 (Predecessor State Set)**.** *Given a state set (l_2, D_2) on TPS_{PTA} for a probabilistic timed automaton PTA and a transition $e = (l_1, a, g, r, l_2)$, a predecessor state set $pred((l_2, D_2), e)$ from (l_2, D_2) through the transition e is defined as follows;*

$$pred((l_2, D_2), e) = \{(l_1, \nu) \mid (r(\nu) + d) \in D_2 \wedge d \in \mathbb{R}_{\geq 0} \\ \wedge (l_1, \nu) \stackrel{a}{\Rightarrow} (l_2, r(\nu)) \wedge (l_2, r(\nu)) \stackrel{d}{\Rightarrow} (l_2, r(\nu) + d) \}.$$

For a given state set and a transition, Algorithm 3.1 computes $pred$. As well as the $succ$ operation, we denote k-th reachable state set from the last state set $(l_n, D_{I(l_n)})$ of the path ρ by $pred^k(\rho)$.

Definition 3.3.4 (Reachable State Set)**.** *Given a path $\rho = l_0 \stackrel{a_0,g_0,p_0,r_0}{\rightarrow} l_1 \stackrel{a_1,g_1,p_1,r_1}{\rightarrow} \ldots \stackrel{a_{n-1},g_{n-1},p_{n-1},r_{n-1}}{\rightarrow} l_n$ with length n on a probabilistic timed automaton PTA, a k-th reachable state set ($k \leq n$) which is reachable from the initial state and also which can reach to l_n is defined as follows.*

$$reachable(\rho, k) = succ(\rho, k) \cap pred(\rho, n - k).$$

Algorithm 3.1 Pred(PTA, R, e)

```
1: /* R = (l_2, D_2), e = (l_1, a, g, p, r, l_2) */
2: D_1 := D_2
3: D_1 := down(D_1)                          /* reverse the time elapse */
4: D_1 := and(D_1, I(l_1))
5: D_1 := free(D_1, r)                       /* remove all constraints on r */
6: D_1 := and(D_1, g)
7: D_1 := and(D_1, I(l_1))
8: return (l_1, D_1)
```

Here, we define the conditions using zone notations to satisfy compatibility on a given set of paths. Compatibility Checking is based on Lemma 3.2.2.

Lemma 3.3.1 (Compatibility Check)**.** *For a given set P of paths on a probabilistic timed automaton PTA, all of the paths over P are said to be compatible if the following property holds;*

> *For the maximum integer $i \geq 0$ such that all the paths in P share the same prefix with the sequence of i transitions,*
>
> $$\bigcap_{\rho \in P} reachable(\rho, i) \neq \emptyset$$
>
> *and, for all $P' \in 2^P$ where $|P'| \geq 2$ and all the paths in P' share the same prefix with the sequence of at least $i+1$ transitions, P' is said to be compatible recursively.*

In the compatibility checking, at each location of the paths, we have to obtain the reachable state set $reach$. Next, we check the compatibility according to Lemma 3.3.1. To obtain the state set $reach$, we have to perform both forward simulation shown in Sec. 3.3.3 and backward simulation for each path, and merge the results. For the result of forward simulation, we can reuse the result obtained in the Simulation step. Then we check the compatibility based on Lemma 3.3.1.

Backward Simulation

In the backward simulation, we calculate a reachable state set along with a given path backwardly. Algorithm 3.1 calculates the predecessor state set by one discrete transition. Functions and, $free$, $down$ used in the algorithm are operation functions on a zone, and are defined in Paper[16]. Formally, for a zone D, a constraint c, and a set r of clock reset, those functions are defined as follows;

Algorithm 3.2 BackwardSimulation(PTA, ρ)

1: /* $PTA = (A, L, l_0, C, I, prob)$
$\rho = l_0 \overset{a_0,g_0,p_0,r_0}{\rightarrow} l_1 \overset{a_1,g_1,p_1,r_1}{\rightarrow} \cdots \overset{a_{n-1},g_{n-1},p_{n-1},r_{n-1}}{\rightarrow} l_n$ */
2: $D^{\rho}_{b,n} := I(\hat{l}_n)$
3: **for** $i := n - 1$ downto 0 **do**
4: $\quad D^{\rho}_{b,i} := Pred(PTA, D^{\rho}_{b,i+1}, e_i)$ /* $e_i = (l_i, a_i, g_i, p_i, r_i, l_{i+1})$ */
5: **end for**
6: **return** D^{ρ}_b

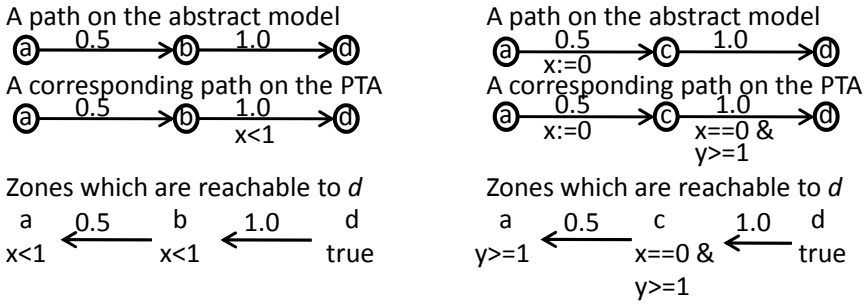


Figure 3.6: Results of Backward Simulation for a Set of Paths

$and(D, c) = \{u \mid u \in D \wedge u \in c\}$, $free(D, r) = \{u \mid r(u) \in D\}$, and $down(D) = \{u \mid u + d \in D \wedge d \in \mathbb{R}_{\geq 0}\}$

Algorithm 3.2 implements the backward simulation.

Figure 3.6 shows results of backward simulation for two paths $\hat{\rho}^{\alpha}$ and $\hat{\rho}^{\beta}$ detected in Sec. 3.3.2.

Determination of Compatibility

In this step, we check compatibility of the set P of paths on the abstract model using the required conditions obtained by both of forward and backward simulation. Algorithm 3.3 checks the compatibility of P using the Algorithm 3.4.

Algorithm 3.4 first checks whether the required conditions of the i-th locations for each path are compatible or not ($l2$-$l8$) using the results of forward and

Algorithm 3.3 IsCompatible(PTA, P, D_f, D_b)

1: /* $PTA = (A, L, l_0, C, I, prob)$, $\hat{P}$ is a set of abstract paths, and D_f and D_b are sets of forward and backward simulation results for each path $\rho \in P$,respectively. */
2: **return** $CompatibleCheck(PTA, P, D_f, D_b, 0)$

Algorithm 3.4 CompatibleCheck(PTA, P, D_f, D_b, i)

1: $D' := true$
2: **foreach** $\rho \in P$ such that $length(\rho) \geq i$ **do**
3: $D^{\rho}_{c,i} := D^{\rho}_{f,i} \cap D^{\rho}_{b,i}$ /* calculate the $reach(\rho, i)$ */
4: $D' := D' \cap D^{\rho}_{c,i}$
5: **if** $D' = \emptyset$ **then**
6: **return** $false$
7: **end if**
8: **end for**
9: S^{P}_{i+1} := $SplitPathSet(P, i+1)$
10: /* split P into a set of its subsets without overlap with respect to the $i+1$-th location and clock reset for each path in P */
11: **foreach** $P' \in S^{P}_{i+1}$ such that $|P'| \geq 2$ **do**
12: **if** $CompatibleCheck(PTA, P', D, i+1)$=$false$ **then**
13: **return** $false$
14: **end if**
15: **end for**
16: **return** $true$

Algorithm 3.5 SplitPathSet(P, i)

1: $S := \emptyset$
2: **foreach** $\rho \in P$ **do**
3: /* $\rho = l_0 \overset{a_0,g_0,p_0,r_0}{\rightarrow} l_1 \overset{a_1,g_1,p_1,r_1}{\rightarrow} \cdots \overset{a_{n-1},g_{n-1},p_{n-1},r_{n-1}}{\rightarrow} l_n$ */
4: **if** $P_{r_{i-1},l_i} \notin S$ **then**
5: $P_{r_{i-1},l_i} := \{\rho\}$
6: $S := S \cup P_{r_{i-1},l_i}$
7: **else**
8: $l_{r_{i-1},l_i} := P_{r_{i-1},l_i} \cup \{rho\}$
9: **end if**
10: **end for**
11: **return** S

backward simulation. Next, the algorithm divides P into its subsets based on the $(i+1)$-th locations and the set of clock reset for each path ($l9$). Then, it checks the compatibility for the following sequences of paths by applying the algorithm into the divided subsets recursively ($l11$-$l15$). Although the predicate $isCompatible$ in the Lemma 3.2.2 checks the compatibility for each subset of P, the algorithm omit redundant checks by dividing P based on the branches of the paths.

For the path ρ^{α} in Sec. 3.3.2, zones at a which is reachable from initial state and which can move to d are given as $D^{\hat{\rho}^{\alpha}}_{f,0} = (x == y)$, and $D^{\hat{\rho}^{\alpha}}_{b,0} = (x < 1)$, respectively. Also, a zone of the product of them is given as $D^{\hat{\rho}^{\alpha}}_{c,0} = (x == y \wedge x <$

1). Similarly, for the path $\hat{\rho}^{\beta}$, the product zone is given as $D_{c,0}^{\hat{\rho}^{\beta}} = (x == y \wedge y > 1)$. Since $D_{c,0}^{\hat{\rho}^{\alpha}}$ and $D_{c,0}^{\hat{\rho}^{\beta}}$ contradict each other, we can conclude that the paths $\hat{\rho}^{\alpha}$ and $\hat{\rho}^{\beta}$ are incompatible each other.

3.3.6 Model Transformation

When the compatibility check algorithm finds that a given set P of paths is incompatible at i-th location, our proposed algorithm resolves the incompatibility by refining behaviors from the i-th location. Our algorithm uses $reachable(\rho, i)$ defined in Definition 3.3.4, which is a product of results of forward and backward simulation for a path $\hat{\rho} \in \hat{P}$. It duplicates locations which are reachable from the zone of $reach(\rho, i)$ by an action associated with the i-th distribution p_i. The compatibility checking algorithm checks compatibility for some subsets of P. Therefore, if such subsets are incompatible, we apply model transformation based on the subsets.

Definition 3.3.5 (Duplicated Locations to Resolve Incompatibility)**.** *For a given (sub)set P of paths which is incompatible at i-th location, (i+1)-th distribution (l_i, a_i, g_i, p_i) , and a probabilistic timed automaton $PTA = (A, L, l_0, C, I, prob)$, we generate the following sets L_d and L_{comp} of duplicated locations;*

$$\begin{aligned} L_d &= \{l_d \mid I(l_{dup}) = succ(reachable(\rho, i), e_{i+1}) \ \wedge \\ &\quad \rho \in P \ \wedge (l, r) \in L \times 2^C \ \wedge \ p_i(l, r) > 0 \ \wedge \ e_{i+1} = (l_i, a_i, g_i, p_i, r, l)\} \\ L_{comp} &= \{l_{comp} \mid I(l_{comp}) = succ(\bigwedge_{\rho \in P} \neg reachable(\rho, i), e_{i+1}) \ \wedge \\ &\quad (l, r) \in L \times 2^C \ \wedge \ p_i(l, r) > 0 \ \wedge \ e_{i+1} = (l_i, a_i, g_i, p_i, r, l) \ \} \end{aligned}$$

Also the algorithm constructs transition relations so that the transformation becomes equivalent transformation. For example, transition relations from a duplicated location are duplicated if the relations are executable from the invariant associated with the duplicated location.

Definition 3.3.6 (Duplicated Distributions to Resolve Incompatibility)**.** *For a given (sub)set P of paths which is incompatible at i-th location, (i+1)-th distribution (l_i, a_i, g_i, p_i), a probabilistic timed automaton $PTA = (A, L, l_0, C, I, prob)$, and a set L_d of duplicated locations, we generate the following distributions;*

- $(l_i, a_i, g_i, p_{d,i})$ *for each* $\rho \in P$*:*
  a duplicate of $(l_i, a_i, g_i, p_i) \in prob$ *where the following property holds.*
  For all $(l, r) \in L \times 2^C$ *such that* $p_i(l, r) > 0$*, there exists the duplicate* l_d *of* l *where* $I(l_d) = succ(reach(\rho, i), (l_i, a_i, g_i, p_i, r, l))$ *holds, and* $p_{d,i}(l_d, r) = p_i(l, r)$ *holds.*

Algorithm 3.6 TransformPTA(PTA, D_c, P, i)

1: $D_{complement} :=$ true
2: **foreach** $\rho \in P$ **do**
3: $L_{dup} := DuplicateLocation(PTA, \rho, D^{\rho}_{c,i}, i)$
4: $L := L \cup L_{dup}$
5: $prob_{dup} := DuplicateDistribution(PTA, \rho, L_{dup}, i)$
6: $prob := prob \cup prob_{dup}$
7: $D_{complement} := D_{complement} \cap D^{\rho}_{c,i}$
8: **end for**
9: $L_{dup} := DuplicateLocation(PTA, \rho, D_{complement}, i)$
10: $L := L \cup L_{dup}$
11: $prob_{dup} := DuplicateDistribution(PTA, \rho, L_{dup}, i)$
12: $prob := prob \cup prob_{dup}$
13: $prob := RemoveDistribution(PTA, l_i, a_i, g_i, p_i)$
14: /* All the paths in P share the same (i+1)-th distribution (l_i, a_i, g_i, p_i) */
15: **return** PTA

- $(l_i, a_i, g_i, p_{comp,i})$ *:*
  a duplicate of $(l_i, a_i, g_i, p_i) \in prob$ *where the following property holds. For all* $(l, r) \in L \times 2^C$ *such that* $p_i(l, r) > 0$*, there exists the duplicate* l_{comp} *of* l *where* $I(l_{comp}) = succ(\bigwedge_{\rho \in P} \neg reachable(\rho, i), (l_i, a_i, g_i, p_i, r, l))$ *holds, and* $p_{comp,i}(l_d, r) = p_i(l, r)$ *holds.*

- (l_d, a, g, p_d) *for each* $l_d \in L_d$
  duplicated distributions from all of the duplicated locations l_d *such that* l_d *is the duplicate of* l *and* $(l, a, g, p) \in prob$*. Also, for all* $(l', r) \in L \times 2^C$ $succ((l_d, D_{I(l_d)}), (l, a, g, p, r, l')) \neq \emptyset$ *and* $p_d(l, r) = p(l, r)$ *hold.*

Definition 3.3.7 (A Removed Distribution to Resolve Incompatibility)**.** *For a given (sub)set* P *of paths which is incompatible at* i*-th location, and a probabilistic timed automaton* $PTA = (A, L, l_0, C, I, prob)$*, we remove the distribution* (l_i, a_i, g_i, p_i) *from* $prob$*.*

Algorithm 3.6 transforms a given PTA with considering its compatibility. The algorithm calls $DuplicateLocation$ (Algorithm 3.7) which duplicates locations, $DuplicateDistribution$ (Algorithm 3.8) which duplicates probabilistic transitions, and $RemoveDistribution$ (Algorithm 3.10) which removes probabilistic transitions. In Algorithms 3.7 and 3.9, the procedure $Succ$ (Algorithm 2.3) is used.

Figure 3.7 shows the transformed PTA by applying the model transformation procedure for the paths ρ^{α} and ρ^{β}. The locations b^1 and c^1 are duplicated locations based on the path ρ^{α} and the zone $D^{\rho^{\alpha}}_{c,0} = (x == y \wedge x < 1)$ on the location a. We associate invariants to b^1 and c^1 based on zones which are reachable from $D^{\rho^{\alpha}}_{c,0}$

Algorithm 3.7 DuplicateLocation(PTA, ρ, D, i)

1: /* $PTA = (A, L, l_0, C, I, prob)$
$\rho = l_0 \overset{a_0,g_0,p_0,r_0}{\rightarrow} l_1 \overset{a_1,g_1,p_1,r_1}{\rightarrow} \cdots \overset{a_{n-1},g_{n-1},p_{n-1},r_{n-1}}{\rightarrow} l_n$ */
2: $L_{dup} := \emptyset$
3: **foreach** $(l, r) \in L \times 2^C$ such that $p_i(l, r) > 0$ **do**
4: $(l, D) := Succ(PTA, (l_i, D), e)$
5: /* $Succ$ is implemented in Algorithm 2.3, whose semantics is given in Definition 3.3.4 */
6: $l_{dup} := newLocation()$
7: $I(l_{dup}) := D$
8: $L_{dup} := L_{dup} \cup \{l_{dup}\}$
9: **end for**
10: **return** L_{dup}

Algorithm 3.8 DuplicateDistribution(PTA, ρ, L_{dup}, i)

1: /* $PTA = (A, L, l_0, C, I, prob)$
$\rho = l_0 \overset{a_0,g_0,p_0,r_0}{\rightarrow} l_1 \overset{a_1,g_1,p_1,r_1}{\rightarrow} \cdots \overset{a_{n-1},g_{n-1},p_{n-1},r_{n-1}}{\rightarrow} l_n$ */
2: $prob_{dup} := \emptyset$
3: $p_{dup} := newDistribution()$ /* generate a new distribution over $L \times 2^C$ */
4: **foreach** $(l, r) \in L \times 2^C$ **do**
5: **if** $p_i(l, r) > 0$ **then**
6: $p_{dup}(l_{dup}, r) := p_i(l, r)$
7: /* l_{dup} is a duplicate location of l generated by DuplicateLocation algorithm */
8: **else**
9: $p_{dup}(l, r) := 0$
10: **end if**
11: **end for**
12: $prob_{dup} := Prob_{dup} \cup \{(l_i, a_i, g_i, p_{dup})\}$
13: **foreach** $l_{dup} \in L_{dup}$ **do**
14: $prob_{dup} := Prob_{dup} \cup DuplicateDistFromDupLoc(PTA, l_{dup})$
15: **end for**
16: **return** $prob_{dup}$

Algorithm 3.9 DuplicateDistFromDupLoc(PTA, l_{dup})

1: /* $PTA = (A, L, l_0, C, I, prob)$, and let l be an original location of l_{dup} */
2: $prob_{dup} := \emptyset$
3: **foreach** $(l, a, g, p) \in Prob$ **do**
4: f_{dup} := true, $p_{dup} := newDistribution()$
5: **foreach** $(l', r) \in L \times 2^C$ **do**
6: **if** $Succ((l, I(l_{dup})), e) \neq \emptyset$ **then**
7: $p_{dup}(l', r) = p(l, r)$ /* $e = (l, a, g, p, r, l')$ */
8: **else**
9: $f_{dup} :=$ false
10: **break**
11: **end if**
12: **end for**
13: **if** f_{dup} **then**
14: /* duplicate the distribution if it is executable from the duplicate location */
15: $prob_{dup} := prob_{dup} \cup \{(l, a, g, p_{dup})\}$
16: **end if**
17: **end for**

Algorithm 3.10 RemoveDistribution(PTA, l_i, a_i, g_i, p_i)

1: $prob := prob \setminus \{(l_i, a_i, g_i, p_i)\}$
2: **return** $prob$

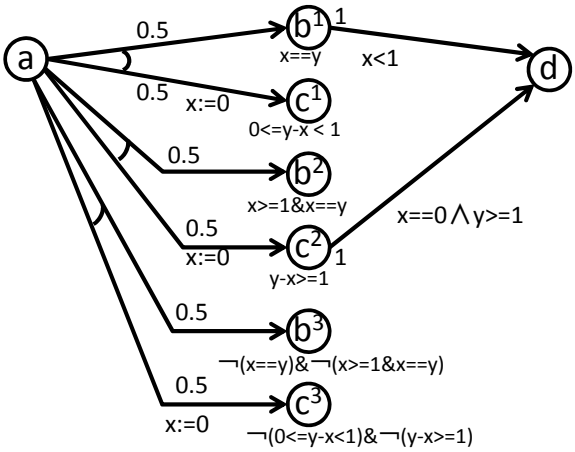


Figure 3.7: The Transformed PTA

through transitions from a to b, and from a to c, respectively. Also, we duplicate a transition from b to d as the transition from b^1 to d because the transition is feasible from the invariant of b^1. On the other hand, we do not duplicate a transition from c to d because the transition is not feasible from the invariant of c^1. Similarly, locations b^2 and c^2 are duplicated locations based on the path ρ^β and the zone $D^{\rho^\beta}_{c,0}$. Locations b^3 and c^3 are generated as complements of the invariant associated with each duplicated location in order to preserve the equivalence.

By transforming the original PTA in such a way, if we remove all clock constraints from the model in Fig.3.7, Value Iteration on its abstract model outputs 0.5 as the maximum probability.

3.4 Correctness Proof

In this section, we prove correctness of our algorithm. To prove it, we have to show correctness of abstraction and correctness of model transformation to solve the incompatibility. The correctness of abstraction is already proved in Chapter 2.

Lemma 3.4.1. *Let PTA' be a probabilistic timed automaton obtained by applying model transformation into a probabilistic timed automaton PTA. Then the semantic models of them are bi-simulation equivalent to each other.*

Proof. Let $TPS_{PTA} = (S, s_0, TSteps)$ and $TPS_{PTA'}(S', s'_0, TSteps')$ be semantic models of PTA and PTA', respectively. Here, we define a relation R :

$S \times S'$ as follows, and prove that R is a bi-simulation relation.

$$R = \{(s, s') \mid s \in S \wedge s' \in S' \wedge ((s = s') \vee (s' \text{ is the duplicate of } s))\}$$

We are going to prove that R is a bi-simulation relation by proving the following properties (i) to $(\mathtt{iv})$.

$(\mathtt{i})$ For all $(s_1, s_1') \in R$ and $a \in A$, if there exists an action transition (s_1, a, μ), there exists a corresponding transition (s_1', a, μ'), and for all $s_2 \in S$ there exists $s_2' \in S'$ such that $\mu(s_2) = \mu'(s_2')$ and $(s_2, s_2') \in R$ hold.

First, we consider the case when $s_1 = s_1'$ holds.

If $(s_1, a, \mu) \in TSteps'$ holds, then the property is obviously satisfied.

In the case when $(s_1, a, \mu) \notin TSteps'$ holds, then the transition (s_1, a, μ) is removed by the Algorithm 3.10. As defined in Definition 3.3.6 and Definition 3.3.7, the removed distribution is duplicated. Therefore, Algorithm 3.8 produces the distribution (s_1, a, μ'). where $\mu'(l, r) = \mu(l, r)$ for all $(l, r) \in L \times 2^C$ such that l does not have its duplication, and for all $(l, r) \in L \times 2^C$ such that l has its duplication l_{dup}, $\mu'(l_{dup}, r) = \mu(l, r)$ and $\mu'(l, r) = 0$ hold. Since l_{dup} is the duplicate of l, $(l, l_{dup}) \in R$ holds. Therefore, the property is satisfied.

Next, we consider the case $s_1 \neq s_1'$. If we let s_1 and s_1' be (l_1, ν) and (l_1', ν), respectively, obviously, l_1' is the duplicated location of l_1 generated by Algorithm 3.7. As defined in Definition 3.3.6, we duplicate distributions from the duplicated location if they are enable. Therefore, there exists a corresponding transition $((l_1', \nu), a, \mu')$ where $\mu'(l, r) = \mu(l, r)$ for all $(l, r) \in L \times 2^C$.

$(\mathtt{ii})$ For every $(s_1, s_1') \in R$ and $d \in \mathbb{R}_{\geq 0}$, if there exists a delay transition (s_1, d, μ), there exists a corresponding transition (s_1', d, μ') and $(s_2, s_2') \in R$ holds where $\mu(s_2) = 1$ and $\mu'(s_2') = 1$ hold.

In the case $s_1 = s_1'$, it obviously holds.

Next, we consider the case $s_1 \neq s_1'$. Let s_1 and s_1' be (l, ν) and (l', ν) respectively. Obviously, l' is the duplicated location of l generated by Algorithm 3.7. As defined in Definition 3.3.5, the duplicated location l' is based on $succ$ operation defined in Definition 2.3.4. According to Lemma 2.3.1, a state set obtained by the $succ$ operation closes under delay transitions. Therefore, if there exists a delay transition $((l, \nu), d, \mu) \in TSteps$ where $\mu(l, \nu + d) = 1$, there also exists a corresponding delay transition $((l_1', \nu), d, \mu') \in TSteps'$ where $\mu'(l', \nu + d) = 1$. Since l' is the duplicate of l, $((l, \nu + d), (l', \nu + d)) \in R$ is satisfied.

$(\mathtt{iii})$ For every $(s_1, s_1') \in R$ and $a \in A$, if there exists an action transition

(s'_1, a, μ'), there exists a corresponding transition (s_1, a, μ), and for all $s'_2 \in S$ there exists $s_2 \in S$ such that $\mu'(s_2) = \mu(s'_2)$ and $(s_2, s'_2) \in R$ hold.

If (s'_1, a, μ') is a transition which is not generated by Algorithm 3.8, $(s'_1, a, \mu') \in TSteps$ holds obviously.

Otherwise, $s'_1 \stackrel{a}{\Rightarrow} s'_2$ does not hold. As implied in Lemma 2.3.2, however, a duplicated transition always has the original one, and therefore, there exists the original transition $s_1 \stackrel{a}{\Rightarrow} s_2$, and $(s_2, s'_2) \in R$.

(iv) For every $(s_1, s'_1) \in R$ and $d \in \mathbb{R}_{\geq 0}$, if there exists a delay transition (s'_1, d, μ'), there exists a corresponding transition (s_1, d, μ) and $(s_2, s'_2) \in R$ holds where $\mu'(s'_2) = 1$ and $\mu(s_2) = 1$ hold.

This is proved by Lemma 2.3.1 in a similar manner of the proof of (ii).

From the proof of (i), (ii), (iii) and (iv), R is proved to be a bi-simulation relation. Thus, $TPS_{PTA'}$ is bi-simulation equivalent to TPS_{PTA}. □

3.5 Experiments

We have implemented a prototype of our proposed approach with Java, and performed some experiments. Though the prototype can check the compatibility of a given set of paths, currently it cannot deal with the model transformation.

The prototype performs k-shortest paths search and simulation concurrently in order to reduce execution time. By implementing the algorithms concurrently, we have not to wait until all of k paths are detected, i.e. if a path is detected by the k-shortest paths search algorithm, we can immediately apply simulation and (if needed) abstraction refinement procedures.

Also, our prototype continues the k-shortest search algorithm when a spurious counter example is detected and the refinement algorithm is applied. If other paths which do not overlap with the previous spurious counter examples are detected, we can apply simulation and refinement algorithms to it again. This helps us reduce the number of CEGAR loop.

3.5.1 Goals of the Experiments

In this experiment, we evaluated the performance of our proposed approach with regard to execution time, memory consumption, and qualities of obtained results. As a target for comparison, we chose the approach of Digital Clocks[45] where they approximate clock evaluations of a PTA by integer values.

Table 3.1: The Experimental Results

		Digital Clocks[45]				Proposed Approach				
D	p	$Result$	$Time$	$State$	MEM	$Result$	$Time$	$Loop$	$State$	$Heap$
5μs	1.09×10^{-1}	false	20.90s	297,232	10.2MB	false	4.19s	10	37	8.0MB
	3.28×10^{-1}	true	20.89s	297,232	10.2MB	true	3.60s	9	36	8.0MB
10μs	1.26×10^{-2}	false	54.80s	685,232	21.7MB	false	8.16s	19	134	8.0MB
	3.79×10^{-2}	true	54.82s	685,232	21.7MB	true	6.57s	15	115	8.0MB
20μs	1.85×10^{-4}	false	176.93s	1,461,232	41.0MB	false	1186.08s	47	477	64.0MB
	5.56×10^{-4}	true	177.46s	1,461,232	41.0MB	true	31.32s	32	435	8.0MB

3.5.2 Example

We used a case study of the FireWire Root Contention Protocol[29] as an example for this experiment. This case study concerns the Tree Identify Protocol of the IEEE 1394 High Performance Serial Bus (called "FireWire") which takes place when a node is added or removed from the network. In the experiment, we checked the probability that a leader is not selected within a given deadline. The probabilistic timed automaton for the example is shown in Fig.1.3, which is composed of two clock variables, 10 locations, and 18 transitions.

3.5.3 Procedure of the Experiments

In this experiment, we checked the property that "the probability that a leader cannot be elected within a given $deadline$ is less than p." We considered three scenarios where the parameter $deadline$ is 5, 10, 20 μs, respectively. Also, for each scenario, we conducted two experiments where the value of p is 1.5 times as an approximate value of the maximum probability obtained by the Digital Clocks approach[45] and a half of it, respectively. In the proposed approach, we searched at most 5000 paths by letting the parameter k of the k-shortest paths search algorithm be 5000. For evaluation of existing approach, we used the probabilistic model checker PRISM[6].

The experiments were performed under Intel Core2 Duo 2.33 GHz, 2GB RAM, and Fedora 12 (64bit).

3.5.4 Results of the Experiments

The results are shown in Table 3.1. The column of D means the value of $deadline$. For each approach, columns of $Results$, $Time$, and $States$ show the results of model checking, execution time of whole process, and the number of states constructed, respectively. The column MEM in the columns of the Digital Clocks shows the memory consumption of PRISM. The columns $Loop$ and

Table 3.2: Analysis of Counter Example Paths

D	p	$Path$	$Probability$	CC
5μs	1.0938×10^{-1}	7	1.2500×10^{-1}	0.7ms
10μs	1.2635×10^{-2}	43	1.2695×10^{-2}	5.9ms
20μs	1.8500×10^{-4}	2534	1.8501×10^{-4}	296.9ms

$Heap$ in the columns of the proposed approach show the number of CEGAR loops executed and the maximum heap size of the Java Virtual Machine (JVM) which executes our prototype, respectively.

Table 3.1 shows that for all cases we can dramatically reduce the number of states and obtain correct results. Moreover, we can reduce the execution time more than 80 percent except for the case when $deadline = 20\mu s$ and $p = 1.85 \times 10^{-4}$. In this case, however, the execution time drastically increases.

Table 3.2 shows the results of analysis of counter example paths obtained when the results of model checking are false. The columns of $Path$, $Probability$ and CC show the number of counter example paths, the summation of occurrence probability of them, and execution time for compatibility checking, respectively. For this example, the obtained sets of counter example paths are compatible in every case.

3.5.5 Discussion

From the results shown in Table 3.1, we can see that our proposed approach is efficient with regard to both execution time and the number of states. Especially, the number of states decreases dramatically. The execution time is also decreased even though we perform model checking several times shown in the column of $Loop$.

On the other hand, in the case when $deadline = 20\mu s$ and $p = 1.85 \times 10^{-4}$, the execution time increases drastically. We think that as shown in Table 3.2 we have to search 2534 paths and this causes the increase of execution time especially for k-shortest paths search. A more detailed analysis shows that the execution time for k-shortest paths search accounts for 1123 seconds of total execution time of 1186 seconds. Also, the results show that the JVM needs 64MB as its heap size in this case. This is because compatibility checking for 2534 of paths needs a large amount of the memory. From the results, we have to resolve a problem of the scalability when the number of candidate paths for a counter example becomes large.

Our abstraction technique restricts the number of paths for a counter example to

the finite number k. Therefore, if k paths are not enough to refute a given property, our technique might output incorrect results. In such a case, however, we can give warning that the result may be incorrect by comparing the result of Value Iteration. The important point is that the probability obtained by Value Iteration is always larger than the actual probability. On the other hand, the probability based on the k paths is always smaller than the actual probability. Therefore, even when the summation of the occurrence probability does not reach to p, if the result of Value Iteration is larger than p, we can warn that the result may be incorrect.

3.6 Summary

This study has proposed the abstraction refinement technique for a probabilistic timed automaton by extending the existing abstraction refinement technique for a timed automaton. In this research, we have shown the concrete algorithms of the reachability analysis method for probabilistic timed automata, and proved its correctness.

Future work includes completion of implementation. General DBM does not support not operator[57]; so we have to investigate efficient algorithms for the not operator.

Chapter 4

Qualitative Analysis of Real-time Distributed Systems Using the Probabilistic Model Checker PRISM

4.1 Introduction

This chapter describes a hybrid technique of probabilistic model checking and simulation to perform qualitative analysis on real-time distributed systems.

Nowadays real-time distributed systems like streaming media systems are widely spreading. These systems require time based transmission such as QoS control to prevent interruption of packet transmission caused by network delay, packet loss, and so on. System developers preliminary have to estimate the QoS by simulation techniques[62] or mathematical analysis[63].

Simulation techniques usually do not guarantee qualitative properties such as the maximum throughput and the minimum jitter, and so on, though they can calculate mean-values along typical traces. On the other hand, mathematical analysis is logically correct, but in many cases the based models are too ideal; hence it is sometimes hard to apply the mathematical analysis to realistic applications. Formal verification techniques, especially model checking techniques[1] are considered as promising techniques for information systems developing due to their ability of exhaustive checking. Among them, probabilistic model checking can evaluate performance, dependability and stability of information processing systems with random behaviors[19].

In order to find if the hybrid approach is applicable to real systems, this study

applies a hybrid analysis technique onto real-time distributed systems, which uses both of simulation and model checking techniques. In our approach, we perform a stepwise analysis using probabilistic models of target systems in different abstract levels. First, we create a probabilistic model with detailed behavior of the system (called detailed model), and apply simulation on the detailed model. Next, based on the simulation results, we create a probabilistic model in an abstract level (called simplified model). Then, we verify qualitative properties using the probabilistic model checking techniques.

4.2 Preliminary

4.2.1 Probabilistic Model Checker PRISM

Here, we simply describe an overview of the probabilistic model checker PRISM[6, 27].

A model checking tool usually has two inputs, a model $\mathcal{M}$ and a logical expression p. The model is typically a transition system which represents behavior of the system to be checked while the logical expression is a temporal logic expression which represents a property to check. The typical output of the model checking tool is whether the logical expression is valid on the model ($\mathcal{M} \models p$). Some model checker outputs a counter example when p is invalid.

The inputs of PRISM include the following three kinds of transition systems as a model:

- Discrete-time Markov chains (DTMCs);
- Continuous-time Markov chains (CTMCs); and
- Markov decision processes (MDPs).

Each of three systems is a probabilistic transition system (Markov chain). The inputs of PRISM also include Probabilistic Computation Tree Logic (PCTL)[35] for DTMC and MDP, and Continuous Stochastic Logic (CSL)[36] for CTMC. They are CTL based logics enchanted with probability.

PRISM has several analysis modes: a simulation mode, a numerical analysis mode, and a verification mode. Using the simulation mode, we can observe the behavior of the given model system visually. The numerical analysis mode can evaluate the value of uncertain variable specified with PCTL or CSL based on the model. Such numerical analysis is considered as a kind of parametric model checking[64]. PRISM can draw a graph with several trials of such numeric analysis. The verification mode is like typical model checking except that PRISM cannot output counter examples.

In this study, we use DTMCs as the model of the network. Here, we describe more precisely on a DTMC. Formally, a DTMC D is a tuple (S, s_init, P, L), where
S : a set of states ("state space") ;
$s\_init \in S$: an initial state;
$P : S \times S \to [0, 1]$ a transition probability matrix where $\Sigma_{s' \in S} P(s, s') = 1$ for all $s \in S$; and
$L : S \to 2^{AP}$ a function labeling states with atomic propositions.

PRISM allows a transition to specify an action and updating expressions on D, where D is a set of variables with finite domain. In other words, a DTMC of PRISM is a kind of an extended automaton with probabilities. Usually, one execution of a transition is translated into a unit time of time elapse (a tick event). Such scheme is known as digital clock view of a DTMC. Using an integer variable (with the upper-bound) explicitly as a clock variable, however, we can also represent a system with discrete time in a DTMC. In this study, we use the latter scheme to avoid the state explosion problem.

In the PRISM description, a model is composed of a number of modules. Each module is a probabilistic automaton which has some variables and probabilistic transitions. In the PRISM model, those modules can interact with each other. In this chapter, the word *module* indicates the module in the PRISM description.

PRISM accepts a Reward Model in which certain values of rewards are assigned to the states and transitions of the probabilistic model[27]. It allows us to evaluate quantitative properties. For example, if we assign a reward of one to all transitions on the model, we can evaluate an expected number of transitions in the paths to reach a given state from an initial state.

4.2.2 Protocols for Net-streaming

Here, we simply summarize typical protocols used in the Internet. Most of the typical protocols have a congestion control mechanism in order to avoid network congestion. For example, TCP (Transmission Control Protocol) uses AIMD (Additive Increase Multiplicative Decrease) type window-flow control as such the mechanism. It controls the data size of sending packets based on the current available bandwidth. Such a scheme has an advantage for the correct data transmission. It, however, allows delays, which is not suitable for real-time data transmission. Therefore, RTSP (Real Time Streaming Protocol) is used for real-time application. RTSP is a protocol for the Internet streaming of voice and movies, on TCP/IP network. Famous congestion control mechanisms for RTSP are RAP (Rate Adaptation Protocol)[65] and TEAR (TCP Emulate At Receivers)[66]. Recently, TFRC (TCP-Friendly Rate Control)[67] attracts attention. Hence, this study models TFRC.

RTSP

RTSP is one of the typical protocols working at end-to-end. RTSP has five states, called SETUP, PLAY, RECORD, PAUSE and TEARDOWN. RTP (Real-time Transport Protocol) is responsible for transmission of stream-data. It determines the throughput of RTP based on the rate control scheme of TFRC using the report message of RTCP (RTP Control Protocol).

TCP Friendly Rate Control TFRC

TFRC is a rate control scheme for fairness between RTP and TCP. It controls the rate in order to avoid bad effects on existing TCP flows in the same network, which increases total effectiveness of the whole network. TFRC controls the rate using the report message of RTCP. The report message contains loss of packets and jitters, which can be estimated via the sequence number of received RTP packets and time stamps, respectively. RFC3448 describes the following formula for determining the throughput:

$$X = \frac{s}{R*\sqrt{2*b*p/3} + (t\_RTO*(3*\sqrt{3*b*p/8}*p*(1+32*p^2)))}$$

X is calculated as Byte/second. The parameters of the formula is summarized in Table 4.1. The calculated throughput is a rate with which a RTSP server should send packets considering the network congestion at the time. Therefore weighted average values of the parameters in a short period are applied into the equation. Paper [67] also defines the calculation methods for the parameters. When the value of X is less than the bandwidth, TFRC lets RTSP set the value as throughput.

Table 4.1: Parameters of the Throughput Estimation Formula

R[seconds]	Round trip time
p[%]	A packet loss rate
s[Byte]	Packet size
b[number of times]	The number of packets acknowledged by a single TCP acknowledgment
t_RTO[seconds]	A TCP retransmission timeout value

4.3 Proposed Approach

Probabilistic model checking which can evaluate complicated properties with a high level of confidence is useful for performance evaluation of information systems[19]. However, if we model whole systems with several simultaneous sessions in detail, we cannot avoid the state explosion problem. To avoid the problem, in our approach we model real-time distributed systems in different abstraction levels. Using both of simulation and model checking techniques, we perform qualitative analysis in a stepwise fashion.

For a probabilistic model in a detailed level, we model behaviors of protocols of RTSP, TCP, and UDP in detail, and perform several trials of simulation in order to analyze throughput, packet loss rates, and so on. A simplified model is based on the simulation results. For the simplified model, data transmission which we want to analyze is modeled in detail, while other data transmission is abstracted. For the transmission which we do not concern, we decide transmission rates probabilistically based on the simulation results.

In the rest of this chapter, we describe the detailed model and simplified model with a case study of a video data streaming system[68]. In our approach, both of the models are described with the PRISM language.

4.3.1 Target System

Here, we introduce an example of a real-time distributed system. As the example system, we select a video data streaming system[68] shown in Fig.4.1. The system is composed of a pair of a video server and its client, a number of pairs of FTP servers and their clients, and a packet generator, which are connected to each other through routers located at the middle of Fig.4.1. The routers are connected through the 10Base–T Ethernet, which is considered as a bottleneck of packet transmission. In the considering scenario, the video server sends 80MB of video data with throughput of 1Mbps using the rate control of TFRC. After 100 seconds from the start of the video streaming, FTP servers and clients start their data transmission through TCP sessions. Also, the packet generator always sends UDP packets with the throughput of 8Mbps as background noise.

4.3.2 The Detailed Model

The main component of the detailed model is a queue which buffers packets of a router in a bottleneck link. Behavior such as packet loss and round trip time is based on the state of the queue. Also, for each application in the system, we model behavior of its server in detail, while behavior of its client is abstracted as

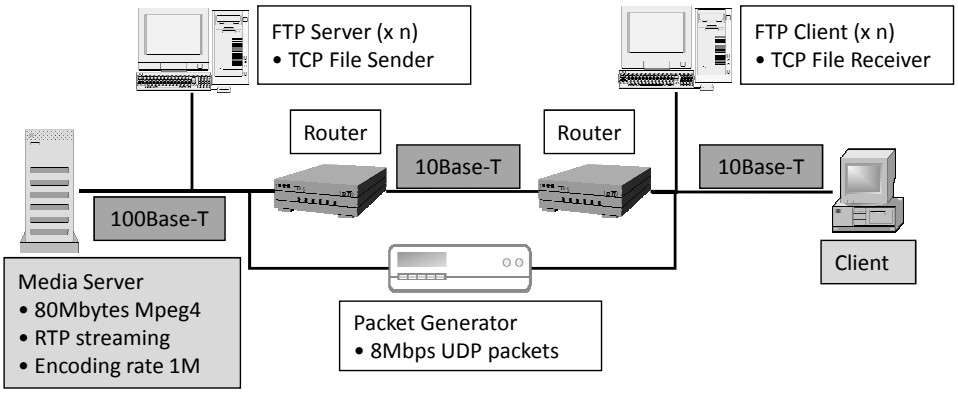


Figure 4.1: A Configuration of Experimental System

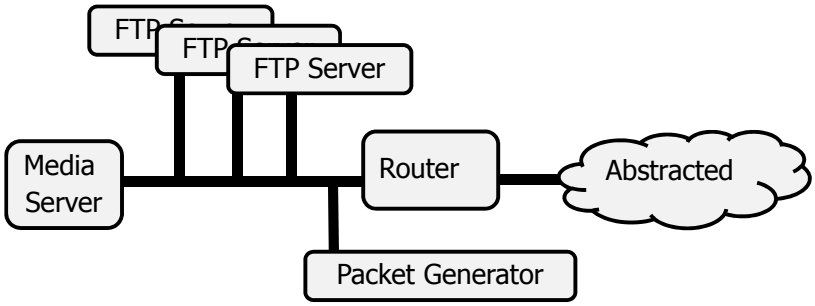


Figure 4.2: An Abstract Outline of the Detailed Model

an operation of dequeue. Time elapsing is controlled discretely with an integer variable. Figure 4.2 is an abstract outline of the detailed model for the case study of Fig.4.1.

The detailed model is composed of seven modules named *Timer*, *Router*, *MediaServer*, *FTPServer(*$\times 3$*)*, *RTTObserver*, *PLRObserver*, while we abstract behaviors of the packet generator as a part of packet transmission behaviors of the module *Router*

Module *Timer*

The module *Timer* manages time elapsing in the detailed model. In this module, we declare an integer variable which represents current time. Time elapsing is based on events such as packet transmission. In the detailed model, each module

registers time of occurrence of the next event. When all modules register the time, the module *Timer* performs time elapsing into the latest time of the registered event. After time elapsing, a corresponding module performs the registered event and registers time of the next event again.

The module *Timer* contains two variables and is implemented as eleven lines of code.

Module *Router*

The module *Router* manages buffer control of the router with a queue which buffers transferring packets. In the module, the current queue length is managed with an integer variable. Enqueue and dequeue behaviors are described in the module as operations. Also, regardless of the current queue length, enqueueing packets are dropped with certain probability. In order to construct the module *Router*, we have to specify the maximum length of the queue, a packet transfer rate of the link between the routers, and a constant probability to drop enqueueing packets randomly as parameters.

In the enqueue operation, if the current queue length becomes larger than the maximum one, the enqueueing packets are dropped (drop tail). The dequeue operation is abstracted; together with the time elapsing operation, a number of packets are output from the queue at a time according to the packet transfer rate of the link.

Figure 4.3 shows the module *Router* described with the PRISM language. The module also manages the history of packet loss intervals used for the congestion control. It contains ten variables and is implemented as about 80 lines of code. In Fig.4.3, the ten variables are firstly declared. Several actions are defined in CCS like expressions with probabilities. For example, the expression

```
[ENQFTP1]  (q_len <= MAXQLEN - pnum_ftp1)
  -> 1 - P_LOSS_RATE : (q_len' = q_len + pnum_ftp1);
   + P_LOSS_RATE : true;
```

stands for that when an action $ENQFTP1$ occurs and $(q\_len$
$\leq MAXQLEN - pnum\_ftp1)$ holds, the variables q_len is updated to $q\_len +$
$pnum\_ftp1$ with probability $1 - P\_LOSS\_$
$RATE$, or do nothing with probability P_LOSS_RATE.

Module *MediaServer*

The module *MediaServer* manages the transmission of RTSP packets. A packet transmission rate is calculated from the throughput equation defined in [67]. To use the equation, we also have to model round trip time and a packet loss rate of the

```
 1. module Router
 2.
 3.   q_len : [0..MAXQSIZE] init 0; //The current queue length
 4.
 5.   //The history of packet loss intervals
 6.   int_p_loss0 : [0..10000] init 0;
 7.   int_p_loss1 : [0..10000] init 10000;
 8.   int_p_loss2 : [0..10000] init 10000;
 9.   int_p_loss3 : [0..10000] init 10000;
10.   int_p_loss4 : [0..10000] init 10000;
12.   int_p_loss5 : [0..10000] init 10000;
13.   int_p_loss6 : [0..10000] init 10000;
14.   int_p_loss7 : [0..10000] init 10000;
15.   int_p_loss8 : [0..10000] init 10000;
16.
17.   //A flag to observe whether the packet loss occurs burstly or not
18.   p_loss_flag : bool init false;
19.
20.   //When the queue length does not reach to its maximum,
21.   //transferring packets are dropped with certain probability
22.   [ENQMS] (q_len <= MAXQLEN - ms_pnum) ->
23.        1 - P_LOSS_RATE :
24.        (q_len' = q_len + ms_pnum) & (p_loss_flag' = false) &
25.        (int_p_loss0' = int_p_loss0 + 1)
26.      + P_LOSS_RATE :
27.        (int_p_loss0' = 0) & (int_p_loss1' = int_p_loss0) &
28.        (int_p_loss2' = int_p_loss1) & (int_p_loss3' = int_p_loss2) &
29.        (int_p_loss4' = int_p_loss3) & (int_p_loss5' = int_p_loss4) &
30.        (int_p_loss6' = int_p_loss5) & (int_p_loss7' = int_p_loss6) &
31.        (int_p_loss8' = int_p_loss7) ;
32.
33.   //When the queue length reaches to its maximum
34.   [ENQMS] (q_len > MAXQLEN - ms_pnum) & (!p_loss_flag) ->
35.        (q_len' = MAXQLEN) & (int_p_loss0' = 0) &
36.        (int_p_loss1' = int_p_loss0 + 1)&
37.        (int_p_loss2' = int_p_loss1) & (int_p_loss3' = int_p_loss2) &
38.        (int_p_loss4' = int_p_loss3) & (int_p_loss5' = int_p_loss4) &
39.        (int_p_loss6' = int_p_loss5) & (int_p_loss7' = int_p_loss6) &
40.        (int_p_loss8' = int_p_loss7) & (p_loss_flag' = true);
41.
42.   // When the packet loss occurs burstly
43.   // (do not update the history of packet loss intervals)
44.   [ENQMS] (q_len > MAXQLEN - ms_pnum) & (p_loss_flag) ->
45.        (q_len' = MAXQLEN) & (p_loss_flag' = (q_len = MAXQLEN));
46.
47.   //The ENQUEUE operations for the three FTP sessions
48.   [ENQFTP1]  (q_len <= MAXQLEN - pnum_ftp1) ->
49.        1 - P_LOSS_RATE : (q_len' = q_len + pnum_ftp1);
50.      + P_LOSS_RATE : true;
51.
52.   [ENQFTP1]  (q_len > MAXQLEN - pnum_ftp1) ->
53.        (q_len' = MAXQLEN);
54.
55.   [ENQFTP2] (q_len <= MAXQLEN - pnum_ftp2) ->
56.        1 - P_LOSS_RATE : (q_len' = q_len + pnum_ftp2)
57.      + P_LOSS_RATE : true;
58.
59.   [ENQFTP2]  (q_len > MAXQLEN - pnum_ftp2) ->
60.        (q_len' = MAXQLEN);
61.
62.   [ENQFTP3]  (q_len <= MAXQLEN - pnum_ftp3) ->
63.        1 - P_LOSS_RATE : (q_len' = q_len + pnum_ftp3)
64.      + P_LOSS_RATE : true;
65.
66.   [ENQFTP3]  (q_len > MAXQLEN - pnum_ftp3) ->
67.       (q_len' = MAXQLEN);
68.
69.   // DEQUEUE is executed together with
70.   //                      the time elapsing event
71.   [TIMER] (q_len != 0) ->
72.      (q_len' = max(0,
73.                q_len - floor(min_lookahead *(RATE_OUT -RATE_PGEN))));
74.
75.   [TIMER] (q_len = 0) -> true;
76.
77. endmodule
```

Figure 4.3: The Module of Router Described with PRISM Language

RTSP session. The module *MediaServer* contains six variables and is described with about 37 lines of code.

In our model, the packet transmission behaviors are abstracted as a number of packets are transmitted simultaneously.

Module *FTPServer*

The slow-start and congestion avoidance behaviors of TCP are embedded to the module *FTPServer*. For each connection of the TCP, we declare two integer variables to manage the slow-start threshold and the window size. In total, we declare four variables and the behavior is implemented as about 26 lines of code for each connection.

Module *RTTObserver*

The module *RTTObserver* observes round trip time of RTSP packets. In the module, the round trip time is obtained using physical delay and delay in the router. The delay in the router is calculated as the time to transmit all packets currently buffered in the router. Therefore, we obtain the delay in the router using current queue length and a packet transmission rate of the link.

Module *PLRObserver*

The module *PLRObserver* calculates a packet loss rate of RTSP packets. In the TFRC specification[67], a packet loss rate is calculated using intervals of packet loss. To avoid the loss rate varying rapidly, a history of the packet loss intervals is used. Nine integer variables are declared to manage the history, which are declared in the module *Router*. In the calculation of the loss rate, recent intervals in the history are weighted heavily.

4.3.3 The Simplified Model

The detailed model described in Sec.4.3.2 is too complicated to verify its qualitative properties using probabilistic model checking.

Here, we create a simplified model based on simulation results on the detailed model in order to perform model checking. Using the simplified model, we can verify the minimum throughput of the media server.

In the simplified model, behavior of application servers which we do not concern is abstracted. The abstraction is based on the simulation results on the detailed model. In the simulation, we obtain probability distributions of transmission rates for the application servers depending on current queue length. The simplified

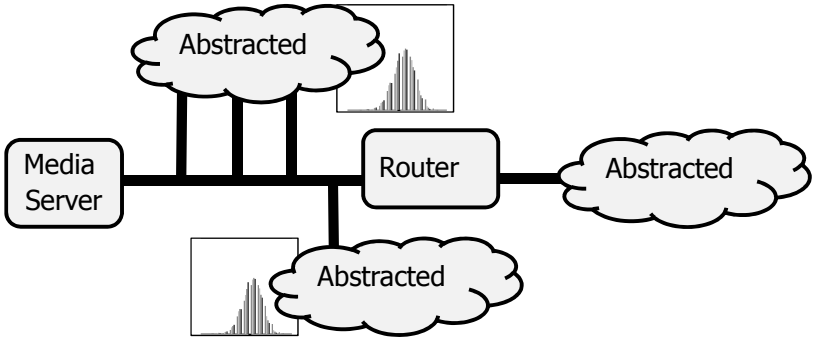


Figure 4.4: An Abstract Outline of the Simplified Model

model decides the packet transmission rate using the distributions to simulate the behaviors of the servers. Fig.4.4 is an abstract outline of the simplified model. In general, the simplified model uses an abstracted module which simulates such a distribution approximately. The number of states is a few, thus the module contributes reducing the number of whole states. The simplified model does not have an integer variable to control time elapsing in order to reduce a state space. We assign a certain period into an action transition of the model. Each module transmits a number of packet due to its current transmission rate. In this study, we let the period be 50ms. The simplified model consists of four modules and has 136 lines of code.

Here, we describe how to analyse such probability distributions on the detailed model and how to construct the abstracted module from the distributions.

Analysis of the Probability Distributions on the Detailed Model

We use reward descriptions of the PRISM to obtain the probability distributions of transmission rates for application servers. In our detailed model, at every one second in the scenario, we calculate a summation of packet transmission rates for all application servers within the time period. If the calculated rate occurs in the range of transmission rates specified by the reward property, we assign the reward one in the reward description. Evaluation of the reward property by the PRISM simulation can generate a histogram of the transmission rates among the extracted paths. Then, we translate the histogram into the discrete probability distribution. Since the distributions depend on the condition of the queue, we add the condition of latest queue length in the reward description.

Figure 4.5 shows a part of the reward descriptions in our detailed model. The

```
rewards "ftp_1th0"
  [CHECK] (prev_q_length >=MAXQSIZE*Q_OCC_LB1) &
          (prev_q_length < MAXQSIZE*Q_OCC_UB1) &
          (ftp_send >= 0) & (ftp_send < 20)
                          : 1; //within 0-80 Kbps
endrewards

rewards "ftp_1th1"
  [CHECK] (prev_q_length >=MAXQSIZE*Q_OCC_LB1) &
          (prev_q_length < MAXQSIZE*Q_OCC_UB1) &
          (ftp_send >= 20) & (ftp_send < 40)
                          : 1; //within 80-160 Kbps
endrewards

rewards "ftp_1th2"
  [CHECK] (prev_q_length >=MAXQSIZE*Q_OCC_LB1) &
          (prev_q_length < MAXQSIZE*Q_OCC_UB1) &
          (ftp_send >= 40) & (ftp_send < 60)
                          : 1; //within 160-240 Kbps
endrewards

rewards "ftp_1th3"
...
```

Figure 4.5: A Part of Reward Descriptions for Analysis of the Distribution

reward is assigned to the transition labeled with the *CHECK* action. The guard condition for each reward description is composed of the conditions of the queue length and those of the transmission rates. The parameters *Q_OCC_LBn* and *Q_OCC_UBn* ($1 \leq n \leq 4$) stand for lower and upper bounds of queue occupancy rates, respectively. Totally, in our detailed model, we specify 140 of reward properties to obtain the histograms for all conditions of the queue.

Construction of the Abstracted Module

The abstracted module is based on the discrete probability distributions obtained from the results of PRISM simulation. Figure 4.6 shows the abstracted module obtained in our experiment shown in Sec.4.4, which simulates the behaviors of four FTP servers in the example of Sec.4.3.1. In the module, there are four transitions labelled with the action *ENQ* with the different guarded conditions involved with the queue length (*q_length*). For each transition, it decides its packet transmission rate (*other_rate*) according to the probability distribution. In the simplified model, the abstracted module interacts with the modules of the media server and queue.

```
module Other_Servers
  other_rate : [10..31] init 10;      // the number of packets in 50 msec]

  [ENQ] (q_length < ceil(85*MAXQSIZE/100)) ->
          0.0000135260 : (other_rate' = 13)       //1040Kbps
        + 0.0000608672 : (other_rate' = 14)       //1120Kbps
        + 0.0004193071 : (other_rate' = 15)       //1200Kbps
        + 0.0022385587 : (other_rate' = 16)       //1280Kbps
        + 0.0094817500 : (other_rate' = 17)       //1360Kbps
        + 0.0281882553 : (other_rate' = 18)       //1440Kbps
        + 0.0671567600 : (other_rate' = 19)       //1520Kbps
        + 0.1294238586 : (other_rate' = 20)       //1600Kbps
        + 0.1955188249 : (other_rate' = 21)       //1680Kbps
        + 0.2156793789 : (other_rate' = 22)       //1760Kbps
        + 0.1728965326 : (other_rate' = 23)       //1840Kbps
        + 0.1040828334 : (other_rate' = 24)       //1920Kbps
        + 0.0497419909 : (other_rate' = 25)       //2000Kbps
        + 0.0188891068 : (other_rate' = 26) //2080Kbps
        + 0.0051669451 : (other_rate' = 27) //2160Kbps
        + 0.0008656662 : (other_rate' = 28) //2240Kbps
        + 0.0001690754 : (other_rate' = 29) //2320Kbps
        + 0.0000067630 : (other_rate' = 30); //2400Kbps

  [ENQ] (q_length >=  ceil(MAXQSIZE*85/100)) & (q_length < ceil(MAXQSIZE*90/100)) ->
          0.0000186290 : (other_rate' = 12) //960Kbps
        + 0.0000521611 : (other_rate' = 13) //1040Kbps
        + 0.0002421768 : (other_rate' = 14) //1120Kbps
        + 0.0009798844 : (other_rate' = 15) //1200Kbps
        + 0.0047541161 : (other_rate' = 16) //1280Kbps
        + 0.0163935037 : (other_rate' = 17) //1360Kbps
        + 0.0452088123 : (other_rate' = 18) //1440Kbps
        + 0.0971091547 : (other_rate' = 19) //1520Kbps
        + 0.1661369826 : (other_rate' = 20) //1600Kbps
        + 0.2123070503 : (other_rate' = 21) //1680Kbps
        + 0.1995238432 : (other_rate' = 22) //1760Kbps
        + 0.1393857652 : (other_rate' = 23) //1840Kbps
        + 0.0729287367 : (other_rate' = 24) //1920Kbps
        + 0.0315314141 : (other_rate' = 25) //2000Kbps
        + 0.0102496656 : (other_rate' = 26) //2080Kbps
        + 0.0025745252 : (other_rate' = 27) //2160Kbps
        + 0.0005178857 : (other_rate' = 28) //2240Kbps
        + 0.0000707901 : (other_rate' = 29) //2320Kbps
        + 0.0000111774 : (other_rate' = 30) //2400Kbps
        + 0.0000037258 : (other_rate' = 31); //2480Kbps

  [ENQ] (q_length >=  ceil(MAXQSIZE*90/100)) & (q_length < ceil(MAXQSIZE*95/100)) ->
          0.0000025649 : (other_rate' = 11) //880Kbps
        + 0.0000205193 : (other_rate' = 12) //960Kbps
        + 0.0000718175 : (other_rate' = 13) //1040Kbps
        + 0.0002872701 : (other_rate' = 14) //1120Kbps
        + 0.0016005048 : (other_rate' = 15) //1200Kbps
        + 0.0065046156 : (other_rate' = 16) //1280Kbps
        + 0.0208270814 : (other_rate' = 17) //1360Kbps
        + 0.0545941412 : (other_rate' = 18) //1440Kbps
        + 0.1149516386 : (other_rate' = 19) //1520Kbps
        + 0.1788974471 : (other_rate' = 20) //1600Kbps
        + 0.2109511461 : (other_rate' = 21) //1680Kbps
        + 0.1836220141 : (other_rate' = 22) //1760Kbps
        + 0.1244212918 : (other_rate' = 23) //1840Kbps
        + 0.0657463764 : (other_rate' = 24) //1920Kbps
        + 0.0270623812 : (other_rate' = 25) //2000Kbps
        + 0.0084539483 : (other_rate' = 26) //2080Kbps
        + 0.0016877118 : (other_rate' = 27) //2160Kbps
        + 0.0002744455 : (other_rate' = 28) //2240Kbps
        + 0.0000230842 : (other_rate' = 29); //2320Kbps

  [ENQ] (q_length >=  ceil(MAXQSIZE*95/100)) ->
          0.0000074358 : (other_rate' = 11) //880Kbps
        + 0.0001041011 : (other_rate' = 12) //960Kbps
        + 0.0002354667 : (other_rate' = 13) //1040Kbps
        + 0.0007460578 : (other_rate' = 14) //1120Kbps
        + 0.0026719279 : (other_rate' = 15) //1200Kbps
        + 0.0078001457 : (other_rate' = 16) //1280Kbps
        + 0.0243274326 : (other_rate' = 17) //1360Kbps
        + 0.0614270772 : (other_rate' = 18) //1440Kbps
        + 0.1240562741 : (other_rate' = 19) //1520Kbps
        + 0.1859939423 : (other_rate' = 20) //1600Kbps
        + 0.2053790519 : (other_rate' = 21) //1680Kbps
        + 0.1726070382 : (other_rate' = 22) //1760Kbps
        + 0.1167344976 : (other_rate' = 23) //1840Kbps
        + 0.0625424460 : (other_rate' = 24) //1920Kbps
        + 0.0255989530 : (other_rate' = 25) //2000Kbps
        + 0.0078497177 : (other_rate' = 26) //2080Kbps
        + 0.0016507458 : (other_rate' = 27) //2160Kbps
        + 0.0002528169 : (other_rate' = 28) //2240Kbps
        + 0.0000148716 : (other_rate' = 29); //2320Kbps
endmodule
```

Figure 4.6: The Abstracted Module for four FTP servers

4.4 Experiments

We have performed some experiments using our PRISM models described in Sec.4.3. We have also modeled the system by NS-2[62, 69] to compare the simulation results. The experiments were performed under an environment of Fedora 13 (64 bit), Intel Core 2 Duo 2.33GHz, and 2.00GB of M.M.

In the experiments, we assume packet transmission parameters as follows: the packet size is 500 Byte, the number of packets acknowledged by a single TCP acknowledgment is one, and the TCP retransmission timeout value is $4 \times RTT$ second. We assume that transmitted packets are lost with the probability 0.001.

In this study, we have performed two experiments.

The first experiment checks the correctness of our detailed model. In the experiment, we performed 1000 trial runs for PRISM and NS-2 simulation, respectively, and compared the simulation results. In this experiment, we consider nine scenarios with respect to the buffer size of routers and the number of the FTP servers. In the scenarios, the buffer sizes are 32, 64, and 128 KB, respectively. Also, the number of the FTP servers are three, four, and five, respectively.

In the second experiment, we performed about 10000 trial runs for PRISM simulation, and created a more simplified PRISM model based on the simulation results. As a target scenario, we select a scenario of 64KB of buffer and four FTP connections. Using the simplified model, we verify the minimum throughput of the RTP session.

4.4.1 Analysis of the Correctness

Before analyzing the 1000 trials of simulation, we extracted one sample from the simulation results by PRISM and NS-2, respectively. Fig.4.7 and Fig.4.8 represent measured throughput and packet loss rates, respectively, in the scenario of 64KB of the buffer and four FTP connections. Throughput in the graph means the average throughput within one second, and a packet loss rate means a calculated value at the time as defined in [67]. In the scenario of the example, file transmission starts after 100 seconds from the start of the RTP session, and this causes the network congestion. Consequently, the throughput of the RTP session goes down and the packet loss rate of it comes up. The simulation results of Fig.4.7 and Fig.4.8 show that our PRISM model and NS-2 model behave similarly even if the network congestion occurs.

To analyze the correctness of our PRISM model in detail, we compare the average, variance, minimum and maximum of throughput of the media server in the 1000 of runs measured by PRISM and NS-2. Tables 4.2, 4.3 and 4.4 represent the analyzed throughput in the period of congestion (after 120 seconds in the simu-

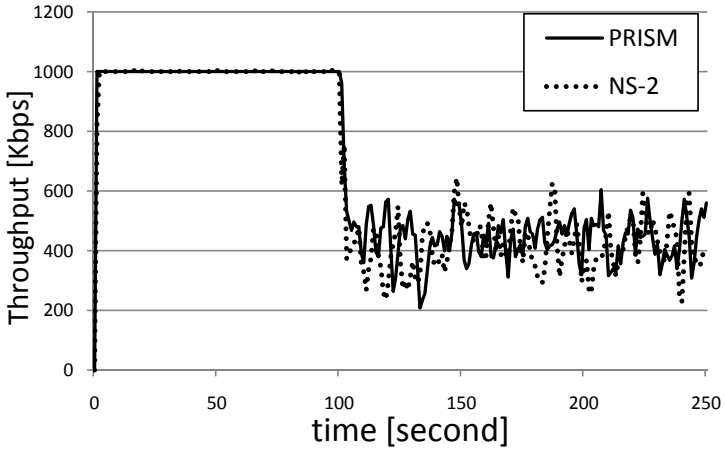


Figure 4.7: Comparison of the Throughput

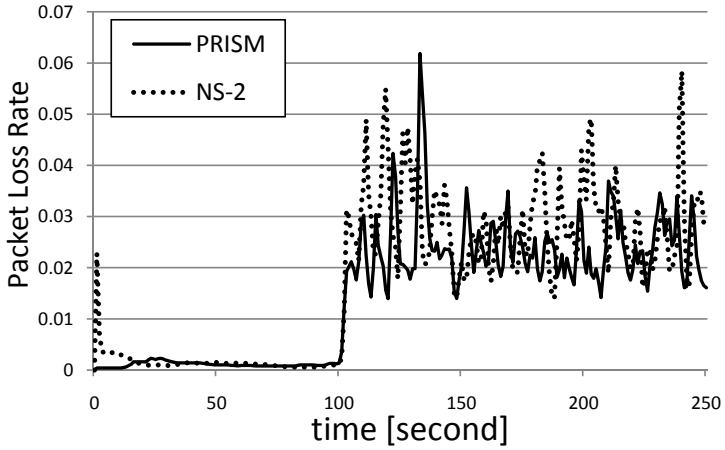


Figure 4.8: Comparison of Packet Loss Rates

lation scenario). The row of $Size$ stands for the buffer size of the router. Also the rows of Max, Min, Ave and Var represent the maximum (Kbps), minimum (Kbps), average (Kbps), and variance of throughput, respectively.

Tables 4.2, 4.3, and 4.4 show that the behavior of our detailed model is similar with that of the NS-2 model for all scenarios. Also we have analyzed packet loss rates and RTT as well. The results also show our detailed model behaves similarly to the NS-2 model. In the cases of the buffer size 32KB, however, we can see the

Table 4.2: Summary of the Analyzed Data (3 FTP servers)

Size	32KB		64KB		128KB	
Model	NS-2	PRISM	NS-2	PRISM	NS-2	PRISM
Max	1000	880	960	920	972	956
Min	44	24	128	132	224	128
Ave	536	523	515	530	535	513
Var	1.55E+04	1.20E+04	1.02E+04	1.03E+04	8.10E+03	1.20E+04

Table 4.3: Summary of the Analyzed Data (4 FTP servers)

Size	32KB		64KB		128KB	
Model	NS-2	PRISM	NS-2	PRISM	NS-2	PRISM
Max	992	796	856	832	768	800
Min	20	4	76	56	100	128
Ave	387	368	399	421	399	399
Var	1.37E+04	7.77E+03	7.82E+03	7.14E+03	5.09E+03	5.92E+03

Table 4.4: Summary of the Analyzed Data (5 FTP servers)

Size	32KB		64KB		128KB	
Model	NS-2	PRISM	NS-2	PRISM	NS-2	PRISM
Max	816	644	688	660	664	680
Min	4	4	36	40	84	100
Ave	289	270	317	332	317	330
Var	1.15E+04	5.63E+03	6.43E+03	5.35E+03	3.77E+03	3.94E+03

difference of the maximum throughput between the detailed model and NS-2. We think one of the reasons is that we strongly abstract a packet sending mechanism in the PRISM model, that is, when the packet transmission rate is high, our model transmits a number of packets at a time. When the buffer size is small, transmitted packets tend to be lost because of such abstraction. We think this causes the differences of behaviors between the PRISM model and NS-2 one.

For one trial run of the PRISM simulation, it takes 2.5 seconds averagely, while it takes 34.1 seconds in the simulation of NS-2.

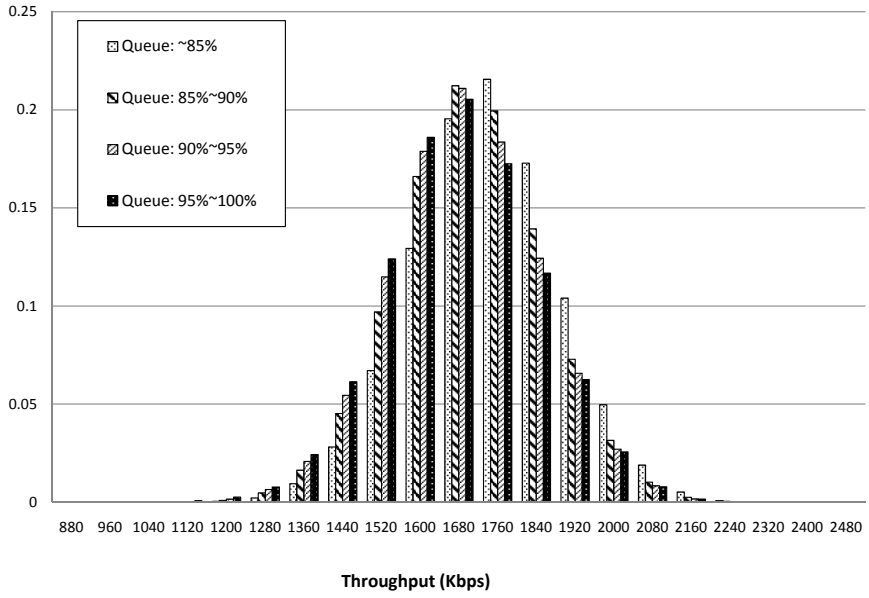


Figure 4.9: The Discrete Probability Distribution of Throughput of the FTP Servers

4.4.2 Verification results for the simplified model

Here, we verify the minimum throughput that the media server may provide in the worst case. In the verification, we use a simplified model based on the simulation results on the detailed PRISM model. The simulation results should contain discrete probability distributions of the throughput of FTP servers and the packet generator. Since the packet generator generates UDP packets at the same rate in the example, we can decide the transmission rate to be the same rate.

The discrete probability distributions of the total throughput of the four FTP servers are shown in Fig.4.9, where the buffer size is 64KB and the number of the FTP is four. The charts are divided with respect to values of buffer occupancy. It takes about 173 minutes to perform 10000 trials of PRISM simulation.

Based on the results of Fig.4.9, we create the simplified model. The simplified model is described with 136 lines of code and seven variables. In order to reduce a state space, we have reduced a range of integer variables for the control of packet loss intervals.

The verification property for checking the minimum throughput is given as follows; $P_{>0}$ [F $measure$ & $throughput$ $\leq x$]. This property means the probability of $throughput$ being the value of x is greater than 0, where $throughput$ is a vari-

able to manage the throughput of the RTP session within one second periodically and $measure$ is a boolean variable which becomes true every one second. The verification is performed with varying the value of x from 0. The minimum throughput is defined as a minimum value of x such that the result of model checking becomes true. In the experiment, we have performed model checking with varying x from 0 to 56 which is the minimum throughput obtained by simulation on the detailed model.

Model checking on the simplified model outputs the minimum throughput 20Kbps. The number of states constructed by PRISM is 10885476, and it takes 940 seconds for model checking. We can see that the obtained minimum throughput doesn't contradict with the simulation results shown in Table 4.3.

4.4.3 Discussion

In the simulation on the detailed model, we have obtained similar results with NS-2. We conclude that we have modeled correctly the behavior of real-time distributed systems. Also, the execution time of PRISM simulation for one trial is shorter than that of NS-2. We think that this is because NS-2 implements behavior of protocols definitely while our detailed model has some abstracted behavior. From the experimental results, we expect that we can use PRISM as a network simulator for the real-time distributed systems.

In the verification on the simplified model, it has taken about 173 minutes for simulation and about 15 minutes for model checking. We think that we can verify the property within the realistic time. For the state space, we have constructed about ten millions of states, though we reduce the space by some abstraction. In order to analyze other qualitative properties or apply our technique into more complicated systems, we have to apply other abstraction techniques to reduce the state spaces as a future work.

For validity, we did not fully show the validity of our simplified model by the experiment. Therefore, we cannot say the results of probabilistic model checking are reliable. We believe that, however, as reported in Paper[19] there are many works in which they apply probabilistic model checking to performance evaluation of information systems , and our hybrid approach is useful to reduce verification costs. We have to show the validity of our simplified model by performing other experiments in the future.

4.5 Summary

This study presents a hybrid evaluation method for a real-time distributed system based on the probabilistic model checking technique and simulation. In our

approach, we perform stepwise analysis using probabilistic models of target systems in different abstract levels (detailed model and simplified model). To validate the correctness of our model, we model it in a model for the well-known network simulator NS-2, and give the comparison of their simulation results. The comparison shows that the result of PRISM simulation is very similar to that of NS-2. It shows that the proposed approach is useful to analyze the network performance. We believe that such analysis is useful for other kind of network analysis.

The future works include validation of our simplified model, and also automatic derivation of the simplified model suitable for model checking analysis. Many abstraction techniques are proposed for model checking. We want to apply such techniques to the process.

Chapter 5

Formal Verification with a Stepwise Abstraction Approach for UML/OCL Based Design of Real-time Systems

5.1 Introduction

In this study, we propose a new method to verify consistency of timeliness QoS of component-based real-time systems. We assume that timeliness QoS is not only given to a whole system (Required QoS) but also associated with each component of a given system (Provided QoS). Timeliness QoS is a time aspect of QoS (Quality of Service) features[52]. In this study, we treat jitter, latency and throughput as timeliness QoS.

The proposed method is a revised version of [53], which uses Linear Programming (LP) for some of verification. The approach has a disadvantage that connection among components has to be acyclic, and it cannot be applied to hierarchical design. The method proposed in this study uses abstract QoS automata instead of using LP; thus it improves the former disadvantage. The heart of the technique is formally to ensure that the required timeliness QoS is satisfied under the provided timeliness QoS, if some properties of the network and the class diagram are given.

In order to avoid state-explosion while performing model checking, we separate the problem into two steps. The first step checks the satisfiability using an abstract model of each of components derived automatically from the provided QoS. The second step performs model checking for each component independently using a more detailed version of the behavioral model of a component. Such an approach

efficiently reduces the number of total states to be checked. Moreover the approach can be extended into hierarchical design; therefore it has good scalability.

5.2 Preliminary

5.2.1 Timeliness QoS

Main building blocks of our model are components. Each component has one or more *interfaces* to the environment, where all interactions between components are conducted via the interfaces. Since we are mostly dealing with real-time systems and timeliness QoS, we shall assume that the interaction of a component with its environment is carried out via *input* and *output* signals. As a result, interfaces of a component specify signals that the component receives or emits.

Each component is associated with a number of input and output signals. In this study, signals are denoted by x, y and z. Time of occurrence of a signal is denoted via a non-negative sequence of rational numbers. For example, the time of occurrence of a signal x is denoted with $x_1, x_2, \ldots$ representing time of first, second, ... occurrence of x.

Timeliness QoS expressions [70] such as jitter, throughput and latency can be expressed via first-order logic formulae on the set of time of occurrence of signals.

Throughput of at least (most) K within the time period T, for signal x can be written as the first order formula $\forall i \in \mathbb{N}: \ x_{i+K-1} - x_i \leq T$ ($\forall i \in \mathbb{N}: \ x_{i+K-1} - x_i \geq T$), respectively.

Notice, paper [55] refers to the above QoS constraint as Non-Anchored throughput.

Jitter , also called Non-Anchored jitter , of a signal x can be defined by the expression $\forall i \in \mathbb{N}: \ T - m \leq x_{i+1} - x_i \leq T + M, \quad$ where T is the period of the jitter and m, M are constant rational numbers.

Latency of at most T unit of time between two signals x and y as $\forall i \in \mathbb{N}: \ 0 < x_i - y_{Ki+K'} \leq T$. A special case of the above definition (for $K = 1$ and $K' = 0$) is the well-known definition of latency $\forall i \in \mathbb{N}: \ 0 < x_i - y_i \leq T$ that applies to the time difference of the i-th occurrence of x and y.

5.2.2 UML/OCL Based Design of Real-time Systems

A real-time system can be designed as a set of components where signal communication links exist among pairs of components. We can describe such components in a UML class diagram in Fig. 5.1.

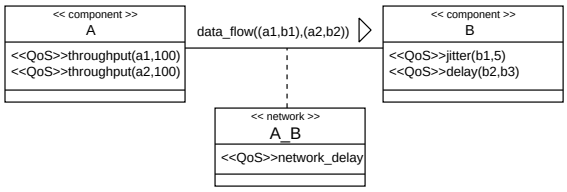


Figure 5.1: A Configuration of Components in UML Class Diagram

Type Component can be specified by Stereotyping "component," by which user can easily extend UML specifications. A signal communication can be specified with the Association.

Each of components has provided QoS, which can be represented via OCL annotation. Each of network links (which has association class with the stereotype "network") also has network properties represented via OCL annotation. The network properties are the same as timeliness QoS. Attribute regions of each class includes special variables for QoS with "QoS" stereotype (in Fig. 5.1). The following is the syntax of the variables.

Throughput Variable := "throughput(" signal "," period ")" ;
Jitter Variable := "jitter(" signal "," period ")" ;
Latency Variable := "delay(" output "," input ")" ;

A class with the "component" stereotype has three categories of timeliness QoS (Jitter, throughput and latency), while a class with "network" has two categories of timeliness QoS (Jitter and latency).

The OCL description is given as follows[70].

QoS description := "context" className invariant* ;
invariant := "inv: self." constraint ;
constraint := variable op constant;
variable := Throughput Variable | Jitter Variable | Latency Variable
op := "$>$" | "$<$" | "$\geq$" | "$\leq$" ;

For example, the followings are examples for Fig. 5.1, where – means a comment line.

context A
inv: self.throughput(a1,100) $\geq$ 20
– signal a1 is emitted at least 20 times in the period 100 units of time
inv: self.throughput(a2,100) $\leq$ 10
– signal a2 is emitted at most 10 times in the period 100 units of time

context B
inv: self.jitter(b1, 5) $<$ 1
– signal b1 has jitter 1 with period 5 units of time
inv: self.delay(b2,b3) $<$ 5
– latency between receiving signal b2 and sending signal b3 is less than 5 units of times
context A_B
inv: delay $\leq$ 100
– latency (network delay) between component A and component B is less than 100 units of time

5.3 The Verification Method

The verification consists of two steps; First Step and Second Step. If some components are not simple enough, we repeat the process again from First Step on each of the components. The following is the abstract level of steps of the proposed method.

input:

- system required timeliness QoS represented in OCL;
- component level provided timeliness QoS represented in OCL; and
- network configuration represented in UML/OCL class diagram.

output:

- component level behavioral specification represented in UML/OCL statechart which satisfies required timeliness QoS under the configuration;
- or failure.

1. First step
   (a) We generate a test automaton from the required timeliness QoS.
   (b) We generate an abstract QoS automaton from each of the provided timeliness QoS.
   (c) We generate a configuration automaton from the network configuration.
   (d) We check the consistency from parallel composition of the above automata.

(e) If deadlock is detected by model checking, return failure. We have to reconfigure the requirement or provided conditions.

(f) If deadlock is not detected, go to Second Step.

2. Second Step

(a) If the component is not small enough to represent simple state-chart, then refine the component by;

i. renaming provided QoS of the component to required QoS;

ii. design sub components and provided QoS of each of them;

iii. design network configuration; and

iv. we repeat the First Step until the component is small enough.

(b) If the component is small enough to represent a simple state-chart, then we describe the state chart of the component.

(c) We translate a test automaton from the provided timeliness QoS.

(d) We design network of timed automata from the state-chart.

(e) We check the consistency from parallel composition of the above automata.

(f) If deadlock is detected by model checking, return failure. We have to reconfigure the state-chart.

(g) If deadlock is not detected, return success.

5.3.1 The First Step

Verification inputs are given as follows.

- Required QoS;
- a set of components with provided QoS; and
- a configuration automaton which represents network properties.

The output is whether a given required QoS is satisfied under a given set of components with provided QoS and a given set of network links with network properties.

In usual methods, designer models behavior of each component in a network of timed automata and for the whole network of timed automata. Then the designer performs model checking, which often results in state explosion. Here, we give a new method, in which timed automata (We call each of them an **abstract QoS automaton**) is derived automatically from the provided QoS. The important point is that derived automata are so small that state-explosion is avoided. Here, we give a translate rule for each provided QoS.

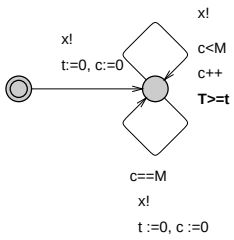


Figure 5.2: An Abstract QoS automaton for Anchored Throughput

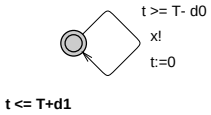


Figure 5.3: An Abstract QoS automaton for Non-Anchored Jitter

Throughput

A translated abstract QoS automaton for throughput is shown in Fig.5.2. The automaton transmits signal x at least M times during the period T. The variable c and clock variable t are used for such the control.

When "throughput must be at least k frames per P ms" is given as a provided QoS, the corresponding abstract QoS automaton is generated with substitution $M = k$ and $T = P$.

Jitter

A translated abstract QoS automaton for jitter is also shown in Fig.5.3. The automaton transmits signal x with the period T. The allowed jitter is $[T - d_0, T + d_1]$. Using the clock variable t, it transmits signal x at every $[T - d_0, T + d_1]$ period.

When "jitter must be $[-d'_0, d'_1]$ with a period T'" is given as a provided QoS, the corresponding abstract QoS automaton is generated from Fig. 5.3 with substitution $T = T'$, $d_0 = d'_0$ and $d_1 = d'_1$.

Latency

For signal x and y, let m and M be the minimum and maximum latency, respectively. A translated abstract QoS automaton for latency with above parameter

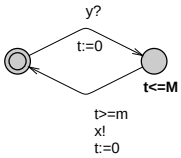


Figure 5.4: An Abstract QoS automaton for Latency

is shown in Fig.5.4. The automaton transmits signal x after receiving signal y with the latency $[m, M]$.

Unfortunately, the automaton does not accept input y until it emits output x. To avoid the problem, a set of the same automata is needed. The number of automaton decided from throughput property of the components.

When "latency for signal x and y must be $[m, M]$" is given as a provided QoS, and also parameter T representing period of signal y is given, the corresponding abstract QoS automata are generated from Fig. 5.4. The number of copies is T.

Configuration Automaton

A configuration automaton models interfaces among components. As each component has several inputs and outputs, such an I/O is represented as a channel in the configuration automaton. Each channel synchronizes with some I/O of some components with provided QoS (abstract QoS automaton). Abstract QoS automata and the configuration automaton communicate each other as described above.

Test Automaton

For a given required QoS, we can verify whether the required QoS is satisfied with the system by generating a corresponding test automaton from the required QoS. Fig.5.5, 5.6, and 5.7 show templates of test automata for throughput, jitter, and latency, respectively. For such templates, substituting each parameter with concrete value specified by the required QoS, we can obtain the test automaton.

Throughput For a non-anchored throughput of which a signal e occurs at least k times in a period T and at most k times in a period $T0$, a network of test automata consisting of k processes of timed automata in Fig.5.5 observes the throughput.

The test automaton observes if $T0 \leq T(e, i+k) - T(e, i) \leq T$ holds for some i, where $T(e, i)$ means the time of i-th occurrence of signal e. With k copies of such test automata, they can observe if $\forall i(T0 \leq T(e, i+k) - T(e, i) \leq T)$ holds.

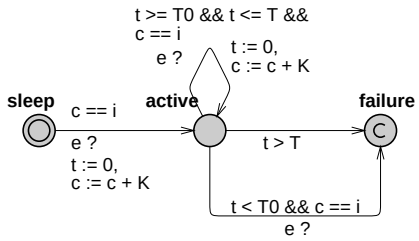


Figure 5.5: A Test Automaton for Throughput

In other words, they can observes at least k times signal e occurs during $[T0, T]$. Parameters of the test automaton are k, T and $T0$.

In the network of test automata, the variables c is shared among automata globally. Each of timed automata is activated by turns along the value of variable K. When there exists common divisor k for $T, T0$ and n, we can reduce the number of copies of the test automaton to k/n with the parameters T/n, $T0/n$ and k/n. The discussion of such a reduction technique is described in Sec.5.4.4.

Jitter Figure 5.6 shows a test automaton for anchored jitter. It observes whether a signal e occurs periodically in the period $[nT - delta_0, nT + delta_1]$, where $n = 1, 2, 3, \ldots$.

The automaton in Fig.5.6 has two clocks $t0$ and $t1$. A path from $s1$ to $s2$ via $se12$ or $sl12$ observes that the time of j-th occurrence of signal e is during the period $[jT - delta0, jT + delta1]$ using clock $t0$, while a path from $s2$ to $s1$ via $se21$ or $sl21$ observes that the time of $j + 1$-th occurrence of signal e is during the period $[(j + 1)T - delta0, (j + 1)T + delta1]$ using clock $t1$.

Latency Figure 5.7 provides a component of test automata for latency between a signal x and y. The test automaton shown in Fig.5.7 observes if $\forall i(T(y, i) - T(x, i) \leq T)$ holds, where $T(e, i)$ means the time of i-th occurrence of signal e.

We have to use T/D copies of such test automata, where D is period of signal x. Variable cx and cy are shared variables with them, which serve to count the occurrence of signal x and y.

Verification

The behavior of whole systems with timeliness properties also is modeled as a network of timed automata, which is called a configuration automaton. Parallel composition of an abstract automaton for every component and the configuration

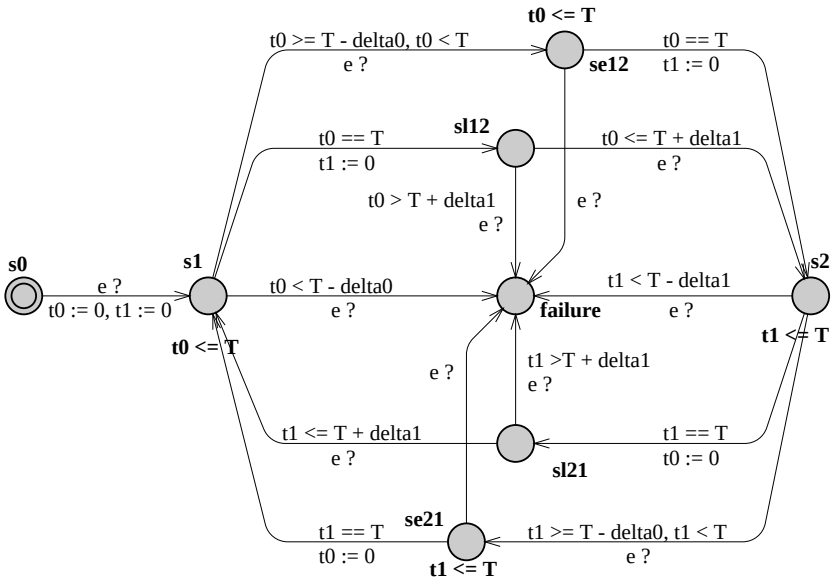


Figure 5.6: A Test Automaton for Jitter

automaton and the test automaton for specified timeliness QoS decides whether the whole system satisfies the specified timeliness QoS.

For the detail of process of the verification, refer [71].

Category Based Model Checking

Verification is performed for every timeliness QoS category (latency jitter and throughput). The idea and approach is very simple. When we want to check only latency as the required QoS, we build an abstract automaton for provided QoS of latency only. The divided and conquer approach reduces the size of states.

5.3.2 The Second Step

For each of components, Second step has the following two cases depending on the component's abstraction

- We repeat First step to the given component recursively.
- We design detailed behavior of the component and verify whether provided QoS is ensured by the design.

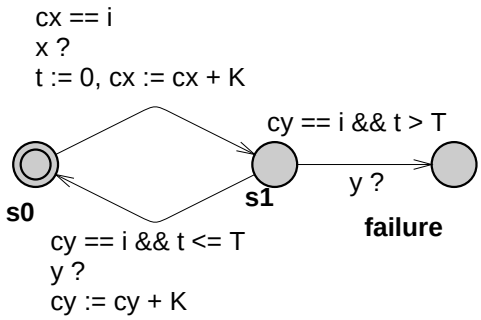


Figure 5.7: A Test Automaton for Latency

If the size of the given component is large and designer has to design the given component from more detail components, then repeats First step. Hereafter, we describe the later case.

At the Second step (of the later case), verification is independently performed for each component. Before the Second step, the designer has to give detailed behavior of each component. Such behavior is given as the UML state-chart. In order to give time constraints on events, the state-chart has clocks.

Verification inputs are given as follows.

- component's behavior given as the UML state-chart with clocks; and
- component's provided QoS.

The output is whether provided QoS is satisfied under the given UML state-chart with clocks.

The verification is performed based on a test automaton. We have to translate the UML state-chart with clocks to a network of timed automata.

A state-chart can represent hierarchical architectures; while a network of timed automata is a simple flat structure model. In general, hierarchical structure can be flatten, but such translation increases the number of states. There are several translations, and we adapt the one in [72]. The translation itself is an algorithm to translate Hierarchical Timed Automata (HTA) to a network of timed automata used by UPPAAL[5]. Thus, we have to translate state-charts to HTA. Fortunately, syntax and semantics of state-chart and HTA are both similar, the translation is simple.

We add the following constraints on the state-chart.

- the state-char diagram has clocks; and

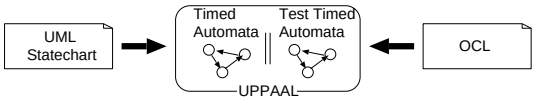


Figure 5.8: Verification on UPPAAL Based on Test Automata

- arcs in the stat-chart has clock constraints in a form of the one same as Timed automata in UPPAAL.

We also use test automata to check timeliness QoS. Test automata for jitter, latency and throughput are given in Sec.5.3.1.

The verification can be performed with UPPAAL[5, 22]. Thanks to test automata, we just check deadlock property for each QoS. Logical expression for the deadlock property in UPPAAL is "**AG** not deadlock."

5.4 Experiment

The proposed method is applied to an example.

5.4.1 The example

A media server is an application delivering video stream and audio stream to Digital Television and Audio System[73, 74]. Each of output devices required timeliness QoS (throughput). Figure 5.9 shows the class diagram of the application, which consists of twelve components.

In order to compare the proposed method to the old method, which uses LP solver to First Step, we merge the twelve components to three components (Server 3 components , Audio client 4 components, and Video clients 5 components).

5.4.2 First Step

The following is the provided QoS.

- Throughput of Component MS–Server is equal or greater than 100 frames/s.
- Processing latency of Component MS-Storage is equal or less than 5ms.
- Network latency between MS–Server and Digital–TV is equal or less than 100ms.

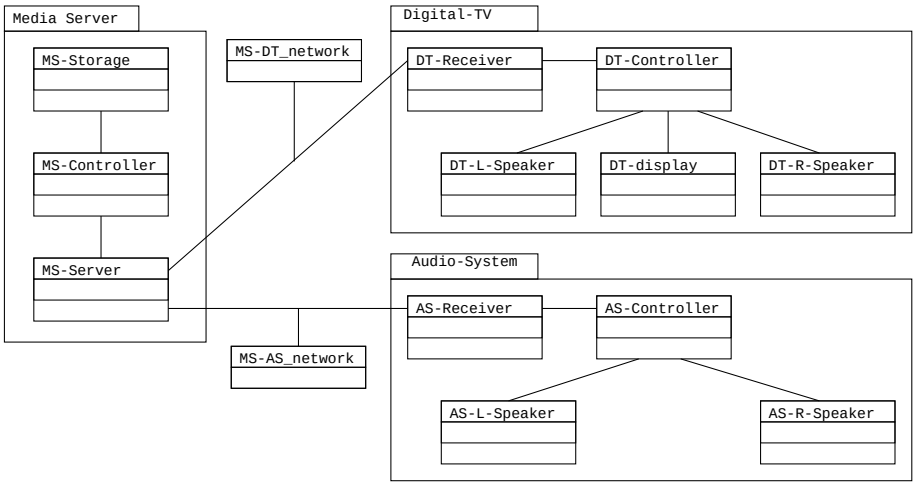


Figure 5.9: The Class Diagram of Media Server

- Network latency between MS–Server and Audio–System is equal or less than 150ms.

We give the following requirement for the Required QoS for the system.

- Throughput of Digital Display (DT Display) must be at least 30 frames/sec.

For these provided QoS, and a configuration automaton derived from the UML class diagram, and Required QoS for the whole system, we apply the verification along with First Step.

Figure 5.10 shows the Configuration automaton for the experiment. The Configuration automaton in Fig.5.10 represents connection among the components. It uses channels to communicate abstract QoS automata providing the provided QoS mentioned above. For example channel x is used for communication to the abstract QoS automaton with throughput 100 frames/sec at a transition between MS–Controller and MS–Server. In order to avoid unfairness that frame communication occurs only between MS–Server and Digital–TV (or only between MS–Server and Audio–System), we use a parameter MAX_FRAME, which is used in a condition that the maximum successive occurrence of signals between the same components. We use a condition $MAX\_FRAME = 1$ for the experiment.

Figure 5.11 shows the test automaton for throughput as Required QoS. The test automaton observes throughput of 3 frames per 100ms. As required QoS, the required value of throughput is 30 frame/sec, We have to need 30 processes of throughput test automata to observe the throughput exactly.

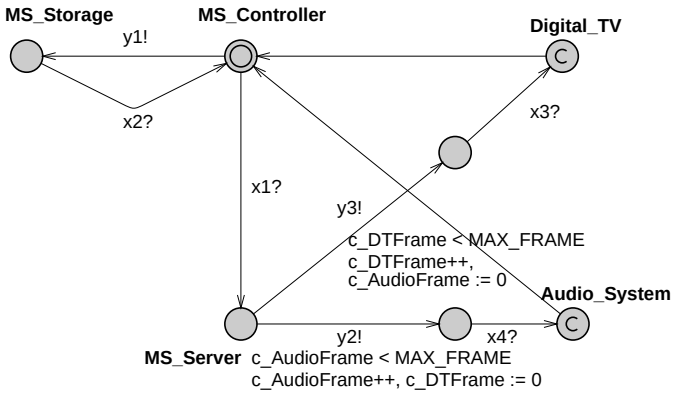


Figure 5.10: The Configuration Automaton

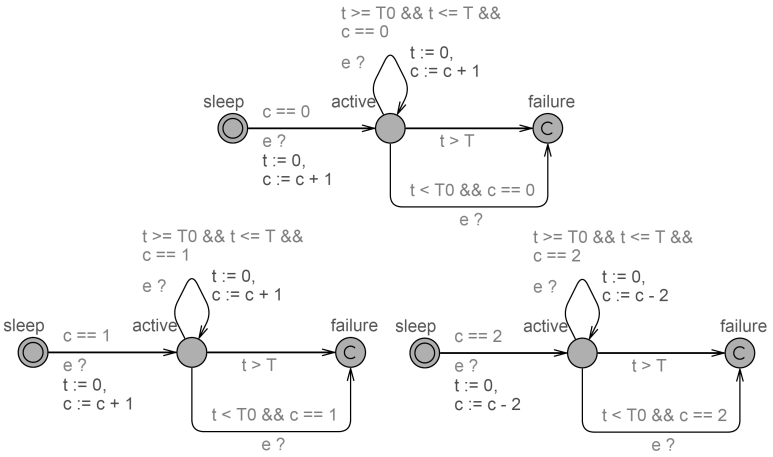


Figure 5.11: The network of test automata for the given required QoS

At the First Step, we have performed verification experiments for two configurations: (1) an abstract QoS which outputs ten frames per 100 msec, and (2) an abstract QoS which outputs 30 frames per 300 msec, respectively, are used for abstraction of provided QoS for MS–Server.

The provided QoS of MS–Server is at least 100 frames/sec. To prevent the jitter of frame signals, we adopt 100ms and 300ms as the period in the experiments. The values are set in the configuration (1) and (2).

Table 5.1: The Result (1) of the First Step

# of P	result	CPU time	Used memories
3	not valid	0.3 ms	23.9MB
6	not valid	1.4 ms	24.4MB
9	valid	28 ms	24.8MB
12	valid	50.6 ms	25.9MB
15	valid	83.6 ms	26.8MB
30	valid	480 ms	34.4MB

Table 5.2: The Result (2) of the First Step

# of P	result	CPU time	Used memories
3	not valid	0.6 ms	23.9MB
6	not valid	0.7 ms	24.4MB
9	not valid	1.2 ms	24.7MB
12	not valid	1.6 ms	25.0MB
15	not valid	2.1 ms	25.1MB
30	valid	957 ms	40.4MB

Each of two experiments is performed with several numbers of test automata: 3, 6, 9, 12, 15 and 30. We have obtained CPU times and sizes of memory consumed. The experiments are performed in the following environment: CPU is Intel Core 2 Duo 2.33GHz, OS is Windows Vista Business and M.M. is 2GB. We used UPPAAL4.1.0 as a model checker. Table 5.1 and Table 5.2 show the results of (1) and (2), respectively. The column of "# of P" shows the number of processes (the number of test automata).

In the previous experiment, we have performed First Step with Linear Programming solver. In the experiment, we have performed it in 78 ms (although it has been performed in different environment).

5.4.3 The Second Step

After First Step, we design inner behavior of each component.

In the example, recursive application of First Step is not performed, because each component is small enough. Behavioral specification is described in UML state-chart. The design must meet the provided QoS. Figure 5.12 shows behavioral specification of Component MS–Storage.

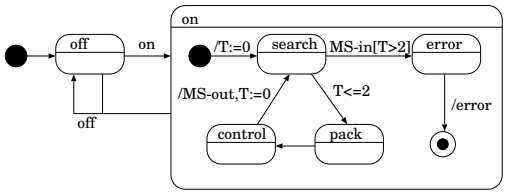


Figure 5.12: The UML Statechart Diagram of Component MS-Storage

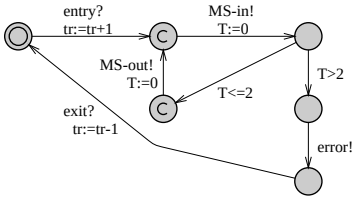


Figure 5.13: The UPPAAL Timed Automaton of Component MS-Storage

In order to verify timeliness QoS for each component, State-chart must to be translated into a network of timed automata and also timeliness QoS is converted into test automata. Figure 5.13 depicts the translated result of *on* part in Fig.5.12.

The translating times are summarized as follows.

Translation time	: 1153 ms
The number of states(before)	: 89
The number of states(after)	: 179

For every component and for every timeliness QoS, verification is performed. The total CPU time is about one seconds. We found that for every component, the verification is performed within a few seconds with UPPAAL without state explosion. Also we found that there is no deadlock.

5.4.4 Discussion

Table 5.1 shows that when we perform the experiment with nine test automata, it outputs the correct result. The result of Tab.5.2, however, shows that we cannot obtain the correct result until the number of test automata increases to 30. When we perform the experiment with 30 test automata, the CPU time increases exponentially. Therefore, we can conclude that there is a trade-off between degree of precision and CPU times. As shown in both tables, the CPU times of the experiments with 30 test automata are too large. Thus, as we consider the trade-off, in

this experiment, the trade-off point is at which the number of process is 15.

Though we cannot exhibit that our proposed method is better than that of linear programming based method with respect to the CPU time, the performance of the proposed method is within useful reasonable time. The linear programming method has many constraints on configuration, while the new proposed method is flexible and is able to apply recursively along with component hierarchy, which are the advantage of the proposed method.

Our proposed method has more acceptable inputs than the former method. The difference between this and that of general class is very small. It is although not faster than the former method, it is more flexible than the former method.

5.5 Summary

We proposed a stepwise verification method for design of real-time systems with UML/OCL focusing on timeliness QoS aspects. The method uses abstract QoS timed automata in order to reduce the possibility of state explosion. The method can be applied to a design with complex connection of components.

Future works include simultaneous verification of several kinds of timeliness QoS, and utilization of feedback information such as verification counter-examples.

Chapter 6

Conclusion

6.1 Summary

In this study, we have proposed model abstraction techniques for model checking of real-time systems.

First, we have proposed a model abstraction technique for timed automata based on the CEGAR framework. In the technique, we remove all of the clock variables from timed automata, and refine the abstract model by modifying transition relation on the model. We have extended the abstraction technique into abstraction for probabilistic timed automata. In the technique, we remove clock variables from given probabilistic timed automata as well as the original one. Then, we apply probabilistic model checking to the generated abstract model which is just a Markov decision process (MDP) with no time attributes. In general, probabilistic model checking does not produce concrete paths as a counter example which are required for abstraction refinement. Therefore, we also perform k-shortest paths search to obtain the concrete paths.

Second, we have proposed a QoS analysis technique of real-time distributed systems based on hybrid analysis of probabilistic model checking and simulation. In the hybrid analysis approach, we perform stepwise analysis using probabilistic models of target systems in different abstract levels. First, we create a probabilistic model with detailed behavior of the system (called detailed model), and apply simulation on the detailed model. Next, based on the simulation results, we create a probabilistic model in an abstract level (called simplified model). Then, we verify qualitative properties using the probabilistic model checking techniques.

Third, we have proposed a technique to verify consistency of timeliness QoS of component-based designed real-time systems. In order to avoid state-explosion while performing model checking, we separate the problem into two steps. The first step checks the satisfiability using abstract model of each of components derived

automatically from the provided QoS. The second step performs model checking each of components independently using more detailed version of behavioral model of a component. Such an approach efficiently reduces the number of total states to check.

6.2 Directions of Future Research

Although applying the CEGAR framework to model checking of timed automata or probabilistic timed automata helps us to reduce the state space of the models, the experimental results have shown that execution time of whole CEGAR processes increases. One direction of future research is to reduce the number of CEGAR loop. In general, the number of CEGAR loop depends on choice of extracted counter example paths. Therefore, we are going to analyze which kinds of counter example paths are effective for reducing the number of CEGAR loop.

In the model checking, diagnostic information, which is provided as a counter example, is thought to be useful for error correction In the probabilistic model checking, however, a detected counter example is represented as a set of paths on a given model. Therefore, when the number of such paths becomes extremely large, the counter example cannot help us to correct detected errors. This problem might occur in our CEGAR technique for probabilistic timed automata. Another direction of future research is to improve the descriptions of a counter example on probabilistic timed automata. We are considering to use a technique proposed in [75], which can describe a counter example on DTMCs with different levels of abstraction.

Bibliography

[1] E. M. Clarke, O. Grumberg, and D. A. Peled, editors. *Model Checking*. MIT Press, 1999.

[2] M. Fitting, editor. *First-order Logic and Automated Theorem Proving*. Springer, 1996.

[3] G.J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, Vol. 23, No. 5, pp. 279–295, May 1997.

[4] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. of the 14th Int. Conf. on Computer Aided Verification*, Vol. 2404 of *LNCS*, pp. 241–268, 2002.

[5] G. Behrmann, A. David, K.G. Larsen andJ. Hakansson, P. Petterson, W. Yi, and M. Hendriks. Uppaal 4.0. In *Proc. of the 3rd Int. Conf. on Quantitative Evaluation of Systems*, pp. 125–126, October 2006.

[6] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A Tool for Automatic Verification of Probabilistic Systems. In *Proc. of the 12th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, Vol. 3920 of *LNCS*, pp. 441–444, March 2006.

[7] T. Nipkow, L.C. Paulson, and M. Wenzel, editors. *Isabelle/HOL*, Vol. 2283 of *LNCS*. Springer, 2002.

[8] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[9] A. Bove, P. Dybjer, and U. Norell. A Brief Overview of Agda - A Functional Language with Dependent Types. In *Proc. of the 22nd Int. Conf. on Theorem*

Proving in Higher Order Logics(TPHOLs 2009), Vol. 5674 of *LNCS*, pp. 73–78, August 2009.

[10] J-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[11] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and V. Helmut. Counterexample-guided Abstraction Refinement for Symbolic Model Checking. *Journal of the ACM*, Vol. 50, No. 5, pp. 752–794, 2003.

[12] E. M. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based Abstraction-Refinement using ILP and Machine Learning Techniques. In *Proc. of the 14th Int. Conf. on Computer Aided Verification*, Vol. 2404, pp. 695–709, 2002.

[13] E. M. Clarke, A. Fehnker, Z. Han, J Ouaknine, O. Stursberg, and M. Theobald. Abstraction and Counterexample-guided Refinement in Model Checking of Hybrid Systems. *Int. Journal of Foundations of Computer Science*, Vol. 14, No. 4, pp. 583–604, 2003.

[14] S. Graf and H. Saidi. Construction of abstract state graphs with PVS . In *Proc. of the 9th Int. Conf. on Computer Aided Verification*, Vol. 1254 of *LNCS*, pp. 72–83, June 1997.

[15] S. Das, D. L. Dill, and S. Park. Experience with Predicate Abstraction. In *Proc. of the 11th Int. Conf. on Computer Aided Verification*, Vol. 1633 of *LNCS*, pp. 160–171, 1999.

[16] J. Bengtsson and W. Yi. Timed Automata: Semantics, Algorithms and Tools. In *Lecture Notes on Concurrency and Petri Nets*, Vol. 3098, pp. 87–124, 2004.

[17] F. Wang, K. Schmidt, G. D. Huang, F. Yu, and B. Y. Wang. Formal Verification of Timed Systems: A Survey and Perspective. In *Proc. of the IEEE*, Vol. 92, pp. 1283–1307, 2004.

[18] R. Alur and L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, Vol. 126, No. 2, pp. 183–235, April 1994.

[19] C. Baier, B. R. Haverkort, H. Hermanns, and J.-P. Katoen. Performance Evaluation and Model Checking Join Forces. *Communications of the ACM*, Vol. 53, No. 9, pp. 76–85, September 2010.

[20] M. Kwiatkowska, G. Norman, J. Sproston, and F. Wang. Symbolic model checking for probabilistic timed automata. *Information and Computation*, Vol. 205, No. 7, pp. 1027–1077, July 2007.

[21] K.G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *Software Tools for Technology Transfer*, Vol. 1, No. 1-2, pp. 134–152, October 1997.

[22] UPPAAL Model Checker. available at http://www.uppaal.com/.

[23] S. Yovine. Kronos: A Verification Tool for Real-time Systems. *Software Tools for Technology Transfer*, Vol. 1, No. 1-2, pp. 123–133, October 1997.

[24] KRONOS. available at http://www-verimag.imag.fr/DIST-TOOLS/TEMPO/kronos/index-english.html.

[25] D.A. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, 2002.

[26] T. Herman. Probabilistic Self-stabilization. *Information Processing Letters*, Vol. 35, No. 2, pp. 63–67, June 1990.

[27] PRISM Manual. http://www.prismmodelchecker.org/manual/.

[28] M. Kwiatkowska, G. Norman, and J. Sproston. Probabilistic Model Checking of the IEEE 802.11 Wireless Local Area Network Protocol. In *Proc. of the 2nd Joint Int. Workshop on Process Algebra and Performance Modelling and Probabilistic Methods in Verification (PAPM/PROBMIV'02)*, Vol. 2389 of *LNCS*, pp. 169–187, July 2002.

[29] M. Kwiatkowska, G. Norman, and J. Sproston. Probabilistic Model Checking of Deadline Properties in the IEEE1394 Firewire Root Contention Protocol. *Formal Aspects of Computing*, Vol. 14, No. 3, pp. 295–318, April 2003.

[30] A. Pnueli. The Temporal Logic of Programs. In *Proc. of the 18th Int. Symp. on Foundation of Computer Science (FOCS)*, pp. 46–57, 1977.

[31] E. Clarke, E. Emerson, and A. Sistla. Automatic Verification of Finite-state Concurrent Systems Using Temporal Logics. *ACM Transactions on Programming Languages and Systems*, Vol. 8, No. 2, pp. 244–263, 1986.

[32] M.Y. Vardi and P. Wolper. Reasoning about Infinite Computations. *Information and Computation*, Vol. 115, No. 1, pp. 1–37, November 1994.

[33] R. Alur, C. Courcoubetis, and D. L. Dill. Model-Checking for Real-Time Systems. In *Proc. of the 5th Annual Symposium on Logic in Computer Science*, pp. 414–425. IEEE, 1990.

[34] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-time Systems. *Information and Computation*, Vol. 111, No. 2, pp. 193–244, June 1992.

[35] H. Hansson and B. Jonsson. A Logic for Reasoning about Time and Probability. *Formal Aspects of Computing*, Vol. 6, No. 5, pp. 512–535, September 1994.

[36] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying Continuous Time Markov Chains. In *Proc. of the 8th Int. Conf. on Computer Aided Verification (CAV'96)*, Vol. 1102 of *LNCS*, pp. 269–276, July 1996.

[37] H. Nakajima and Y. Kameyama. Improvement on Real-Time Model Checking using Abstraction-Refinement (In Japanese). *Transactions of Information Processing Society of Japan*, Vol. 45, No. SIG12 (PRO23), pp. 11–24, 2004.

[38] S. Kemper and A. Platzer. SAT-based Abstraction Refinement for Real-time Systems. In *Proc. of the Third Int. Workshop on Formal Aspects of Component Software*, Vol. 182, pp. 107–122, 2006.

[39] H. Dierks, S. Kupferschmid, and K G. Larsen. Automatic Abstraction Refinement for Timed Automata. In *Proc. of the 5th Int. Conf. on Formal Modelling and Analysis of Timed Systems*, Vol. 4763, pp. 114–129, 2007.

[40] R. Alur, R.K. Brayton, T.A. Henzinger, S. Qadeer, and S.K. Rajamani. Partial-Order Reduction in Symbolic State-Space Exploration . *Formal Methods in System Design*, Vol. 18, No. 2, pp. 97–116, March 2001.

[41] J. Bengtsson, B. Jonsson, J. Lilius, and W. Yi. Partial Order Reductions for Timed Systems . In *Proc. of the 9th Int. Conf. on Concurrency Theory (CONCUR'98)*, Vol. 1466 of *LNCS*, pp. 485–500, September 1998.

[42] E.A. Emerson and A.P. Sistla. Symmetry and Model Checking. *Formal Methods in System Design*, Vol. 9, No. 1-2, pp. 105–131, August 1996.

[43] M. Hendriks, G. Behrmann, K. Larsen, P. Niebert, and F. Vaandrager. Adding Symmetry Reduction to UPPAAL. In *Proc. of the 1st Int. Conf. on Formal Modeling and Analysis of Timed Systems (FORMATS 2003)*, Vol. 2791 of *LNCS*, pp. 46–59, September 2003.

[44] M. Kwiatkowska, G. Norman, and D. Parker. Symmetry Reduction for Probabilistic Model Checking . In *Proc. of the 18th Int. Conf. on Computer Aided Verification (CAV 2006)*, Vol. 4144 of *LNCS*, pp. 234–248, August 2006.

[45] M. Kwiatkowska, G. Norman, and J. Sproston. Performance Analysis of Probabilistic Timed Automata Using Digital Clocks. *Formal Methods in System Design*, Vol. 29, No. 1, pp. 33–78, August 2006.

[46] M. Kattenbelt, M. Kwiatowska, G. Norman, and D. Parker. A Game-based Abstraction-Refinement Framework for Markov Decision Processes. *Int. Journal on Formal Methods in System Design*, Vol. 36, No. 3, pp. 246–280, September 2010.

[47] A. Morimoto, R. Komagata, and S. Yamane. Probabilistic Timed CEGAR (In Japanese). In *IEICE Technical Report*, Vol. 109, pp. 25–30, June 2009.

[48] E. M. Clarke, A. Donzé, and A. Legay. On Simulation-Based Probabilistic Model Checking of Mixed-Analog Circuits. *Formal Methods in System Design*, Vol. 36, No. 2, pp. 97–113, June 2010.

[49] S. Tschirner, L. Xuedong, and Y. Yi. Model-Based Validation of QoS Properties of Biomedical Sensor Networks. In *Proc. of the 8th ACM Int. Conf. on Embedded Software*, pp. 69–78, October 2008.

[50] Object Management Group: Unified Modeing Language Specification version 2.1. available at http://www.omg.org/.

[51] B. Bordbar, J. Derrick, and A. G. Waters. A UML Approach to the Design of Open Distributed Systems. *Formal Methods and Software Engineering*, Vol. 2495, pp. 561–571, November 2002.

[52] R. Staehli, F. Eliassen, J. Aagedal, and G. Blair. Quality of Service Semantics for Component-Based Systems. In *Proc. of the 2nd Int. Workshop on Reflective and Adaptive Middleware Systems*, pp. 153–157, October 2003.

[53] E. Nagai, A. Makidera, K. Okano, and K. Taniguchi. A Method to Develop Distributed Real–Time Applications Based on UML/OCL (in Japanese). *IEICE Transactions on Information and Systems*, Vol. J89-D, No. 4, pp. 683–692, April 2006.

[54] L. Aceto, P. Bouyer, A. Burgue no, and K. G. Larsen. The Power of Reachability Testing for Timed Automata . In *Proc. of the 18th Int. Conf. on Foundations of Software Technology and Theoretical Computer Science*, Vol. 1530 of *LNCS*, pp. 245–256, December 1998.

[55] H. Bowman, G. Faconti, and M. Massink. Specification and Verification of Media Constraints Using UPPAAL. In *Proc. of the 5th Eurographics Work-*

shop on the Design, Specification and Verification of Interactive Systems, pp. 261–277, June 1998.

[56] R.Alur. *Techniques for Automatic Verification of Real-time Systems*. PhD thesis, Stanford University, 1991.

[57] A. David, J. Hakansson, K G. Larsen, and P. Pettersson. Model Checking Timed Automata with Priorities using DBM Subtraction. In *Proc. of the 4th Int. Conf. on Formal Modelling and Analysis of Timed Systems (FORMATS 2006)*, Vol. 4202 of *LNCS*, pp. 128–142, 2006.

[58] M. Lindahl, P. Pettersson, and W. Yi. Formal Design and Analysis of a Gear Controller . *Int. Journal on Software Tools for Technology Transfer (STTT)*, Vol. 3, No. 3, pp. 353–368, August 2001.

[59] C. Derman, editor. *Finite-State Markovian Decision Processes*. New York: Academic Press, 1970.

[60] D. P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, 1995.

[61] H. Aljazzar and S. Leue. Directed Explicit State-Space Search in the Generation of Counterexamples for Stochastic Model Checking. *IEEE Transactions on Software Engineering*, Vol. 36, No. 1, pp. 37–60, January 2010.

[62] The Network Simulator - ns-2. available at http://www.isi.edu/nsnam/ns/.

[63] E. J. Kim, K. H. Yum, and C. R. Das. Calculation of Deadline Missing Probability in a QoS Capable Cluster Interconnect. In *Proc. of the IEEE Int. Symp. on Network Computing and Applications (NCA'01)*, p. 36, October 2001.

[64] R. Alur, T. A. Henzinger, and M. Y. Vardi. Parametric Real-time Reasoning. In *Proc. of the 25th ACM Annual Symp. on the Theory of Computing (STOC'93)*, pp. 592–601, May 1993.

[65] R. Rejaie, M. Handley, and D. Estrin. RAP: An End-to-end Rate-based Congestion Control Mechanism for Realtime Streams in the Internet. In *Proc. of IEEE INFOCOM 1999*, Vol. 3, pp. 1337–1345, March 1999.

[66] I. Rhee, V. Ozdemir, and Y. Yi:. TEAR: TCP Emulation at Receivers – Flow Control for Multimedia Streaming. Technical report, NCSU, 2000.

[67] M. Handley, S. Floyd, J. Padhye, and J. Widmer. TCP Friendly Rate Control (TFRC): Protocol Specification. RFC 3448 (Proposed Standard), January 2003. Obsoleted by RFC 5348.

[68] Y. Taniguchi, A. Ueoka, N. Wakamiya, M. Murata, and F. Noda. Implementation and Evaluation of Proxy Caching System for MPEG-4 Video Streaming with Quality Adjustment Mechanism. In *Proc. of the 5th Association of East Asian Research Universities Workshop on Web Technology*, pp. 27–34, October 2003.

[69] D. Mahrenholz and S. Ivanov. Real-Time Network Emulation with ns-2. In *Proc. of the 8th IEEE Int. Symp. on Distributed Simulation and Real-Time Applications*, IEEE Computer Society, pp. 29–36, October 2004.

[70] UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms. Request for Proposal, available at http://www.omg.org.

[71] B. Bordbar and K. Okano. Verification of Timeliness QoS Properties in Multimedia Systems. In *Proc. of the 5th International Conference on Formal Engineering Methods (ICFEM '03)*, Vol. 2885 of *LNCS*, pp. 523–540, November 2003.

[72] A. David and M. O. Möller. From HUPPAAL to UPPAAL: Translation from Hierarchical Timed Automata to Flat Timed Automata. Technical Report RS-01-11, BRICS, 2001.

[73] K. Havelund, A. Skou, K. G. Larsen, and K. Lund. Formal Modelling and Analysis of an Audio/Video Protocol: An Industrial Case Study Using UPPAAL. In *Proc. of the 18th IEEE Real-Time Systems Symposium*, pp. 2–13, December 1997.

[74] D. Akehurst, J. Derrick, and A.G. Waters. Design and Verification of Distributed Multi-media Systems. In *Proc. of the 6th IFIP WG 6.1 Int. Conf. on Formal Methods for Open Object-Based Distributed Systems*, Vol. 2884 of *LNCS*, pp. 276–292, November 2003.

[75] E. Abraham, N. Jansen, R. Wimmer, J-P. Katoen, and B. Becker. DTMC Model Checking by SCC Reduction. In *Proc. of the 7th Int. Conf. on Quantitative Evaluation of Systems (QEST'10)*, IEEE Computer Society, pp. 37–48, September 2010.