MODEL CHECKING STOCHASTIC SYSTEMS IN PAT

SONG SONGZHENG

NATIONAL UNIVERSITY OF SINGAPORE

2013

MODEL CHECKING STOCHASTIC SYSTEMS IN PAT

SONG SONGZHENG

(BEng., Tianjin Univeristy (China), 2009)

A THESIS SUBMITTED FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

NUS GRADUATE SCHOOL FOR INTEGRATIVE SCIENCES AND ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2013

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.
This thesis has also not been submitted for any degree in any university previously.

宋嵩峥

Song Songzheng
15 August 2013

Acknowledgements

This thesis would not be possible without the help of many kind people around me, to only some of whom it is possible to give particular mention here.

First of all, I really appreciate the help of my supervisor Dr. Dong Jin Song, whose kindness begins before I came to Singapore. I still remember Dr. Dong encouraged me to apply for NGS scholarship in NUS, and gave me the chance to pursue my PhD here. His continuous suggestions and constant encouragement eliminate my doubts and anxiety during my PhD study. Without his various support, I would not have completed the writing of this thesis.

Furthermore, I would like to thank my mentors: Dr. Sun Jun and Dr. Liu Yang. They help me to decide my PhD topic soon after I arrived, which is very important for me to find the right track quickly. Their academic vision and timely discussions always inspire me from time to time.

In addition, I would like to acknowledge the support of my thesis advisory committee chair: Dr. Joxan Jaffar for his participation and constructive comments on my research.

To my labmates, thank you so much for your support and friendship through my PhD study, and this journey with you will be my precious memory.

I would like to thank my parents and my younger brother, for their continuous love and encouragement for letting me go further and further, both in distance and my achievements.

Last, but by no means least, many special thanks go to my fiancee Nina Lu. I appreciate her company, support and trust during the last years. Her patience and thoughtfulness get me where I am today.

Contents

Summary

Stochastic systems are useful in modeling real-world complicated systems. Probabilistic model checking is an important approach for automatic verification of stochastic systems. However, this approach faces various challenges. Previous work on specifying and verifying stochastic systems relies on simple modeling languages. Reasoning about complicated stochastic systems however requires not only efficient verification algorithms but also expressive modeling languages. Moreover, it is worthwhile to apply probabilistic model checking approach in specific domains to benefit their analysis. In this thesis, we focus on designing new modeling languages which capture the characteristics of stochastic systems, proposing optimized model checking algorithms, and applying these techniques in analyzing multi-agent systems.

First, we propose a formal model language PCSP# to specify and verify discrete probabilistic systems. PCSP# supports hierarchical structure, shared variables, concurrency and probability. In order to capture full nondeterminism and probability, the semantic model of PCSP# is Probabilistic Automata (PA). We develop a verification engine for PCSP# to support reachability checking, Linear Temporal Logic (LTL) checking, reward checking and trace refinement checking. Here a refinement relationship (with probability) is from a PCSP# model representing a system and a non-probabilistic model representing properties. Meanwhile, two optimizations are used to speed up the verification. We show that trace refinement checking can be used to verify complex LTL *safety* properties. In this case, original automata-based LTL checking is avoided, and the verification of such properties is faster. In addition, anti-chain based approach can be used to further increase the efficiency of the refinement checking.

Second, we use PCSP# to model and verify multi-agent systems to demonstrate the expressiveness and effectiveness of our approaches. Particularly, two representing scenarios are investigated: robustness of negotiation strategies and dynamics of dispersion game. Their characteristics are well captured by PCSP#, and desired properties are supported either by our existing approaches, or specific designed algorithms. Moreover, counter abstraction technique is used in the modeling and verification of these cases, so that the state space explosion problem can be tackled to some extent.

Third, many stochastic systems are described by Discrete-time Markov Chain (DTMC) instead of PA due to their lack of nondeterminism, such as the dispersion game mentioned above. Therefore, we develop a novel divide-conquer approach to speed up reachability analysis in DTMC. Reachability analysis is used to decide the probability of reaching certain disastrous state in a DTMC, and traditional methods for calculating reachability probability have their drawbacks in scalability or efficiency. One source of the low efficiency is the existence of loops in a DTMC. Therefore, we propose to divide the whole state space of a DTMC into several partitions, and abstract them individually. This divide-and-abstract can be repeated iteratively to eliminate loops. Afterwards, the remaining acyclic DTMC can be solved efficiently via value iteration method.

Last but not least, we extend PCSP# to supported real-time characteristics since timing constraints exist widely. Another formal modeling language called PRTS is proposed for hierarchical probabilistic real-time systems. Based on PCSP#, PRTS introduces timed process constructors such as *within* and *deadline*. However, dense-time semantics in PRTS generates infinite number of states. To tackle this issue, zone abstraction is used to construct a finite-state PA from PRTS, which is subject to model checking. Furthermore, we develop a method to model check PRTS models with the assumption of non-Zenoness, which is known to be conflicting with zone abstraction.

All approaches proposed in this thesis are integrated in our home-grown verification framework PAT, which has user friendly editor, simulator and verifier. PCSP# and PRTS are developed as two modules in PAT, focusing on stochastic systems without/with timing constraints respectively. Meanwhile, the experimental results show the applicability and efficiency of our approaches.

Key words: **Stochastic Systems, Real-time Systems, Formal Verification, Probabilistic Model Checking, Reachability Analysis, Multi-agent Systems, PAT**

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction and Overview

Stochastic systems are common in practice. Different from concurrent systems, stochastic systems have probabilistic characteristics in their behaviors, which means some behaviors follow specific probabilistic distributions. This kind of systems widely exist in many domains, from communication protocols to biology systems [48, 83, 50, 63, 64, 77]. For example, in the randomized leader election protocol [71], multiple processes want to elect one leader. Each process will first randomly choose a natural number from a specific range as its id. The process with a unique highest id will be elected as a leader. If several processes have the same highest id, the selection procedure will repeat. The uniform distribution is necessary for each process picking an id, therefore probabilistic behaviors exist in this election system. Because of the wide existence of stochastic systems, their correctness is critical.

As an automatic verification technique, model checking [37, 18] has been applied to a variety of domains from hardware to software, and from concurrent systems to stochastic systems. In concurrent systems, people always require them be absolutely correct without any failure. However, it is meaningful to guarantee that a stochastic system behaves as desired with a certain probability. For example, for a real message channel with environment noise, it is acceptable that this channel can transfer message successfully with 99%. As a result, probabilistic verification aims at different targets compared with traditional model checking.

In a nutshell, probabilistic model checking is a systematic way of analyzing finite-state probabilistic systems. Given a finite-state model of a probabilistic system and a property, a probabilistic model checker calculates the (range of) probability that the model satisfies the property. There have been a number of probabilistic model checkers and corresponding algorithms. Some of these tools are used to model and verify various systems, and the results are promising. However, there are still some limitations existing in current stochastic systems verification, which are summarized as follows.

- Existing probabilistic model checkers have been designed for hierarchically simple systems. For instance, the state-of-the-art probabilistic model checker PRISM [80] supports a simple state-based language, based on the Reactive Modules formalism of Alur and Henzinger [11]. The MRMC checker supports a rather simple input language too [72]. The input language of the LiQuor checker [35], named Probmela, is based on an extension of Promela supported by the SPIN model checker. None of the above tools supports analysis of hierarchical complex probabilistic systems, therefore some complicated systems cannot be verified efficiently.
- Existing fundamental probabilistic verification algorithms are not optimal in all settings. Linear Temporal Logic (LTL) verification in Probabilistic Automata (PA) [18] is based on automata-theoretic approach, which is complicated and unnecessary in some cases; reachability analysis in Discrete-time Markov Chains (DTMCs) always applies value iteration method and may confront the slow convergence problem. Therefore optimizations of these algorithms are necessary.
- Although model checking approach has been applied to some other domains, e.g., biological systems [63, 64, 77], more effort should be done to widen its application. Multi-agent systems (MASs) are widely used to model system composed by different parties, and their formal verification should be paid much attention [134]. However, not that much work has been done on applying model checking techniques in MAS, especially for MAS with probabilistic dynamics.
- Few existing works focus on formal verification of probabilistic real-time systems. UPPAAL [23] supports real-time, concurrency and recently data operations as well as probability (in the extension named UPPAAL-PRO), but lacks support for hierarchical control flow and is limited to maximal probabilistic reachability checking. PRISM [80] supports the verification of Probabilistic Timed Automata (PTA), which combines real-time and probability. However, it does not support hierarchical systems, but rather networks of flat finite state systems. In addition, most of the tools support only simple data operations, which could be insufficient in modeling systems with

complicated structures and complex data operations that are common in real-life cases.

To tackle these limitations, in this thesis, we are aiming at automatic and systematic methods to verify hierarchical stochastic systems with/without timing requirements. Expressive modeling languages and efficient verification algorithms are designed to make our work benefit to this domain. Moreover, we apply our approach in MAS to analyze its dynamics and generate promising outputs for MAS community.

1.1 Summary of This Thesis

In this section, we briefly introduce the scope of this thesis.

First, we develop a model checker for verifying hierarchical complex probabilistic systems. A language called PCSP# is proposed for stochastic system modeling. It is an expressive language, combining Hoare's CSP [69], data structures, and probabilistic choices. It extends previous work on combining CSP with probabilistic choice [94] or on combining CSP with data structures [113]. PCSP# combines low-level programs, e.g., sequence programs defined in a simple imperative language or any C# program, with high-level specifications (with process constructs like parallel, choice, hiding, etc.), as well as probabilistic choices. It supports shared variables as well as abstract events, making it both state-based and event-based. The semantic model of PCSP# is Probabilistic Automata (PA). We have implemented PCSP# model checker in PAT model checking framework.

In order to increase the verification efficiency of PCSP# models, we propose two optimized algorithms.

- We show that refinement checking can be used to verify complex Linear Temporal Logic (LTL) *safety* properties. Here a refinement relationship (with probability) is from a PCSP# model representing a system and a non-probabilistic model representing properties. In this case, original automata-based LTL checking in PA is avoided, and the verification of such properties is faster.
- Due to the potential nondeterminism in the specification, refinement checking often relies on the classic subset construction approach. Therefore, we show that anti-chain

can be used to speed up the refinement checking between a probabilistic implementation and a non-probabilistic specification.

Next, focusing on widening the applications of our approach, we apply model checking techniques in MAS domain to analyze its dynamics. Two MAS scenarios are taken into consideration: *robustness* of negotiation strategies, and dynamics of *dispersion games* [120]. According to the characteristics of these two cases, different semantic models are used to capture their behaviors. Since no stochastic behaviors exist in robustness analysis in our setting, Labeled Transition System (LTS) is applied to model the negotiation systems. On the contrary, we show that Discrete-time Markov Chain (DTMC) is suitable for representing dispersion games. Because LTS and DTMC can be viewed as specific PAs, PCSP# has the capability to model both systems. Meanwhile, counter abstraction technique is used to reduce the state space of both MAS models due to the symmetric property existing in the systems, thus making the analysis using model checking techniques both feasible and efficient. Further, for specific properties such as robustness requirements, we have designed dedicated verification algorithms to fulfill the verification.

According to our experience of DTMC systems, e.g., dispersion games, we propose a divide and conquer approach to improve reachability analysis in DTMC. Traditional methods of reachability analysis in DTMC have their drawbacks. For example, value iteration may confront slow convergence problem when huge loops exist in DTMC. Our approach partitions the state space of a DTMC and abstracts each group individually. Loops can be eliminate afterwards, therefore the existing slow convergence problem can be solved to some extent.

Finally, because in real-life cases many stochastic systems have timing requirements, we develop a model checker for probabilistic real-time systems based on PCSP#. A modeling language called PRTS is defined, which captures the behaviors of systems with stochastic dynamics, timing requirements and hierarchical control flows. The semantic model of PRTS is also PA. Because PRTS has dense-time semantics, there are potentially infinite number of states in corresponding PA. To tackle this issue, we use dynamic zone abstraction approach to generate finite-state abstract PA. Furthermore, we develop a method to model check PRTS models with the assumption of non-Zenoness, which is known to be conflicting with zone abstraction. This approach is also implemented in PAT.

1.2 Thesis Structure

This thesis has 7 chapters in total. The remaining chapters are structured as follows.

Chapter 2 recalls the preliminary knowledge which are fundamental in this thesis. In this chapter, we first introduce the modeling formalisms used in our approach: Probabilistic Automata (PA), Discrete-time Markov Chain (DTMC) and Labelled Transition System (LTS). The first two both support probabilistic choices, while PA also captures full nondeterminism. LTS is for non-probabilistic systems, which supports concurrency. Second, a widely used temporal logic: Linear Temporal Logic (LTL) is introduced. Third, we explain reachability analysis and LTL verification in PA. Relative algorithms such as value iteration method and automata-based approach are briefly presented. Lastly, since all model checkers and corresponding algorithms proposed by this thesis are implemented in PAT model checking framework, we introduce this toolkit.

Chapter 3-6 are the main content of this thesis, and they have the similar following structure. First we give specific introduction to the content of this chapter. Then specific preliminary knowledge (if there is any) for this chapter is followed. Next, we discuss the main content of this chapter with experimental results. Related work is presented in the end.

Chapter 3 introduces our model checker for probabilistic systems. First, The syntax and operational semantics of language PCSP# are formally defined. Further, we prove that LTL safety properties can be verified via trace refinement checking between a PCSP# model and a non-probabilistic specification, therefore the verification efficiency of corresponding properties can be increased. Moreover, we show that anti-chain approach can be used to speed up the mentioned trace refinement checking.

Chapter 4 introduces the application of our model checking approach in analyzing dynamics of multi-agent systems. First, we use traditional model checking approach to check the robustness of negotiation strategies in MAS. Later, probabilistic model checking is used to analyze the stochastic behaviors of a multi-learner system, in particular, a scenario called dispersion game. Convergence, deviation, and convergence rate are calculated in the model of this game.

Chapter 5 introduces the divide and conquer approach to improve the reachability analysis in DTMC. We show that traditional methods have their drawbacks, e.g., slow convergence problem in value iteration approach. Then we present the state space of a DTMC can be partitioned to small groups, each of which can be abstracted individually. After iterative partitioning and abstraction, the resulting DTMC is acyclic which can be verified efficiently.

Chapter 6 introduces our model checker for probabilistic real-time systems. First, the syntax and semantics of language PRTS are formally defined. Further, dynamic zone abstraction is used to generated finite-state abstract PA which is subject to model checking. Moreover, we develop an algorithm to model check PRTS models against LTL properties with non-Zenoness assumption.

Chapter 7 concludes this thesis with some further directions of research.

1.3 Acknowledgement of Published Work

Most of the work presented in this thesis has been published in international conference proceedings.

- **Model Checking Hierarchical Probabilistic Systems** [119]. This paper was published at the 12th International Conference on Formal Engineering Methods (ICFEM 2010). The work is presented in Chapter 3.
- **More Anti-chain Based Refinement Checking** [132]. This paper was published at the 14th International Conference on Formal Engineering Methods (ICFEM 2012). The work is presented in Chapter 3.
- **Probabilistic Model Checking Multi-agent Behaviors in Dispersion Games Using Counter Abstraction** [60]. This work was published at the 15th International Conference on Principles and Practice of Multi-Agent Systems (PRIMA 2012). The work is presented in Chapter 4.
- **Improved Reachability Analysis in DTMC via Divide and Conquer** [119]. This paper was published at the 10th International Conference on integrated Formal Methods (iFM 2013). The work is presented in Chapter 5.
- **PRTS: An Approach for Model Checking Probabilistic Real-Time Hierarchical Systems** [118]. This paper was published at the 13th International Conference on Formal Engineering Methods (ICFEM 2011). Its short version is published in 24th Computer Aided Verification (CAV 2012) as a tool demonstration paper. The work is presented in Chapter 6.

Moreover, the work related to apply model checking approach in robustness analysis of negotiation strategies, which is presented in Chapter 4, has been submitted for publication.

For all these publications, I have substantial contributions in both theory and implementation.

Chapter 2

Preliminaries

In this chapter, we define some general and fundamental notations and concepts used in our work. First, several modeling formalisms are introduced. As the semantic model of our language, Probabilistic Automata (PA) [110] is presented in details; meanwhile, Discrete-time Markov Chain (DTMC) is also presented since it is critical to define related concepts in PA; in addition, Labeld Transition System (LTS) is introduced because of its significance in modeling concurrent systems. Further, the syntax and semantics of Linear Temporal Logic (LTL) is presented, followed by the introduction of reachability checking and LTL checking in PA. Moreover, we introduce PAT model checking framework, which is the fundamental toolkit for our model checker and algorithms. Other concepts will be introduced in later chapters where they are relevant.

2.1 Modeling Formalisms

2.1.1 Probabilistic Automata

When modeling probabilistic systems (particularly, discrete-time stochastic control processes), PA is one of the popular models since it supports probabilistic choices and full nondeterminism. A PA is a directed graph whose transitions are labeled with events or probability. The following notations are used to denote different transition labels. τ denotes an unobservable event; Act denotes the set of observable events such that $\tau \notin Act$; a special event $\checkmark \in Act$ indicates the termination of a process; Act_τ denotes $Act \cup \{\tau\}$. Given

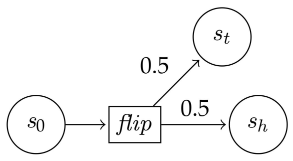


Figure 2.1: Transitions Representing a Fair Coin Flip

a set of states S, a distribution is a function $\mu : S \to [0, 1]$ such that $\Sigma_{s \in S}\, \mu(s) = 1$. μ is a *trivial* distribution or is trivial if and only if there exists a state $s \in S$ such that $\mu(s) = 1$. Let $Distr(S)$ be the set of all distributions over S. Formally, we have the following definition.

Definition 1 *A PA is a tuple $\mathcal{D} = (S, s_{init}, Act, Pr, AP, L)$ where S is a countable set of states; $s_{init} \in S$ is the initial state*[1]*; $Pr \subseteq S \times Act_\tau \times Distr(S)$ representing the transition relation, i.e., states in PA can reach different distributions via the same action; AP is a set of atomic propositions and L: $S \to 2^{AP}$ is a labeling function.*

A PA $\mathcal{D}$ is *finite* if and only if S and $Distr(S)$ are finite. **In this thesis, we just focus on finite PA.** For simplicity, a transition is written as $s \xrightarrow{x} \mu$ such that $s \in S$; $x \in Act_\tau$ and $\mu \in Distr(S)$. If μ is trivial, i.e., $\exists\, s' \in S$ satisfying $\mu(s') = 1$, the transition can be simplified as $s \xrightarrow{x} s'$. One example of transitions in PA is demonstrated in Fig. 2.1. In this example, one can 'flip' a fair coin, generating equal probabilities to 'head' (s_h) and 'tail' (s_t). Here the rectangle represents the action, and circles represent states in the system. $(s_0, \mathit{flip}, \mu)$ is in the transition relation, where $\mu(s_t) = \mu(s_h) = 0.5$.

There are two kinds of transition labels in our setting. An observable transition is labeled with an action in Act. An un-observable transition is labeled with τ. An action x is *enabled* in state s if and only if $\exists\, \mu \in Distr(S), (s, x, \mu) \in Pr$. Given a state s, let $Act(s) = \{(x, \mu) \mid (s, x, \mu) \in Pr\}$. ; state s' is called an *successor* of s if and only if $\exists (x, \mu) \in Act(s)$ satisfying $\mu(s') > 0$. $Pre(s')$ is defined as $\{s \mid s'$ is successor of $s\}$, which are pre-states of s'. Given a set of states C, $Pre(C) = \{s \mid \exists\, c \in C, s \in Pre(c)\}$. An infinite *path* in $\mathcal{D} = (S, s_{init}, Act, Pr, AP, L)$ is an infinite sequence $\langle s_0, x_1, \mu_1\, s_1, x_2, \mu_2, s_2, x_3, \mu_3 \cdots \rangle \in (S \times Act \times Distr(S))^\omega$, which can be denoted as

$$\pi = s_0 \xrightarrow{x_1, \mu_1} s_1 \xrightarrow{x_2, \mu_2} s_2 \xrightarrow{x_3, \mu_3} \cdots.$$

[1]Without loss of generality, we assume there is only one initial state in the system.

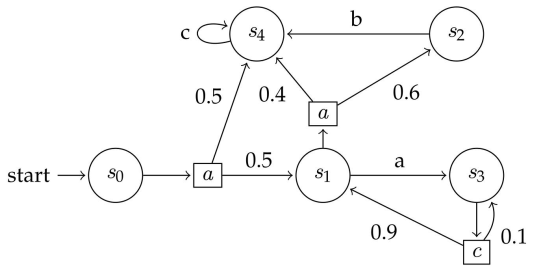


Figure 2.2: A PA Example

such that $\forall\, i \geq 0, (s_i, x_{i+1}, \mu_{i+1}) \in Pr \wedge \mu_{i+1}(s_{i+1}) > 0$. A corresponding infinite *trace* of π is denoted as $\rho = s_0 s_1 s_2 \cdots$. Any finite prefix of π (or ρ) that ends in a state is a finite path (or trace). $Paths(s)$ (or $Traces(s)$) denotes the set of infinite paths (or traces) that start in state s; $Paths_{fin}(s)$ (or $Traces_{fin}(s)$) denotes the set of finite paths (or traces) that start in s. A path is rooted if it starts with s_{init}. Hereafter, by traces and paths we mean rooted traces and paths unless otherwise stated.

In order to verify temporal logic, we recall the definition of Maximal End Components (MEC) [18]. First, sub-PA is defined as follows.

Definition 2 *Let $\mathcal{D} = (S, s_{init}, Act, Pr, AP, L)$ be a PA. A* sub-PA *of $\mathcal{D}$ is a pair* (T, A) *where $\emptyset \neq T \subseteq S$ and $A \subseteq T \times Act \times Distr(S)$ is a relation such that: 1) for all states $s \in T$, there exist $x \in Act$ and $\mu \in Distr(S)$ satisfying $(s, x, \mu) \in A$ and 2) $(s, x, \mu) \in A$ implies $\{t \in S \mid \mu(t) > 0\} \subseteq T$.*

An *end component* of PA $\mathcal{D}$ is a sub-PA (T, A) such that the graph induced by (T, A) is strongly connected, i.e., $\forall\, s, s' \in T, \exists\, p = \langle s_0, x_1, \mu_1, s_1, x_2, \mu_2, s_2, x_3, \mu_3, \cdots s_n \rangle \in path_{fin}(s_0)$ satisfying $s_0 = s \wedge s_n = s' \wedge (\forall\, i \geq 0, (s_i, x_{i+1}, \mu_{i+1}) \in A)$.

An end component (T, A) of $\mathcal{D}$ is called *maximal* if there is no end component *(T', A')* such that $(T, A) \neq (T', A') \wedge (\forall\, s \in T, T \subseteq T' \wedge A(s) \subseteq A'(s)$. A *bottom MEC* is an MEC without outgoing transitions. We write $MEC(\mathcal{D})$ to denote all MECs contained in $\mathcal{D}$. Note MECs are disjoint, in other words, one state belongs to at most one MEC.

One simple PA is demonstrated in Fig. 2.2. Note a transition following a trivial distribution is labeled with an action only. In this example, $S = \{s_0, s_1, s_2, s_3, s_4\}$; $s_{init} = s_0$; $Act = \{a, b, c\}$. $\pi_1 = s_0 \xrightarrow{a,\mu_1} s_1 \xrightarrow{a,\mu_2} s_2 \xrightarrow{b,\mu_3} s_4 \cdots \in Path(s_0)$ and $\pi_2 = s_0 \xrightarrow{a,\mu_1} s_1 \xrightarrow{a,\mu_2} s_2 \xrightarrow{b,\mu_3} s_4 \in Path_{fin}(s_0)$. $MEC(\mathcal{D}) = \{(\{s_1, s_3\}, \{(s_1, a, \mu_4), (s_3, c, \mu_5)\}), (\{s_4\}, \{(s_4, c, \mu_6)\})\}$. Here $\mu_1, \mu_3, \mu_5, \mu_6$ are obvious in Fig 2.2. μ_2 and μ_4 are both distributions from s_1 via action a. μ_2 satisfies $\mu_2(s_4) = 0.4$ and $\mu_2(s_2) = 0.6$, while μ_4 is a trivial distribution satisfying $\mu_4(s_3) = 1$.

Throughout this paper, we assume PAs are deadlock-free following the standard practice. A deadlocking PA can be made deadlock-free by adding self loops labeled with τ with probability 1 to the deadlocking states, without affecting the result of probabilistic verification.

2.1.2 Discrete-time Markov Chains

From the definition, we can find that a state of a PA $\mathcal{D}$ may have multiple outgoing actions, which means nondeterminism exists in $\mathcal{D}$. A *scheduler* for $\mathcal{D}$ is a function to resolve the nondeterminism, whose definition is $\Gamma : traces_{fin}(s_{init}) \rightarrow Act_\tau \times Distr(S)$. A scheduler is called *memoryless* if and only if for each trace $s_{init} s_1 s_2 \cdots s_n$ and $s_{init} t_1 t_2 \cdots t_m$, as long as $s_n = t_m$:

$$\Gamma(s_{init} s_1 s_2 \cdots s_n) = \Gamma(s_{init} t_1 t_2 \cdots t_m).$$

Therefore, a memoryless scheduler can be viewed as a function $\Gamma : S \rightarrow Act_\tau \times Distr(S)$, i.e., it always chooses the same action and same distribution in a given state.

Given an PA $\mathcal{D}$ and a scheduler δ, a Discrete Time Markov Chain [18] (DTMC) $\mathcal{D}^\delta$ can be defined, which just has one action and one corresponding outgoing distribution in every state. The formal definition of DTMC is as follows.

Definition 3 *A DTMC is a tuple $(S, s_{init}, Act, Pr, AP, L)$ where S is a countable set of states; $s_{init} \in S$ is the initial state; Pr is a function: $S \rightarrow Act_\tau \times Distr(S)$ representing the transition relation; AP is a set of atomic propositions and L: $S \rightarrow 2^{AP}$ is a labeling function.*

Our definition of DTMC is slightly different from the traditional one [18] since we take the actions into consideration. For each state in DTMC, there is a unique action and corresponding distribution enabled. Without loss of generality, we have the following two assumptions for DTMCs in this thesis. 1) There is only one initial state in the whole system and 2) DTMC is deadlock free.

Given a DTMC $\mathcal{D}^\delta = (S, s_{init}, Act, Pr, AP, L)$, a transition is written as $s \xrightarrow{x,p} s'$ such that $s, s' \in S$; $x \in Act_\tau$; $\exists \mu \in Distr(S)$ satisfying $Pr(s) = (x, \mu)$ and $p = \mu(s') > 0$. If μ is

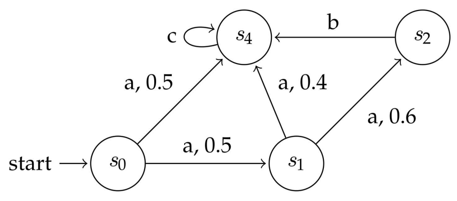


Figure 2.3: A DTMC Example

a trivial distribution, the transition can be simplified as $s \xrightarrow{x} s'$. In each transition, we denote $P(s, s') = \mu(s')$ as long as $Pr(s) = (x, \mu)$. A *path* of $\mathcal{D}^\delta$ is a finite or infinite sequence $\pi = \langle s_0, s_1, s_2, \cdots \rangle$ of states where $s_i \in S$ such that $P(s_i, s_{i+1}) > 0$ for all i. Actions and distributions are ignored in the paths since they are unique for each state. Let $Paths(\mathcal{D}^\delta, s)$ denote the set of all paths of $\mathcal{D}^\delta$ starting in state s and let $Paths_{fin}(\mathcal{D}^\delta, s)$ denote the set of all finite paths of $\mathcal{D}^\delta$ starting in s. $Paths(\mathcal{D}^\delta)$ and $Paths_{fin}(\mathcal{D}^\delta)$ are respectively used to denote all paths and finite paths in $\mathcal{D}^\delta$ starting in an arbitrary state. A state s' is called *reachable* from state s if and only if there is a finite path from s to s'. One simple DTMC is demonstrated in Fig. 2.3, which is generated from the PA in Fig. 2.2 with a **memoryless** scheduler δ satisfying $\delta(s_0) = (a, \mu_1)$; $\delta(s_1) = (a, \mu_2)$; $\delta(s_2) = (b, \mu_3)$ and $\delta(s_4) = (c, \mu_6)$. Related distributions are defined in Fig. 2.2.

A set of states $C \subseteq S$ is called *connected* in $\mathcal{D}^\delta$ iff $\forall\, s, s' \in C$, there is a finite path $\pi = \langle s_0, s_1, \cdots, s_n \rangle$ satisfying $s_0 = s \wedge s_n = s' \wedge \forall\, i \in [0, n], s_i \in C$. Strongly Connecte Components (SCCs) are those maximal sets of states which are mutually connected in a DTMC. An SCC without outgoing transitions is called *bottom SCC (BSCC)*. An SCC is called *trivial* if it just has one state without a self-loop. An SCC is *nontrivial* iff it is not trivial. A DTMC is *acyclic* iff it only has *trivial* SCCs. Note that one state can only be in one SCC. In other words, SCCs are disjoint. Take the DTMC in Fig. 2.3 as an example. It just has one non-trivial SCC: $\{s_4\}$, and it is a BSCC.

The *cylinder set* of a finite path π of $\mathcal{D}^\delta$ is defined as $Cyl(\pi) = \{\pi' \in Paths(\mathcal{D}^\delta) \mid \pi'$ is infinite and π is a prefix of $\pi'\}$. The probability of the cylinder sets denoted as $P^\delta_{\mathcal{D}}$ is given by

$$\mathcal{P}^\delta_{\mathcal{D}}(Cyl(s_0 \cdots s_n)) = \Pi_{i=0}^{n-1} P(s_i, s_{i+1}).$$

For finite paths $\pi \in Paths_{fin}(\mathcal{D}^\delta, s_0)$ we set $\mathcal{P}^\delta_{\mathcal{D}_{fin}}(\pi) = \mathcal{P}^\delta_{\mathcal{D}}(Cyl(\pi))$. For a set of paths $A \in Paths_{fin}(\mathcal{D}^\delta, s)$ we define $\mathcal{P}^\delta_{\mathcal{D}_{fin}}(A) = \Sigma_{\pi \in A'} \mathcal{P}^\delta_{\mathcal{D}_{fin}}(\pi)$ with $A' = \{\pi \in A \mid \forall\, \pi' \in A, \pi'$ is not a prefix of $\pi\}$. Similarly, if π is infinite, then $\mathcal{P}^\delta_{\mathcal{D}}(\pi) = \Pi_{i=0}^{\infty} P(s_i, s_{i+1})$. For a set of paths $A \in Paths(\mathcal{D}^\delta, s)$ we define $\mathcal{P}^\delta_{\mathcal{D}}(A) = \Sigma_{\pi \in A} \mathcal{P}^\delta_{\mathcal{D}}(\pi)$. Note that for an infinite path set the

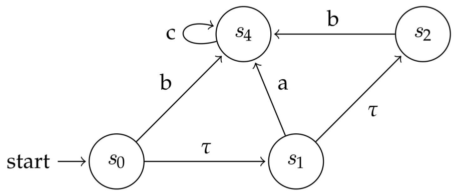


Figure 2.4: An LTS Example

definition may involve an infinite sum, but it always defines a probability mass between 0 and 1.

2.1.3 Labeled Transition System

Labeled transition system is a semantic formalism widely used in concurrent systems, in which states are labeled with atomic propositions and transitions are labeled with actions.

Definition 4 *A Labeled Transition System (LTS) $\mathcal{L}$ is a tuple $(S, s_{init}, Act, T, AP, L)$ where S is a finite set of states; and $init \in S$ is an initial state; Act is an alphabet; $T \subseteq S \times Act \times S$ is a labeled transition relation; AP is a set of atomic propositions and L: $S \to 2^{AP}$ is a labeling function.*

A transition label can be either a visible event or an invisible one (which is referred to as τ). A τ-transition is a transition labeled with τ. For simplicity, we write $s \xrightarrow{e} s'$ to denote $(s, e, s') \in T$. If $s \xrightarrow{e} s'$, then we say that e is enabled at s. Let $s \rightsquigarrow s'$ to denote that s' can be reached from s via zero or more τ-transitions; we write $s \overset{e}{\rightsquigarrow} s'$ to denote there exists s_0 and s_1 such that $s \rightsquigarrow s_0 \xrightarrow{e} s_1 \rightsquigarrow s'$. A path of $\mathcal{L}$ is a sequence of alternating states/events $\pi = \langle s_0, e_0, s_1, e_1, \cdots \rangle$ such that $s_0 = init$ and $s_i \xrightarrow{e_i} s_{i+1}$ for all i. The set of path of $\mathcal{L}$ is written as $paths(\mathcal{L})$. Given a path π, we can obtain a sequence of visible events by omitting states and τ-events. The sequence, written as $trace(\pi)$, is a trace of $\mathcal{L}$. The set of traces of $\mathcal{L}$ is written as $traces(\mathcal{L}) = \{trace(\pi) \mid \pi \in paths(\mathcal{L})\}$.

A set of states $C \subseteq S$ is called *connected* in $\mathcal{L}$ iff $\forall\, s, s' \in C$, there is a finite path $\pi = \langle s_0, e_0, s_1, e_1 \cdots, s_n \rangle$ satisfying $s_0 = s \wedge s_n = s' \wedge \forall\, i \in [0, n], s_i \in C$. Strongly Connecte Components (SCCs) are those maximal sets of states which are mutually connected in an LTS. An SCC without outgoing transitions is called *bottom SCC (BSCC)*. An SCC is called *trivial* if it just has one state without a self-loop. An SCC is *nontrivial* iff it is not trivial. SCCs in LTS are also disjoint. Take the LTS in Fig. 2.4 as an example. $s_0 \xrightarrow{b} s_4$ and $s_0 \overset{a}{\rightsquigarrow} s_4$. Meanwhile, it just has one non-trivial SCC: $\{s_4\}$, and it is a BSCC.

$$\begin{tikzcd}
 & s_4 \arrow[loop left, "c"] & & s_2 \arrow[ll, "b"'] \\
\text{start} \arrow[r] & s_0 \arrow[u, "b"] \arrow[rr, "\tau"] & & s_1 \arrow[ull, "a"] \arrow[u, "\tau"']
\end{tikzcd}$$

2.2 State/Event Linear Temporal Logic (SE-LTL)

In this part, we introduce a widely used temporal logic: Linear Temporal Logic (LTL), which is also one main kind of properties studied in this thesis. Traditional LTL was introduced to specify the properties of executions of a system [98]. In [31], LTL is extended to build up from not only state propositions but also events[2]. The extended LTL is referred to as SE-LTL. Given a PA $\mathcal{D} = (S, s_{init}, Act, Pr, AP, L)$, an SE-LTL formula ϕ can be composed by not only atomic state propositions but also actions. The syntax is

$$\phi ::= p \mid \alpha \mid \neg\phi \mid \phi \wedge \phi \mid X\phi \mid \phi U \phi, \text{ where } p \in AP \text{ and } \alpha \in Act.$$

The semantics of SE-LTL is defined as follows.

Definition 5 *Let $\pi = \langle s_0, x_0, \mu_1\, s_1, x_1, \mu_2 \cdots \rangle$ be a path in a PA $\mathcal{D}$ and π_i the suffix of π starting at s_i. The path satisfaction relation is defined as follows:*

- $\pi \models p$ *iff* $p \subseteq L(s_0)$;
- $\pi \models \alpha$ *iff* $\alpha = x_0$;
- $\pi \models \neg\phi$ *iff* $\pi \not\models \phi$;
- $\pi \models \phi_1 \wedge \phi_2$ *iff* $\pi \models \phi_1$ *and* $\pi \models \phi_2$;
- $\pi \models X\phi$ *iff* $\pi_1 \models \phi$;
- $\pi \models \phi_1 U \phi_2$ *iff there exists* $k \geq 0$ *satisfying* $\pi_k \models \phi_2$ *and for all* $0 \leq j < k$, $\pi_j \models \phi_1$.

Informally, $\neg\phi$ means ϕ does not hold; $X\phi$ indicates ϕ should be true in next state; U means "until", i.e., $\phi_1 U \phi_2$ indicates that ϕ_1 must be true **until** ϕ_2 is true. Other properties can be extended from these basic syntax. For example, $\Diamond p$ meaning eventually p can be expressed as $true\ U\ p$, and $\Box p$ meaning "always p" can be represented as $\neg\Diamond\neg p$.

[2]Events and actions are interchangeable in this thesis

2.3 Reachablity Checking and SE-LTL Checking in PA

In this section, we recall the algorithms of reachability checking and SE-LTL (LTL for short) checking in PA. The reason that these two properties are chosen is because they play the key role in the properties specification in this thesis.

2.3.1 Reachability Checking

Reachability checking in PA indicates the computation of reachability probability from one state to another. Generally, given a PA $\mathcal{D}$ $(S, s_{init}, Act, Pr, AP, L)$ and $T \subseteq S$ as a set of target state, it is meaningful to measure the probability from other states to T. In order to decide this, DTMCs should be constructed from $\mathcal{D}$, and the reachability probabilities for each DTMC should be calculated.

Here the interest is the maximal, or dually, the minimal probability of reaching a state in T when starting in state $s \in S$. For maximal probabilities this amounts to determining

$$\mathcal{P}^{max}_{\mathcal{D}}(s \models \diamond T) = \sup_{\delta} \mathcal{P}^{\delta}_{\mathcal{D}}(s \models \diamond T)$$

There are potentially infinitely many schedulers in $\mathcal{D}$. Fortunately, theorems in [18] guarantee that for any $s \in S$, there exists a memoryless scheduler which maximizes the probabilities of reaching T. Therefore, the supremum can replaced by a maximum. Meanwhile, the number of memoryless schedulers is finite since $\mathcal{D}$ is finite.

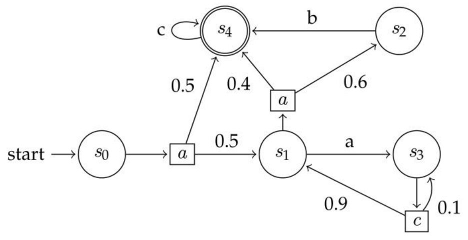
Similarly, the minimal reachability probability is defined as

$$\mathcal{P}^{min}_{\mathcal{D}}(s \models \diamond T) = \inf_{\delta} \mathcal{P}^{\delta}_{\mathcal{D}}(s \models \diamond T)$$

Again, there exists a memoryless scheduler to minimize the probability. Therefore the infimum can be replaced by a minimum.

Because all properties supported in this thesis can be reduced to reachability checking, in the following all schedulers used are assumed to be *memoryless* unless mentioned otherwise. Next, we use maximal reachability property to demonstrate how to solve reachability probabilities.

Given the transition relation of a PA, a equation system representing the transition probability from one state to another can be built. After the target states are decided, each state



$$\begin{cases} p_0 = 0.5 \times p_1 + 0.5 \times p_4 \\ p_1 = max\{0.4 \times p_4 + 0.6 \times p_2, p_3\} \\ p_2 = p_4 \\ p_3 = 0.1 \times p_3 + 0.9 \times p_1 \\ p_4 = 1 \end{cases}$$

Figure 2.5: Equation System of PA

in the equation system can be represented by a variable, which means the probability of reaching the target states from this state. Take the PA in Fig. 2.2 as an example, whose corresponding equation system is shown in Fig 2.5. Here s_4 is set to be the target state, and p_i in the equation system represents the **maximal** probability from s_i to s_4.

To solve the equation system, value iteration method [18] is popular due to its good scalability. This approach starts from the target states, and uses a backward format to update the value of the variables in the equation system step by step. Imagine we want to calculate the maximal probability from s_3 to s_4 in Fig. 2.5. Assume p_i^k is the maximal probability of s_i after the k-th iteration. Starting from the target state s_4, in k-th iteration we update the probability of states which could reach s_4 in exact k steps. Obviously, $p_i^0 = 0, 0 \leq i \leq 3$. As $p_4^k = 1$ for any k, k is ignored in this state. In the 1st iteration, p_0, p_1 and p_2 can be updated. $p_0^1 = 0.5$, $p_2^1 = 1$ and $p_1^1 = \max\{0.4 \times p_4 + 0.6 \times p_2^0, 1 \times p_3^0\} = max\{0.4, 0\} = 0.4$. In the 2nd iteration, both p_1 and p_3 can be updated. It is trivial to show $p_3^2 = 0.1 \times p_3^1 + 0.9 \times p_1^1 = 0.9$, and $p_1^2 = \max\{0.4 \times p_4 + 0.6 \times p_2^1, 1 \times p_3^1\} = max\{1, 0\} = 1$. In the 3rd iteration, only p_3 can be updated. $p_3^3 = 0.1 \times p_3^2 + 0.9 \times p_1^2 = 0.99$. Iteratively, p_3 in the long run can be calculated. A user-defined threshold is usually necessary to terminate the calculation, according to the desired precision.

2.3.2 LTL Checking

Automata-theoretic approach is used to check LTL properties in stochastic systems [18]. Given a PA $\mathcal{D}$ and an LTL formula ϕ, the steps of deciding the probability that $\mathcal{D}$ satisfies ϕ are given as follows.

1. A deterministic Rabin automaton (DRA) equivalent to ϕ formula is built. The properties definable by DRA are the ω-regular languages [18], therefore for each LTL formula, there is a corresponding DRA.

2. The product of the DRA and $\mathcal{D}$ is then computed, and the result is still a PA, denoted as $\mathcal{D}'$.
3. MECs in $\mathcal{D}'$ which satisfy the Rabin acceptance condition are identified. Any path of $\mathcal{D}'$ reaching these MECs can be proved to satisfy ϕ, therefore states in these MECs are set to be target states.
4. Now the LTL checking problem is reduced to a reachability checking issue. The probability of reaching any state of the targets from the initial state of $\mathcal{D}'$ is calculated, which equals the probability that $\mathcal{D}$ satisfies ϕ.

2.4 PAT Model Checking Framework

The model checkers and related algorithms are implemented in our home-grown model checking framework Process Analysis Toolkit (PAT)[3] [114]. Therefore, in this section we briefly introduce this toolkit.

PAT is a self-contained verification framework, which supports composing, simulating and verifying concurrent systems, real-time systems, and probabilistic systems. Developed mainly in C# language, PAT supports multiple operating systems include Windows, Linux and Mac OS.

In order to handle different kinds of systems, multiple modules are supported in PAT, and some fundamental modules are listed as follows.

- CSP module focuses on concurrent systems. A rich modeling language called CSP# is defined by extending CSP language with shared variables.
- Real-time System (RTS) module supports analysis of real-time systems. In RTS module, a system is modeled using a hierarchical timed process with mutable data. Timed operators such as $deadline$ and $timeout$ are used to capture the dense-time scenarios.
- Web Service module is developed to offer practical solutions to important issues in Web Services paradigm.

[3] `http://www.patroot.com`

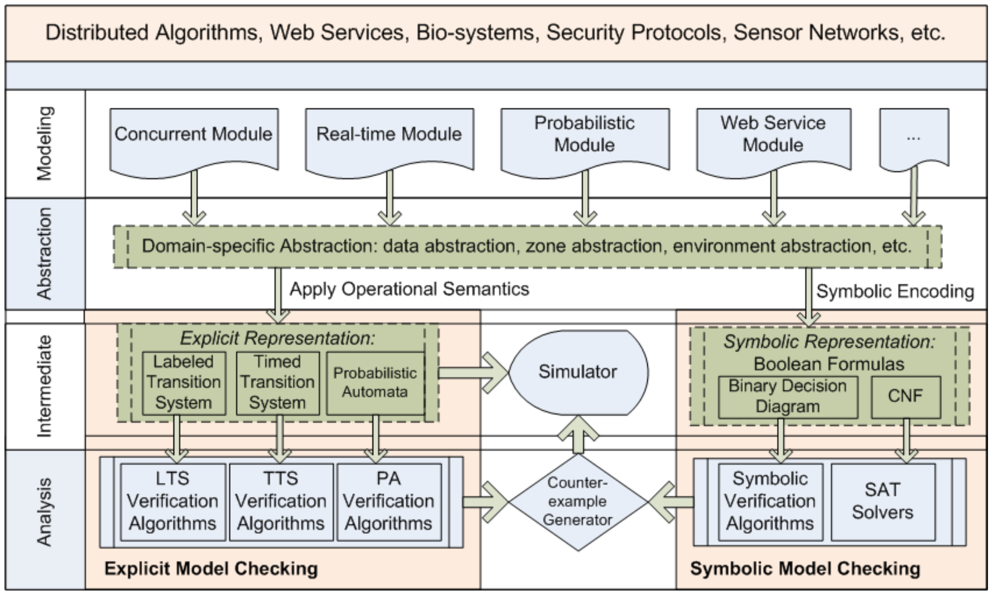

Distributed Algorithms, Web Services, Bio-systems, Security Protocols, Sensor Networks, etc.
Modeling
Concurrent Module
Real-time Module
Probabilistic Module
Web Service Module
...
Abstraction
Domain-specific Abstraction: data abstraction, zone abstraction, environment abstraction, etc.
Apply Operational Semantics
Symbolic Encoding
Intermediate
Explicit Representation:
Labeled Transition System
Timed Transition System
Probabilistic Automata
Simulator
Symbolic Representation: Boolean Formulas
Binary Decision Diagram
CNF
Analysis
LTS Verification Algorithms
TTS Verification Algorithms
PA Verification Algorithms
Counter-example Generator
Symbolic Verification Algorithms
SAT Solvers
Explicit Model Checking
Symbolic Model Checking

Figure 2.6: Architecture of PAT

- NesC module is designed for the verification of sensor networks. The modeling language of this module is NesC [144, 142, 143], which provides fine-grained control over the underlying devices and resources.

Moreover, PAT implements various model checking techniques catering for different properties such as deadlock-freeness, divergence-freeness, reachability, LTL properties with fairness assumptions [115, 114], refinement checking [112, 88, 89] and probabilistic model checking.

For development convenience, PAT has a loosely layered architecture shown in Fig. 2.6. We briefly introduce the functionality of different layers in the following.

- On the top is the modeling layer. Each module has its own specific modeling language to capture the dynamics of the related systems.
- Next, some potential abstraction techniques are used before or with the operational semantics analysis in order to get a semantic model which is subject to efficiently model checking.
- In the intermediate layer, different semantic models are used to represent the original system. For example, Labeled Transition System (LTS) is used to represent the concurrent systems and Timed Transition System (TTS) represents the real-time systems.

- At the bottom layer, suitable algorithms for different semantic models are applied to fulfill the verification.

Because of PAT's architecture, it is a highly extensible and modularized framework for the technical and practical convenience of designing purpose specific model checkers. Pat has the guide for users for customizing the syntax and semantics for their own modeling language, and a corresponding model checker is treated as a new module in PAT. Existing abstraction techniques and verification algorithms in PAT framework can be used in this new module conveniently.

Chapter 3

Model Checking Hierarchical Probabilistic Systems

3.1 Introduction

Designing and verifying probabilistic systems is becoming an increasingly difficult task due to the widespread applications and increasing complexity of such systems. Existing probabilistic model checkers have been designed for hierarchically simple systems. For instance, the popular PRISM checker [80] supports a simple state-based language, based on the Reactive Modules formalism of Alur and Henzinger [11]. The MRMC checker supports a rather simple input language too [72]. The input language of the LiQuor checker [35], named Probmela, is based on an extension of Promela supported by the SPIN model checker. None of the above checkers supports analysis of hierarchical complex probabilistic systems.

In this chapter, we aim to develop a useful tool for verifying hierarchical complex probabilistic systems. First, we propose a language called PCSP# for system modeling. PCSP# is an expressive language, combining Hoare's CSP [69], data structures, and probabilistic choices. It extends previous work on combining CSP with probabilistic choice [94] or on combining CSP with data structures [113]. PCSP# combines low-level programs, e.g., sequence programs defined in a simple imperative language or any C# program, with high-level specifications (with process constructs like parallel, choice, hiding, etc.), as well as probabilistic choices. It supports shared variables as well as abstract events, making it both state-based and event-based. Its underlying semantics is based on Probabilistic Automata (PA).

Second, we propose to verify complex safety properties by showing a refinement relationship (with probability) from a PCSP# model representing a system and a non-probabilistic model representing properties. Note that we assume that the property model is non-probabilistic, i.e., the model is an LTS. We view probability as a necessary devil forced upon us by the unreliability of the system or its environment. In contrast, properties which characterizes correct system behaviors are often irrelevant of the likelihood of some low-level failures. Refinement checking has been traditionally used to verify variants of CSP [104, 105]. It has been proven useful by the success of the FDR checker [105]. Verification of such properties are reduced to the problem of probabilistic model checking against deterministic finite automata, which has been previously solved (see for example [18]). Nonetheless, we present a slightly improved algorithm which is better suited for our setting.

Third, instead of the standard method for model checking SE-LTL formulae, we improve it by safety/co-safety recognition. That is, if an LTL formula or its negation is recognized as a safety property, then the model checking problem is reduced to a refinement checking problem and solved using our refinement checking algorithm. Though the worst-case complexity remains the same, we show that safety/co-safety recognition offers significantly memory/time saving in practice.

Fourth, due to the potential non-determinism in the LTS representing the specification, its normalization may have exponentially more states than the original LTS. As a result, refinement checking may suffers from state space explosion. We show that anti-chain can be used to improve the efficiency of the above-mentioned refinement checking in some particular cases, based on the value iteration method.

Organization The remainder of this chapter is organized as follows. Section 3.2 presents relevant technical definitions. Section 3.3 introduces the syntax and semantics of PCSP#. Section 3.4 presents the verification of PCSP# models, including our trace refinement checking and our approach for verifying SE-LTL formulae with safety recognition. Section 3.5 presents applying anti-chain to probabilistic refinement checking. Section 3.6 evaluates our methods. Section 3.7 surveys the related work. Section 3.8 summarizes the content in this chapter.

3.2 Preliminaries

3.2.1 Normalization of LTS

An LTS is deterministic if and only if given any s and e, there exists only one s' such that $s \xrightarrow{e} s'$. An LTS is non-deterministic if and only if it is not deterministic. A non-deterministic LTS can be translated into a trace-equivalent deterministic LTS by determinization. Furthermore, non-deterministic LTSs containing τ-transitions can be translated into trace-equivalent deterministic LTSs without τ-transitions. The process is known as normalization [104].

Definition 6 (Normalization) *Let $\mathcal{L} = (S, init, Act, T, AP, L)$ be an LTS. The normalized LTS of $\mathcal{L}$ is $nl(\mathcal{L}) = (S', init', Act, T', AP, L')$ where $S' \subseteq 2^S$ is a set of sets of states, $init' = \{s \mid init \rightsquigarrow s\}$ and T' is a transition relation satisfying the following condition: $(N, e, N') \in T'$ if and only if $N' = \{s' \mid \exists\, s : N.\ s \overset{e}{\rightsquigarrow} s'\}$.*

Normalization groups states which can be reached via the same trace. Given two LTSs $\mathcal{L}_0$ and $\mathcal{L}_1$, it is often useful to check whether $traces(\mathcal{L}_0)$ is a subset of $traces(\mathcal{L}_1)$ (or equivalently $\mathcal{L}_0$ trace-refines $\mathcal{L}_1$). There are existing algorithms and tools for trace inclusion check [104]. The idea is to construct the product of $\mathcal{L}_0$ and $nl(\mathcal{L}_1)$ and then search for a state of the form (s, s') such that s enables more visible events than s' does. In the worse case, this algorithm is exponential in the number of states of $\mathcal{L}_1$. It is nonetheless proven to be practical for real-world systems by the success of the FDR checker [105].

3.2.2 Safety/Liveness Recognition in LTL Formulae

SE-LTL formulae can be categorized into either safety or liveness. Informally speaking, safety properties stipulate that "bad things" do not happen during system execution. A finite execution is sufficient evidence to the violation of a safety property. In contrast, liveness properties stipulate that "good things" do happen eventually. A counterexample to a liveness property is an infinite system execution (which forms a loop if the system has finitely many states). In this paper, we adopt the definition of safety and liveness in [6]. For instance, $\Box(a \Rightarrow \Box b)$ and $\Diamond a \Rightarrow \Box b$ are safety properties; $\Box a \Rightarrow \Diamond b$ is a liveness property, whose negation, however, is a safety property. A liveness property whose negation is safety is referred to as co-safety, e.g., $\Diamond a$ is co-safety. We remark that a formula may be neither safety nor liveness, e.g., $\Box\Diamond a \wedge \Box b$. It has been shown in [108] that recognizing whether an LTL formula is safety is PSPACE-complete. A number of methods have been proposed to identify subsets of safety. For instance, *syntactic LTL safety formulae* (which is constituted by $\wedge$, $\vee$, $\Box$, U, X, and propositions or negations of propositions) can be recognized efficiently. A number of methods have been proposed to translate safety LTL to finite state automata [76, 86].

It has been proved in [128] that for every LTL formula ϕ, there exists an equivalent Büchi Automaton. There are many sophisticated algorithms on translating LTL to an equivalent Büchi automaton [52, 109]. In addition, it is possible to tell whether an LTL formula represents safety by examining its equivalent Büchi automaton. For instance, it has been proved in [6] that a (reduced) Büchi automaton specifies a safety property if and only if making all of its states accepting does not change its language. Based on this result, *a Büchi automaton representing a safety property can be viewed as an LTS for simplicity*. The reason is that all of its infinite traces must be accepting and therefore the acceptance condition can be ignored.

3.2.3 Trace Refinement Checking with Anti-Chain

In concurrent systems, given an implementation $\mathcal{L}_1$ and a specification $\mathcal{L}_2$, the standard trace refinement checking is to construct (often on-the-fly) the product $\mathcal{L}_1 \times nl(\mathcal{L}_2)$ and then try to construct a state of the product (s_1, s_2) (where s_1 is a state of $\mathcal{L}_1$ and s_2 is a set of states in $\mathcal{L}_2$) such that s_2 is an empty set. Such a 'co-witness' state is called a TR-witness state. In the worst case, this algorithm has a complexity exponential in the number of states of $\mathcal{L}_2$.

It has been shown that trace refinement checking based on anti-chain offers significantly better performance [136]. Given two LTSs $\mathcal{L}_1$ and $\mathcal{L}_2$, the anti-chain method explores a 'simulation' relation in $\mathcal{L}_1 \times nl(\mathcal{L}_2)$. Given any two states (s_1, s_2) and (s_1', s_2') of $\mathcal{L}_1 \times nl(\mathcal{L}_2)$, let $(s_1', s_2') \leq (s_1, s_2)$ denote $s_1 = s_1'$ and $s_2 \subseteq s_2'$.

Proposition 3.2.1 *If $(s_1', s_2') \leq (s_1, s_2)$ and $(s_1, s_2) \xrightarrow{e} (u, v)$, then there exists (u', v') such that $(s_1', s_2') \xrightarrow{e} (u', v')$ and $u' = u$ and $v \subseteq v'$.* $\Box$

By the above proposition, it can be readily shown that a TR-witness state is reachable from (s_1', s_2') implies that a TR-witness state must be reachable from (s_1, s_2). As a result, if (s_1, s_2) has been explored, we can skip (s_1', s_2').

Algorithm 1 Trace Refinement Checking Algorithm with Anti-chain

```
let $working$ be a stack containing a pair $(init_1, \{s \mid init_2 \rightsquigarrow s\})$;
let $antichain := \varnothing$;
while $working \neq \varnothing$ do
    pop $(impl, spec)$ from $working$;
    $antichain := antichain \Cup (impl, spec)$;
    for all $(impl, e, impl') \in T_1$ do
        if $e = \tau$ then
            $spec' := spec$;
        else
            $spec' := \{s' \mid \exists\, s \in spec.\ s \overset{e}{\rightsquigarrow} s'\}$;
        end if
        if $spec' = \varnothing$ then return false;
        end if
        if $(impl', spec') \Subset antichain$ is not true then
            push $(impl', spec')$ into $working$;
        end if
    end for
end while
return true;
```

Formally, an anti-chain is a set A of sets such that $x \nsubseteq y$ and $y \nsubseteq x$ for all $x \in A$ and $y \in A$, i.e., any pair of sets in A are incomparable. An anti-chain supports two operations. One is to check whether it contains a subset of a given set. let x be the given set, we denote $x \Subset A$ if and only if there exists $y \in A$ such that $y \subseteq x$. The other is to add a given set x in A. $A \Cup x$ is defined as $\{y \mid y \in A \wedge x \nsubseteq y\} \cup \{x\}$, i.e., $A \Cup x$ contains x and all sets in A which is not a superset of x. Obviously, an empty set is an anti-chain by definition.

Algorithm 1 shows the anti-chain based algorithm. In an abuse of notation, we write $(s, X) \Subset A$ to denote that the set $(\{s\} \cup X) \Subset A$; and $A \Cup (s, X)$ to denote $A \Cup (\{s\} \cup X)$. The algorithm works as follows. After initialization, the algorithm pops one state $(impl, spec)$ from $working$ and adds it to the set $antichain$, and then generates all successors of the state and adds them to $working$ unless $(impl', spec') \Subset antichain$ is true, till the stack $working$ is empty or a TR-witness state is found. We remark that $antichain$ keeps to be an anti-chain during this algorithm, because line 5 and line 14 guarantee there are no subsets or supersets of the new added state in the updated $antichain$. Soundness of the algorithm can be referred to in [4] [136].

3.3 Hierarchical Modeling

In this section, we present PCSP#, which is designed for modeling and verifying probabilistic systems. We remark that the LiQuor checker, which is based on Probmela, makes a step towards an expressive useful modeling language. Nonetheless, Probmela is not capable of modeling fully hierarchical systems.

3.3.1 Language Syntax

PCSP# extends the CSP# language [113] with probabilistic choices. CSP# integrates low-level programs with high-level compositional specification. It is capable of modeling systems with not only complicated data structures (which are manipulated by the low-level programs) but also hierarchical systems with complex control flows (which are specified by the high-level specification). Compared with PCSP [94], PCSP# supports explicit complex data structures/operations.

A PCSP# model is a 3-tuple $(Var, init, P)$ where Var is a set of global variables (with bounded domains) and channels; $init$ is the initial values of Var; P is a process. A variable can be either of simple types like boolean, integer, arrays of integers or any user-defined data type (which could be defined in an external imperative languages such as C# and Java). The process P is an extension of Hoare's classic CSP. Part of its syntax is defined as follows.

$$\begin{array}{lll}
P ::= & Stop \mid Skip & \text{– primitives} \\
\mid & e \rightarrow P & \text{– event prefixing} \\
\mid & a\{program\} \rightarrow P & \text{– data operation prefix} \\
\mid & P \Box Q \mid P \sqcap Q \mid \textbf{if } b \textbf{ then } P \textbf{ else } Q & \text{– choices} \\
\mid & case\{b_0 : P_0;\ b_1 : P_1;\ \cdots;\ b_k : P_k\} & \text{– multiple conditional choices} \\
\mid & P;\ Q & \text{– sequence} \\
\mid & P \parallel Q \mid P ||| Q & \text{– concurrency} \\
\mid & P \setminus X & \text{– hiding} \\
\mid & Q & \text{– process referencing} \\
\mid & \textbf{pcase } \{pr_0 : P_0;\ pr_1 : P_1;\ \cdots;\ pr_k : P_k\} & \text{– probabilistic multi-choices}
\end{array}$$

where P, P_i and Q range over processes, e is a simple event, a is the name of a sequential program; b is a Boolean expression, pr_i is a positive integer to express the probability weight. Process $Stop$ does nothing. Process $Skip$ terminates. Process $e \rightarrow P$ engages in event e first and then behaves as P. Combined with parallel composition, event e may serve as a multi-party synchronization barrier. Process $a\{program\} \rightarrow P$ generates an event a, executes a sequential program $program$ at the same time, and then behaves as P. External C# data operations can be invoked in $program$.

A variety of choices are supported, e.g., $P \Box Q$ for external choice; $P \sqcap Q$ for internal non-determinism and **if** b **then** P **else** Q for conditional branching. Multiple conditional choices is denoted as $case\{b_0 : P_0;\ b_1 : P_1;\ \cdots;\ b_k : P_k\}$ where b_i is boolean variable. At each state, there must be one and only one b_i is true, and P_i is chosen afterwards. Process $P;\ Q$ behaves as P until P terminates and then behaves as Q. Parallel composition of two processes is written as $P \parallel Q$, where P and Q may communicate via multi-party event synchronization. If P and Q only communicate through channels or variables, then it is written as $P|||Q$. Process $P \setminus X$ hides occurrence of any event in X. Recursion is supported through process referencing. Lastly, probabilistic choice is written in the form of **pcase** $\{pr_0 : P_0;\ pr_1 : P_1;\ \cdots;\ pr_k : P_k\}$. Intuitively, it means that with $\frac{pr_i}{pr_0+pr_1+\cdots+pr_k}$ probability, the system behaves as P_i. Note pr_i can be a constant, an integer variable, or even a function whose return value is integer, which makes the modeling of probabilistic choices flexible. The probability of each transition can be decided at run time.

Pacemaker A pacemaker is an electronic implanted device which functions to regulate the heart beat by electrically stimulating the heart to contract and thus to pump blood throughout the body. Common pacemakers are designed to correct bradycardia, i.e., slow heart beats. A pacemaker mainly performs two functions, i.e., sensing and pacing. Sensing is to monitor the heart's natural electrical activity, helping the pacemaker to gather information on the heart beats and react accordingly. Pacing is when a pacemaker sends electrical stimuli, i.e., tiny electrical signals, to heart through a pacing lead, which starts a heart beat. A pacemaker can operate in many different modes, according to the implanted patient's heart problem. The following is a high-level abstraction of the simplest mode of pacemaker, i.e., the AAT mode.

```
var $count$ = 0;
$AAT$     = $(Heart \parallel Pacing) \setminus \{missingPulseA, missingPulseV\}$
$Heart$   = pcase {
            $[pA] : missingPulseA \rightarrow pulseV \rightarrow Heart$
            $[pV] : pulseA \rightarrow missingPulseV \rightarrow Heart$
            $[1 - pA - pV] : pulseA \rightarrow pulseV \rightarrow Heart$
            };
$Pacing$  = $pulseA \rightarrow Pacing \Box pulseV \rightarrow Pacing$
            $\Box missingPulseA \rightarrow add\{count++\} \rightarrow$ pcase {
            $[99.54] : pulseB\{count--\} \rightarrow Pacing$
            $[0.46] : Pacing$
            }
            $\Box missingPulseV \rightarrow add\{count++\} \rightarrow$ pcase {
            $[99.54] : pulseW\{count--\} \rightarrow Pacing$
            $[0.46] : Pacing$
            };
```

Variable $count$ is an integer (with a default bound) which records the number of skipped pulses. A (mode of the) pacemaker is typically modeled in the following form: $Heart \parallel Pacing$ where $Heart$ models normal or abnormal heart condition; $Pacing$ models how the pacemaker functions. In this particular mode, process $Heart$ generates two events $pulseA$ (i.e., atrium does a pulse) and $pulseV$ (i.e., ventricle does a pulse), periodically for a normal heart or with one of them missing once a while for an abnormal heart. In the latter case, event $missingPulseA$ or $missingPulseV$ is generated. Constant pA is the (patient-dependent) probability of $pulseA$ missing; pV is the probability of $pulseV$ missing. Process $Pacing$ synchronizes with process $Heart$. If event $missingPulseA$ (denoting the missing of event $pulseA$) is monitored, variable $count$ is incremented by one. Notice that the event add is associated with the simple program of updating $count$. In general, it can be associated with any state update function. It is in this way that state update is introduced in an event-based language. Ideally, the pacemaker helps the heart to beat by generating event $pulseB$. Once $pulseB$ is generated, $count$ is decremented by one. Similarly, it generates $pulseW$ when $pulseV$ is missing. Note that it has been reported that pacemaker may malfunction for certain rate (exactly 0.46%) [92]. This is reflected in the model again using **pcase**. If a $pulseB$ or $pulseW$ is skipped, $count$ is not decremented.

At the top level, the pacemaker system is a choice of different modes. Each mode is often a parallel composition of multiple components. Each component may have internally hierarchies due to complicated sensing and pacing behaviors. We skip the details (refer to [20]) and remark that our modeling language is more suitable for such systems than those supported by existing probabilistic model checkers. □

3.3.2 Operational Semantics

The underlying semantics of PCSP# is PA, which is expressive enough to capture systems with probabilistic choices as well as nondeterminism and concurrency. A concrete system configuration in PCSP# model is a pair (V, P) where V is a variable valuation and $P \in \mathcal{P}$ is a process. For simplicity, an empty valuation is written as $\emptyset$. A transition of the system is written in the form $(V, P) \xrightarrow{x} (V', P')$ such that $x \in Act_\tau$. The operational semantics is defined by associating a set of firing rules with every process construct. A special event $\checkmark \in Act$ indicates the termination of a PCSP process. Further, we assume a function $upd(V, prog)$ which, given a sequential program $prog$ and V, returns the modified valuation function V' according to the semantics of the program. Given a model P, $\alpha(P)$ is used to denote all observable actions contained in P; $En(V, P)$ represents the enabled actions in state (V, P). The firing rules associated with probability are presented as follows.

$$\frac{}{(V, Skip) \xrightarrow{\checkmark} (V, Stop)} \quad [\ sk\ ]$$

- Rule sk indicates $Skip$ can execute the termination action and become $Stop$. Variables are unchanged in this step.

$$\frac{}{(V, e\{prog\} \rightarrow P) \xrightarrow{e} (upd(V, prog), P)} \quad [\ as\ ]$$

- Rule as indicates the event prefixing process can engage the event, and update the value of variables according to the associated program. Afterwards, the process behaves as P.

$$\frac{(V, P) \xrightarrow{x} (V', P')}{(V, P \Box Q) \xrightarrow{x} (V', P')} \quad [\ ext1\ ]$$

$$\frac{(V, Q) \xrightarrow{x} (V', Q')}{(V, P \Box Q) \xrightarrow{x} (V', Q')} \quad [\ ext2\ ]$$

- Rules $ext1$ and $ext2$ capture the behavior of external choice. Both branches are possible as long as they are executable.

$$\frac{}{(V, P \sqcap Q) \xrightarrow{\tau} (V, P)} \quad [\ int1\ ]$$

$$\frac{}{(V, P \sqcap Q) \xrightarrow{\tau} (V, Q)} \quad [\ int2\ ]$$

- Rules $int1$ and $int2$ capture the behavior of internal choice. An extra invisible action τ is used to choose the following behavior of the process. Variables keep unchanged in this step.

$$\frac{V \models b}{(V, \textbf{if}\ (b)\ \{P\}\ \textbf{else}\ \{Q\}) \xrightarrow{\tau} (V, P)} \quad [\ if1\ ]$$

$$\frac{V \not\models b}{(V, \textbf{if}\ (b)\ \{P\}\ \textbf{else}\ \{Q\}) \xrightarrow{\tau} (V, Q)} \quad [\ if2\ ]$$

- Rules $if1$ and $if2$ capture the behavior of conditional choice. According to whether b is true or not, P and Q are chosen to take over the control of the process.

$$\frac{P \xrightarrow{\checkmark} P'}{(V, P;\ Q) \xrightarrow{\tau} (V, Q)} \quad [\ se1\ ]$$

$$\frac{(V, P) \xrightarrow{e} (V', P'), \checkmark \notin En(V, P)}{(V, P;\ Q) \xrightarrow{\tau} (V', P';\ Q)} \quad [\ se2\ ]$$

- The above two rules describe the behaviors of sequential processes. $se1$ indicates if termination event happens in P, then Q can take over control of the process; $se2$ indicates if ✓ is not activated currently, the whole process still behaves as P.

$$\frac{(V, P) \xrightarrow{e} (V', P'), e \notin \alpha(Q)}{(V, P \parallel Q) \xrightarrow{e} (V', P' \parallel Q)} \quad [\ pl1\ ]$$

$$\frac{(V,Q) \xrightarrow{e} (V',Q'), e \notin \alpha(P)}{(V,P \parallel Q) \xrightarrow{e} (V',P \parallel Q')} \quad [\ pl2\ ]$$

$$\frac{P \xrightarrow{e} P', Q \xrightarrow{e} Q', x \in (\alpha(Q) \cap \alpha(P))}{(V,P \parallel Q) \xrightarrow{e} (V,P' \parallel Q')} \quad [\ pl3\ ]$$

- Rules $pl1$, $pl2$ and $pl3$ describe the behaviors of parallel processes. If an action is not the common action in both P and Q, it can be executed with variable updates, and the process without this event keeps unchanged. Otherwise, both processes will execute this action simultaneously, and no variable can be updated in this scenario.

$$\frac{(V,P) \xrightarrow{e} (V',P'), e \neq \checkmark}{(V,P|||Q) \xrightarrow{e} (V',P'|||Q)} \quad [\ inl1\ ]$$

$$\frac{(V,Q) \xrightarrow{e} (V',Q'), e \neq \checkmark}{(V,P|||Q) \xrightarrow{e} (V',P|||Q')} \quad [\ inl2\ ]$$

$$\frac{P \xrightarrow{\checkmark} P', Q \xrightarrow{\checkmark} Q'}{(V,P|||Q) \xrightarrow{\checkmark} (V,P'|||Q')} \quad [\ inl3\ ]$$

- Rules $inl1$, $inl2$ and $inl3$ describe the behaviors of interleaving processes. Processes can execute their actions without affecting others, as long as the action is not ✓. All ✓ should happen simultaneously in all interleaving processes, denoted in $inl3$.

$$\frac{(V,Q) \xrightarrow{x} (V',Q'), P \widehat{=} Q}{(V,P) \xrightarrow{x} (V',Q')} \quad [\ def\ ]$$

- Rule def indicates that if P is a reference of Q, then they have the same behavior.

$$\frac{(V,P) \xrightarrow{a} (V',P')}{(V,P) \xrightarrow{a} \mu \text{ such that } \mu(V',P') = 1} \quad [\ pb1\ ]$$

$$\frac{}{\begin{array}{l}(V, \textbf{pcase}\ \{pr_0 : P_0;\ \cdots;\ pr_k : P_k\}) \xrightarrow{\tau} \mu \\ \text{where } \mu((V, P_i)) = \frac{pr_i}{pr_0 + \cdots + pr_k} \text{ for all } i \in [0, k]\end{array}} \quad [\ pb2\ ]$$

- Rules $pb1$ and $pb2$ define the semantics of $pcase$. $pb2$ states that if **pcase** is activated, then it transmits to a distribution μ via action τ. The probability of reaching the successive states follows the probability weight. Note the valuation of variables keeps unchanged. $pb1$ indicates if no $pcase$ is activated in P, the distribution from (V, P) is always a trivial distribution.

3.4 Probabilistic Refinement Checking

Refinement checking has been traditionally used to verify CSP [69]. Different from temporal-logic based model checking, refinement checking works by taking a model (often in the same language) as a property. The property is verified by showing a refinement relationship from the system model to the property model. There are different refinement relationships designed for proving different properties. In the following, we focus on trace refinement and remark that our approach can be extended to stable failures refinement or failures/divergence refinement. For instance, one way of verifying the pacemaker is to check whether the pacemaker model (present in Example 3.3.1) trace-refines the following model (without variables) which models a 'fine' heart.

$$\begin{array}{l} OKHrt = pulseA \rightarrow pulseV \rightarrow OKHrt \Box pulseA \rightarrow pulseW \rightarrow OKHrt \Box \\ \qquad\qquad pulseB \rightarrow pulseV \rightarrow OKHrt \Box pulseB \rightarrow pulseW \rightarrow OKHrt \end{array}$$

In theory, it is possible to encode the property model as temporal logic formulae (as temporal logic is typically more expressive than LTS) and then apply temporal-logic based model checking to verify the property. It is, however, impractical. For instance, LTL model checking is exponential in the size of the formulae and therefore it cannot handle formulae which encode non-trivial property model. In short, refinement checking allows users to verify a different class of properties from temporal logic formulae.

3.4.1 Refinement Checking PCSP#

Because of probabilistic choices, refinement checking in our setting is not simply to verify whether traces of a PCSP# model is subset of those of another. Instead, it is 'how likely' the system behaves as specified by the property model (in the presence of unreliability of system components). Because we assume the property model is non-probabilistic, the problem is thus to calculate the probability of a PA (i.e., the semantics of PCSP# model) trace-refines an LTS (i.e., the semantics of a non-probabilistic PCSP# model).

Definition 7 (Refinement Probability) *Let $\mathcal{M}$ be a PA and $\mathcal{L}$ be an LTS. The maximum probability of $\mathcal{M}$ trace-refines $\mathcal{L}$ is defined by $\mathcal{P}^{max}(\mathcal{M} \sqsupseteq \mathcal{L}) = \mathcal{P}^{max}_{\mathcal{M}}(traces(\mathcal{L}))$. The minimum is defined by $\mathcal{P}^{min}(\mathcal{M} \sqsupseteq \mathcal{L}) = \mathcal{P}^{min}_{\mathcal{M}}(traces(\mathcal{L}))$.* □

Intuitively, the probability of $\mathcal{M}$ refines $\mathcal{L}$ is the sum of the probability of $\mathcal{M}$ exhibiting every trace of $\mathcal{L}$. The probability may vary due to different scheduling. One way of calculating the maximum/minimum probability [18] is to (1) build a deterministic LTS $\mathcal{L}^{-1}$ which complements $\mathcal{L}$ (such that $traces(\mathcal{L}^{-1}) = \Sigma^* \setminus traces(\mathcal{L})$); (2) compute the product of $\mathcal{M}$ and $\mathcal{L}^{-1}$; 3) calculate the maximum/minimum probability of paths of the product.

In the following, we present a slightly improved algorithm which avoids the construction of $\mathcal{L}^{-1}$. Note that for a complicated language like PCSP#, computing $\mathcal{L}^{-1}$ is highly non-trivial. The algorithm is inspired by the refinement checking algorithm in FDR. Firstly, we normalize $\mathcal{L}$ using the standard powerset construction. Next, we compute the synchronous product of $\mathcal{M}$ and $nl(\mathcal{L})$, written as $\mathcal{M} \times nl(\mathcal{L})$. It can be shown that the product is still a PA.

Definition 8 (Product PA) *Let $\mathcal{M} = (S_{\mathcal{M}}, init_{\mathcal{M}}, Act, Pr_{\mathcal{M}}, AP_{\mathcal{M}}, L_{\mathcal{M}})$ be a PA, and $\mathcal{L} = (S_{\mathcal{L}}, init_{\mathcal{L}}, Act, T_{\mathcal{L}}, AP_{\mathcal{L}}, L_{\mathcal{L}})$ be a deterministic LTS without τ-transitions. The product is the PA $\mathcal{M} \times \mathcal{L} = (S_{\mathcal{M}} \times S_{\mathcal{L}}, (init_{\mathcal{M}}, init_{\mathcal{L}}), Act, Pr, AP_{\mathcal{M}} \cup AP_{\mathcal{L}}, L)$ such that Pr is the least transition relation which satisfies the following conditions.*

- *If $s_m \xrightarrow{\tau} \mu$ in $\mathcal{M}$, then $(s_m, s_l) \xrightarrow{\tau} \mu'$ in $\mathcal{M} \times \mathcal{L}$ for all $s_l \in S_{\mathcal{L}}$ such that $\mu'((s_m', s_l)) = \mu(s_m')$ for all $s_m' \in S_{\mathcal{M}}$.*
- *If $s_m \xrightarrow{e} \mu$ in $\mathcal{M}$ and $s_l \xrightarrow{e} s_l'$ in $\mathcal{L}$, then $(s_m, s_l) \xrightarrow{e} \mu'$ in $\mathcal{M} \times \mathcal{L}$ such that $\mu'((s_m', s_l')) = \mu(s_m')$ for all $s_m' \in S_{\mathcal{M}}$.*

L is a new labeling function satisfying $L(s, s') = L_{\mathcal{M}}(s) \cup L_{\mathcal{L}}(s')$, $s \in S_{\mathcal{M}}$ and $s' \in S_{\mathcal{L}}$.

In the product, there are two kinds of transitions, i.e., τ-transitions from $\mathcal{M}$ with the same probability or transitions labeled with a visible event with probability 1. Note that τ-transitions are not synchronized, whereas visible events must be jointly performed by $\mathcal{M}$ and $\mathcal{L}$. Let $G \subseteq S_{\mathcal{M}} \times S_{\mathcal{L}}$ be the least set of states satisfying the following condition: for every pair $(s, s') \in G$, $s' = \emptyset$. Intuitively, $(s, s') \in G$ if and only if a trace of $\mathcal{M}$ leading to s is not possible in $\mathcal{L}$. The following theorem states our main result on refinement checking.

Theorem 3.4.1 *Let $\mathcal{M}$ be a PA; $\mathcal{L}$ be an LTS; $\mathcal{D} = \mathcal{M} \times nl(\mathcal{L})$. $\mathcal{P}^{max}(\mathcal{M} \sqsupseteq \mathcal{L}) = 1 - \mathcal{P}^{min}_{\mathcal{D}}(G)$ and $\mathcal{P}^{min}(\mathcal{M} \sqsupseteq \mathcal{L}) = 1 - \mathcal{P}^{max}_{\mathcal{D}}(G)$.*

Proof Let δ be any scheduler for $\mathcal{M}$. Note that δ can be extended to be a scheduler for $\mathcal{D}$ straightforwardly. For simplicity, we use δ to denote both of them. Let $X \subseteq paths(\mathcal{M})$. The following shows that the equivalence holds with any scheduler.

$$\begin{array}{ll}
\mathcal{P}^{\delta}_{M}(\{\pi \in paths(\mathcal{M}) \mid trace(\pi) \in traces(\mathcal{L})\}) & \\
\equiv 1 - \mathcal{P}^{\delta}_{M}(\{\pi \in paths(\mathcal{M}) \mid trace(\pi) \notin traces(\mathcal{L})\}) & \text{– by def.} \\
\equiv 1 - \mathcal{P}^{\delta}_{\mathcal{D}}(\{\pi \in paths(\mathcal{D}) \mid trace(\pi) \notin traces(\mathcal{L})\}) & \text{– (1)} \\
\equiv 1 - \mathcal{P}^{\delta}_{\mathcal{D}}(G) & \text{– (2)}
\end{array}$$

(1) is true because for every path of $\mathcal{M}$, there is a path of $\mathcal{D}$ with the same probability (as $\mathcal{L}$ is non-probabilistic) and the same trace; and vice versa. (2) is true because by [104], a path of $\mathcal{D}$ such that $trace(\pi) \notin traces(\mathcal{L})$ if and only if it visits some state in G. It can be shown then $\mathcal{P}^{max}(\mathcal{M} \sqsupseteq \mathcal{L})$, which is $\mathcal{P}^{max}_{M}(\{\pi \in paths(\mathcal{M}) \mid trace(\pi) \in traces(\mathcal{L})\})$, is $1 - \mathcal{P}^{min}_{\mathcal{D}}(G)$ and $\mathcal{P}^{min}(\mathcal{M} \sqsupseteq \mathcal{L})$ is $1 - \mathcal{P}^{max}_{\mathcal{D}}(G)$. □

Intuitively, the theorem holds because, with any scheduler, the probability of $\mathcal{M}$ *not* refining $\mathcal{L}$ is exactly the probability of reaching G in $\mathcal{M} \times nl(\mathcal{L})$. As a result, refinement checking is reduced to reachability probability in $\mathcal{D}$. There are known approaches to compute $\mathcal{P}^{max}_{\mathcal{M}}(G)$ and $\mathcal{P}^{min}_{\mathcal{M}}(G)$, e.g., using an iterative approximation method or by solving linear programs [18].

3.4.2 SE-LTL Probabilistic Model Checking as Refinement Checking

Another way of specifying properties is through temporal logic. In this section, we examine the problem of model checking PCSP# models against SE-LTL formulae. SE-LTL is an effective property language for PCSP# as it can be constituted by state propositions as well as events. In the pacemaker example, an SE-LTL formula could be stated as follows: $(\square count \leq 10) \wedge \square(missingPulseA \Rightarrow X\ pulseB)$ which states $count$ must be always less than 10 and event $missingPulseA$ must lead to an occurrence of event $pulseB$ next. Given a PA $\mathcal{M}$ and an SE-LTL formula ϕ, let $\mathcal{P}^{max}_{\mathcal{M}}(\phi)$ (and $\mathcal{P}^{min}_{\mathcal{M}}(\phi)$) denote the maximum (and minimum) probability of $\mathcal{M}$ satisfying ϕ.

The standard LTL probabilistic model checking method is the automata-theoretic approach [18], which is computationally expensive due to multiple reasons. Firstly, the construction of the deterministic Rabin automaton is expensive. Given a Büchi automaton $\mathcal{B}$, its equivalent deterministic Rabin automaton, in the worse case, is of size $2^{O(n \log n)}$ where n is the size of $\mathcal{B}$. Secondly, identifying the end components is expensive. The worse case complexity is bounded by $|S| \times (|S| + |T|)$ where $|S|$ is the number system states and $|T|$ is the number of the system transitions. In this section, we show that by recognizing safety properties, we can improve probabilistic model checking of certain class of SE-LTL formulae by avoiding constructing the Rabin automaton or computing the end components.

Given a formula ϕ, we check whether ϕ is a safety property using the following approach. Firstly, we check whether it is a *syntactic LTL safety formula* [108]. If it is not, we generate an equivalent Büchi automaton using an existing approach [52], and then check whether all states of the Büchi automaton are accepting. If positive, by the result proved in [6], ϕ is a safety property. If we cannot conclude that ϕ is safety, we assume that it is not. This is a sound but not complete method for recognizing safety. In practice, we found that it is effective in recognizing most of the commonly used safety properties, including for example $\square(a \Rightarrow \square b)$ and $\Diamond a \Rightarrow \square b$.

Next, we adopt the workflow shown in Fig. 3.1 to improve probabilistic model checking. Let ϕ be an SE-LTL formula and $\mathcal{B}$ be the equivalent Büchi automaton. If ϕ is a safety property, then $\mathcal{B}$ can be simply treated as an LTS, as discussed in Section 3.2. The problem of model checking a system model $\mathcal{M}$ against ϕ is thus reduced to calculate the probability of $\mathcal{M}$ refines the LTS $\mathcal{B}$. If ϕ cannot be determined as a safety property, then we check whether ϕ is a co-safety property. A Büchi automaton $\mathcal{B}'$, equivalent to $\neg\,\phi$, is generated. If $\mathcal{B}'$ is a safety property, the problem of model checking ϕ is thus reduced to calculate the

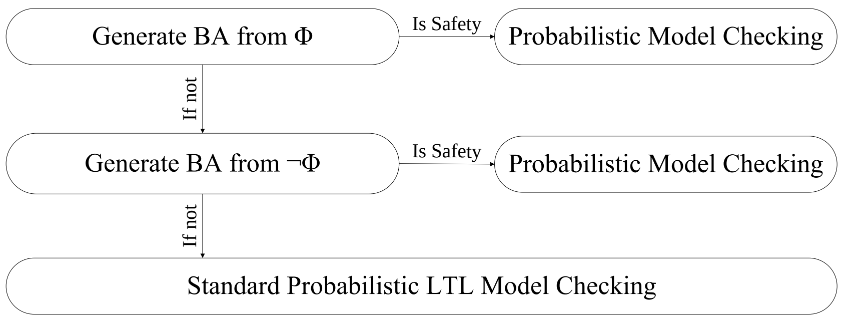

Generate BA from Φ
Is Safety
Probabilistic Model Checking
If not
Generate BA from ¬Φ
Is Safety
Probabilistic Model Checking
If not
Standard Probabilistic LTL Model Checking

Figure 3.1: Workflow

probability of $\mathcal{M}$ refines the LTS $\mathcal{B}'$.

Theorem 3.4.2 *Let $\mathcal{M}$ be an PA; ϕ be an SE-LTL formula; $\mathcal{B}$ be the Büchi automaton equivalent to ϕ. Let $\mathcal{B}^{-1}$ be the Büchi automaton equivalent to $\neg\phi$. If ϕ is safety, then $\mathcal{P}^{max}_{\mathcal{M}}(\phi) = \mathcal{P}^{max}(\mathcal{M} \sqsupseteq \mathcal{B})$ and $\mathcal{P}^{min}_{\mathcal{M}}(\phi) = \mathcal{P}^{min}(\mathcal{M} \sqsupseteq \mathcal{B})$; If ϕ is co-safety, then $\mathcal{P}^{max}_{\mathcal{M}}(\phi) = 1 - \mathcal{P}^{min}(\mathcal{M} \sqsupseteq \mathcal{B}^{-1})$ and $\mathcal{P}^{min}_{\mathcal{M}}(\phi) = 1 - \mathcal{P}^{max}(\mathcal{M} \sqsupseteq \mathcal{B}^{-1})$.* □

The proof of the theorem is sketched as follows. If ϕ is a safety property, any trace of $\mathcal{B}$ is accepting. It can be shown that any trace of $\mathcal{M}$ which is not a trace of $\mathcal{B}$ is a counterexample to ϕ. Therefore, the probability of $\mathcal{M}$ exhibiting a trace of $\mathcal{B}$ (i.e., the probability of $\mathcal{M}$ trace-refines $\mathcal{B}$) is the probability of $\mathcal{M}$ satisfying ϕ. Next, the theorem states that the probability of $\mathcal{M}$ not-refining $\mathcal{B}$ is the probability of $\mathcal{M}$ executing a finite prefix of any traces which is not possible for $\mathcal{B}$. Similarly, we can prove the result for co-safety properties.

By the theorem, probabilistic model checking of safety LTL formula or co-safety LTL formula is reduced to probabilistic refinement checking, which is considerably more efficient as we avoid constructing the deterministic Rabin automaton or identifying end components. This is confirmed by the experiments conducted in Section 3.6.

3.5 Probabilistic Refinement Checking with Anti-Chain

In this section, we show that anti-chain can be used to improve our probabilistic refinement checking, i.e., the implementation is given as a PA and the specification is given as an LTS. We first introduce a lemma in the following.

Lemma 1 *Let $\mathcal{D} = (S_d, init_d, Act_d, Pr_d, AP_d, L_d)$ be a PA; $\mathcal{L} = (S_l, init_l, Act_l, T, AP_l, L_l)$ be an LTS. Let $\mathcal{P}$ be $\mathcal{D} \times nl(\mathcal{L})$. Let G be the set of TR-witness states of $\mathcal{P}$. For all state (u_1, v_1) and (u_2, v_2) of $\mathcal{P}$ s.t. $(u_2, v_2) \leq (u_1, v_1)$, $Pr_{max}(\mathcal{P}, (u_1, v_1), G) \geq Pr_{max}(\mathcal{P}, (u_2, v_2), G)$ and $Pr_{min}(\mathcal{P}, (u_1, v_1), G) \geq Pr_{min}(\mathcal{P}, (u_2, v_2), G)$.*

Proof The above can be proved with an induction. The base case is that (u_2, v_2) is in G. By definition, (u_1, v_1) must be in G and therefore the lemma holds. Next, we show the induction step. Assume that (u_2', v_2') satisfies the lemma above. For every distribution μ_2 from (u_2, v_2), by Definition 8, there must exist a distribution μ_1 from (u_1, v_1) and for every state (u_2', v_2'), there exists (u_1', v_1') such that $\mu_2((u_2', v_2')) = \mu_1((u_1', v_1'))$ and $(u_2', v_2') \leq (u_1', v_1')$. By induction hypothesis, we have $Pr_{max}(\mathcal{P}, (u_1', v_1'), G) \geq Pr_{max}(\mathcal{P}, (u_2', v_2'), G)$ and $Pr_{min}(\mathcal{P}, (u_1', v_1'), G) \geq Pr_{min}(\mathcal{P}, (u_2', v_2'), G)$. Thus we have $Pr_{max}(\mathcal{P}, (u_1, v_1), G) \geq Pr_{max}(\mathcal{P}, (u_2, v_2), G)$ and $Pr_{min}(\mathcal{P}, (u_1, v_1), G) \geq Pr_{min}(\mathcal{P}, (u_2, v_2), G)$. Therefore, we conclude that the lemma holds. □

Compared to probabilistic reachability calculation for a general PA, the above lemma gives us additional information, which can be potentially useful in speeding up the calculation. In the following, we discuss how we can make use of the information so as to improve the probabilistic refinement checking using the iterative calculation method.

The first step is building the product PA meanwhile finding the target states, which is shown in Algorithm 2. The implementation and specification are defined in Definition 8. Different from the non-probabilistic cases, *the state space cannot be reduced in the probabilistic models*; instead, we define a function sub of the product state s satisfying $s.sub = \{t \mid t \in S \land s \leq t\}$, where S is the state space of the product PA. Then the refinement checking is reduced to probabilistic reachability of a set of target states, denoted by $Target$. During the iterative calculation, whenever the probability of state s is updated, e.g., to p, according to lemma 1, **all states in $s.sub$ whose probability is less than p could be set to p directly.** This could speed up each iteration and potentially improve probabilistic refinement checking.

3.6 Evaluations

Our methods have been implemented in PAT. We extend PAT with a module to support PCSP#, integrating the existing CSP# language with probabilistic choices. Furthermore,

Algorithm 2 Building PA in Probabilistic Refinement Checking with Anti-chain

```
1: let $working$ be a stack containing a pair $(init_d, \{s \mid init_l \rightsquigarrow s\})$;
2: let $visited := \{(init_d, \{s \mid init_l \rightsquigarrow s\})\}$; let $Target = \varnothing$; ;
3: while $working \neq \varnothing$ do
4:     pop $(impl, spec)$ from $working$;
5:     for all $(impl, e, \mu) \in Pr_d$ do
6:         if $e = \tau$ then
7:             $spec' := spec$;
8:         else
9:             $spec' := \{s' \mid \exists\, s \in spec.\ s \overset{e}{\rightsquigarrow} s'\}$;
10:        end if
11:        for all $impl' \in S_d$ do
12:            if $\mu(impl') > 0 \wedge (impl', spec') \notin visited$ then
13:                push $(impl', spec')$ into $working$;
14:                $visited := visited \cup (impl', spec')$;
15:                if $spec' = \varnothing$ then
16:                    $Target := Target \cup (impl', spec')$;
17:                end if
18:                for all $(impl', spec'') \in visited$ do
19:                    if $(impl', spec'') \leq (impl', spec')$ then
20:                        $(impl', spec'').sub.Add(impl', spec')$;
21:                    else if $(impl', spec') \leq (impl', spec'')$ then
22:                        $(impl', spec').sub.Add(impl', spec'')$;
23:                    end if
24:                end for
25:            end if
26:        end for
27:    end for
28: end while
29: return true;
```

we extend the library of model checking algorithms in PAT with probabilistic refinement checking and probabilistic SE-LTL model checking with safety recognition. Moreover, anti-chain based approach is used to speed up the refinement checking.

We evaluate our implementation using benchmark systems. Three sets of experiments are conducted, focusing on 1) the efficiency of probabilistic refinement checking in PCSP#; 2) the improvement of refinement checking using safety recognition; 3) the improvement of refinement checking using anti-chain. We use the value iteration method in calculating the probability and set termination threshold as relative difference 1.0E-6 (same as state-of-the-art probabilistic model checking PRISM [80]). The testbed for all experiments is a PC with

Intel Core 2 Quad 9550 CPU at 2.83GHz and 2GB RAM.

In the first two sets of experiments, we compare our results with PRISM version 4.0.1. In order to perform a fair comparison, we use existing PRISM models; re-model them using PCSP# language and re-verify them using PAT. It should be noticed that our language is capable of specifying hierarchical systems which are beyond PRISM. Working with existing PRISM models, which are not hierarchical, is not justified to show our advantage. Nonetheless, we show that even for those systems, PCSP# offers an intuitive and compact representation and PAT offers comparable performance. The following models are adopted for comparison.

- Model ME describes a probabilistic solution to N-process mutual exclusion problem, which is based on [100].
- Model RC is a shared coin protocol of the randomized consensus algorithm, which is based on [13]. Note that N is the number of coins and K is a parameter used to generate suitable probability.
- Model DP is the probabilistic N-dining philosophers under fairness, based on [87].
- Model CS is the IEEE 802.3 CSMA/CD (Carrier Sense, Multiple Access with Collision Detection) protocol, which is based on [95]. Note that N is the number of stations and K is the exponential backoff limit.

In the third set of experiments, we use PAT with anti-chain approach to compare with PAT without this optimization in refinement checking. This is fair since the only difference between these two tools is the anti-chain method.

All models with configurable parameters are embedded in the latest version of PAT. In the following, we discuss the experiments in details.

3.6.1 Performance of Refinement Checking

In general, refinement checking and temporal logic verification are good at different classes of properties. For instance, using temporal logic formulae to capture the process $OKHrt$ (shown in Section 3.4) would result in a large formula which in turn result in in-efficient verification. Our experiments, however, show that even for those properties designed

System	Property	Result(Pmax)	PAT (s)	PRISM (s)
ME (N=5)	mutual exclusion	1	0.359	0.282
ME (N=8)	mutual exclusion	1	9.831	1.234
ME (N=10)	mutual exclusion	1	81.192	3.127
RC (N=4,K=4)	consensus	1	0.218	0.328
RC (N=6,K=6)	consensus	1	2.813	2.543
RC (N=8,K=8)	consensus	1	19.642	14.584
DP (N=5)	once eat, never hungry	1	3.333	37.769
DP (N=6)	once eat, never hungry	1	53.062	389.334

Table 3.1: Experiments on refinement checking

for temporal-logic based verification, probabilistic refinement checking offers comparable performance. Given any safety property of the above mentioned models, we build a property model and verify the property by refinement checking. Table 3.1 presents the experiment results. PAT performs worse than PRISM for ME, comparable for RC and better for DP. The main reason that PAT outperforms PRISM for the DP model is that PAT has less states and its refinement checking algorithm has less computation than temporal logic-based model checking. Note that because the models are designed to satisfy the properties, the result probability is all 1.

3.6.2 Performance Improvement Using Safety Recognition

Next, we show that safety recognition improves probabilistic LTL model checking and allows PAT to outperform PRISM in many cases. Safety recognition in PAT is based on syntax analysis or simple heuristics based on the generated Büchi automata. The computational overhead is negligible. Table 3.2 presents the experiment results on verifying the models against safety, co-safety and properties which are neither. Column $PAT(w)$ ($PAT(w/o)$) shows the time taken with (without) safety recognition. If the property is neither safety or co-safety, safety recognition becomes computational overhead. The cost is however negligible as evidenced in the table. For safety or co-safety properties, PAT performs better with safety recognition. In comparison with PRISM, PAT outperforms PRISM (for almost all properties) for some models, e.g., ME and RC. This is mainly because the PAT models have much less states, because of the difference in modeling. For some other models (e.g., DP and CS), safety recognition allows PAT to outperform PRISM.

In general, PRISM handles more states per time unit than PAT. Apart from the fact that

System	Property	Result(Pmax)	PAT (w)	PRISM	PAT (w/o)
ME (N=5)	co-safety	1	2.356	231.189	27.411
ME (N=8)	co-safety	1	94.204	-	8901.295
ME (N=10)	co-safety	1	1076.217	-	-
RC (N=4,K=4)	co-safety(1)	0.99935	0.379	21.954	12.150
RC (N=4,K=4)	neither	0.54282	6.106	45.612	6.087
RC (N=4,K=4)	co-safety(2)	0.15604	6.703	35.144	7.868
RC (N=6,K=6)	co-safety(1)	1	5.854	1755.984	585.706
RC (N=6,K=6)	neither	0.53228	457.815	-	442.008
RC (N=6,K=6)	co-safety(2)	0.12493	355.027	-	453.362
RC (N=8,K=8)	co-safety(1)	1	52.906	-	-
RC (N=8,K=8)	neither	0.52537	10179.796	-	10107.268
RC (N=8,K=8)	co-safety(2)	0.10138	5923.086	-	9420.430
DP (N=5)	safety	1	1.162	37.769	10.006
DP (N=6)	safety	1	9.760	389.334	164.423
DP (N=5)	co-safety	1	1.039	38.347	544.307
DP (N=6)	co-safety	1	9.091	384.231	-
CS (N=2, K=4)	co-safety(1)	1	0.615	0.921	0.736
CS (N=2, K=4)	co-safety(2)	0.99902	0.933	2.314	1.034
CS (N=3, K=2)	co-safety(1)	1	6.118	1.733	6.707
CS (N=3, K=2)	co-safety(2)	0.85962	6.284	7.233	7.484

Table 3.2: Experiments on LTL checking

PRISM has been optimized for many years, the main reason is the complexity in handling hierarchical models. Note that though these models have simple structures, there is overhead for maintaining underlying data structures designed for hierarchical systems. PRISM is based on MTBDD, whereas PAT is based on explicit state representation currently. Symbolic methods like BDD are known to handle more states [29]. Applying BDD techniques to hierarchical complex languages like PCSP# is highly non-trivial. It remains as one of our ongoing work. The experiment results are not to be taken as the limit of PAT. The fact that PAT handles less states per time unit does not imply that PAT is always slower than PRISM, as evidenced in the experiments. The main reason is that 1) a system modeled using PRISM may have more states than its model in PCSP# due to its language limitation; 2) safety/co-safety recognition which avoid much computation in probabilistic model checking.

System	Size	Verification Time (s)			#States Involved in Iterations		
		W/o AC	With AC	Gain	W/o AC	With AC	Gain
K = 2	20600	2.74	2.21	19.3%	4.2M	3M	28.6%
K = 3	45584	15.98	12.04	24.6%	18.6M	11.7M	37.1%
K = 4	86704	48.72	37.50	22.6%	55.5M	36.2M	34.8%
K = 5	117408	123.9	80.83	34.9%	130.7M	76.3M	41.6%
K = 6	231440	271.2	182.6	32.7%	272.1M	160.7M	40.9%
K = 7	342544	511.1	340.3	33.5%	515.2M	298.8M	42.0%

Table 3.3: Experiments: Probabilistic Concurrent Stack Implementation

3.6.3 Performance Improvement Using Anti-chain

Now we evaluate whether anti-chain method is indeed beneficial. We evaluate it using a modified system based on the implementation of a distributed concurrent stack example [123]. Probabilistic choices are used to model a concurrent stack model composed by *two processes*, so as to capture the situation in which the communication between different processes fails from time to time. Failures do exist in real world cases and the experiments results are summarized in Table 3.3.

We compare the efficiency of the implementation with and without (W/o) Anti-chain (AC) using several cases. *K* means length of the stack; *Size* indicates the number of states in the whole system; *#States Involved in Iterations* represents the total number of states involved in the iterative calculation. For example, a state s updates its probability in two iterations, then *#States* should increase two. From the experiments, we can see that the anti-chain approach could reduce the total number of states accumulated during the calculation, through dynamically updating states' probability based on the subset relation sub. This speeds up the verification around 29%. We remark that the gains here are not as significant as the non-probabilistic cases, because the state space cannot be reduced; however, in some cases, it does shorten the verification time.

3.7 Related work

This chapter is related to methods and tools for probabilistic system modeling and verification. Existing probabilistic model checkers include at least PRISM [80], MRMC [72] and LiQuor [35]. PRISM is the most popular probabilistic model checker. It supports a variety of probabilistic models as well as property specification languages. The input of PRISM is a simple state-based language [11]. LiQuor is a probabilistic model checker for reactive systems [35]. MRMC is a command-line based model checker for a variety of probabilistic models and a rather simple input language. The extended PAT checker complements the existing checkers by 1) offering a language that is both state-based and event-based and is capable of modeling hierarchical systems; 2) supporting both SE-LTL model checking and probabilistic refinement checking and 3) offering a user-friendly environment for not only model checking but also simulation.

The language PCSP# is related to many works on integrating probabilistic behaviors into process algebras or programs, among which the most relevant are [94, 93, 33, 145]. In [94], an extension of CSP is proposed to incorporate probabilistic behaviors in the name of refinement checking. In [93], issues on integrating probability with Event-B has been discussed. In [33], issues on integrating probability with non-determinism have been addressed. Compared to [94, 93, 33, 145], this work focuses on developing a practical tool for systematic modeling and verification of probabilistic systems.

Our work on improving temporal logic model checking with safety recognition is related to work on categorizing safety and liveness. The work presented in [6] offers theoretical results for recognizing safety and liveness given a Büchi automaton. Others have also considered the problem of model checking safety LTL properties. In [108], a categorization of safety, liveness and fairness is discussed. Further, it showed that recognizing safety LTL properties is PSPACE-complete. Later, many theoretical results and algorithms have been presented in [76], which generalizes the earlier work presented in [108]. A forward direction version of the algorithm in [76] is evidenced in [53]. In [86], the author presented a translation of safety LTL formula to a finite state automaton which detects bad prefixes. Model checking safety properties expressed using past temporal operators has been considered in [62]. Our safety recognition is based on [6, 108]. Different from the above, we present methods/algorithms which improve model checking of not only safety properties but also a class of liveness properties; not only finite state systems but also probabilistic systems.

The anti-chain approach is related to research on anti-chain based model checking. Wulf *et al.* proposed the anti-chain based approach for checking the language universality and trace refinement of NFA [136]. It has been shown that the anti-chain based approach may outperform the standard ones by several orders of magnitude. Their following works show that significant improvements can also be brought to the model checking problem of LTL by using anti-chain based algorithms [45, 137]. Later Abdulla *et al.* improved the approach through exploiting a simulation relation on the states of NFA [4]. Remotely related are anti-chain based methods for solving other problems, e.g., the LTL realizability and synthesis problem [32, 49] and the universality and language inclusion problem of tree automata [4, 26]. In our work, we focus on and probabilistic refinement checking.

3.8 Summary

In this chapter, we propose a model checker for hierarchical complex probabilistic systems. A modeling language called PCSP# is defined, which combines high-level specification language CSP with low-level imperative languages such as C# and Java. Shared variables, user-defined data types and probabilistic choices are supported in PCSP# model. We integrated this model checker to our home-grown verification framework PAT. Meanwhile, an alternative way of probabilistic system verification, i.e., refinement checking, is proposed in PCSP# verification. In our setting, this indicates the trace refinement between a probabilistic model and a non-probabilistic specification. In addition, PAT improves LTL probabilistic model checking by supporting SE-LTL and safety recognition. Moreover, anti-chain based approach is used to further improve the refinement checking in PCSP#. Various experiments are conducted to demonstrate the effectiveness and efficiency of our approach.

Chapter 4

Applying Model Checking in Multi-agent Systems

4.1 Introduction

A multi-agent system (MAS) is a system composed of multiple interacting intelligent agents within an environment. Multi-agent systems can be used to solve problems that are difficult or impossible for an individual agent or a monolithic system to solve. Because of the co-existence of multiple parties, a multi-agent system usually exhibits complicated behaviors, which can be quite complex and difficult to analyze. To have a better understanding of the system's dynamics and further optimize the system's performance, an accurate analysis of the system's behavior beforehand becomes particularly important.

Most of the existing work on analyzing MAS is based on extensive simulations [120, 38], which is the most convenient approach to take. The disadvantage of this approach is that the simulation results are usually inaccurate and also some important properties of the system (e.g., convergence) cannot be directly proved [120]. Another line of research is to analyse the system's behavior theoretically through the construction of a mathematical model of the system [129, 55, 127]. This approach has its merits in that it can give a better understanding of the system's dynamics than simulation-based approach and also the properties of the system can be proved directly. The downside is that the proof construction is in general quite tedious and usually requires a good deal of ingenuity. Moreover, in some cases, the system may be too complex such that it is impossible to construct an accurate mathematical model.

To tackle the problems in these existing approaches, we propose using model checking to analyze the behavior of an MAS. This approach is different from both simulation and mathematical analysis techniques. It is not only automatic as simulation technique does, but also provides exact rather than approximated analysis results, since it takes all possible behaviors that the system may exhibit into consideration. To make the discussion concrete, in this chapter, we focus on two important scenarios in MAS.

Robustness Analysis of Negotiation Strategy in MAS Negotiations exist in many aspects of our daily life for resolving conflicts among different parties. Generally, in an environment with multiple agents, automated negotiation techniques can, to a large extent, alleviate human effort, and also facilitate better negotiation outcomes by compensating the limited computational abilities of humans when they are faced with complex negotiations. As a result, a lot of automated negotiation strategies and mechanisms have been proposed in different negotiation scenarios [47, 106, 67].

The most commonly adopted criterion for evaluating a negotiation strategy is its *efficiency*, i.e., the average payoff it can obtain under different negotiation scenarios against other negotiation strategies. One supporting example is the annual *automated negotiating agents competition (ANAC)* [15, 14] which provides a general platform enabling different negotiation agents to be evaluated against a wide range of opponents under various realistic negotiation environments. The *efficiency* criterion currently adopted in the competition corresponds to the expected payoff under the competition tournament over all participating agents averaged over all negotiation domains. However, the *efficiency* criterion does not reveal much information about the *robustness* of the negotiation strategies in different negotiation scenarios, since it assumes that each agent's strategy is fixed (determined by its designer(s)) beforehand under the tournament. In contrast, in practical negotiations agents are free to choose any strategy available to them (e.g., from the pool of strategies developed in ANAC competitions) and may change their strategies anytime to improve their personal benefits. Given a strategy s, it is important for us to investigate whether an agent adopting s has the incentive to unilaterally deviate to other strategies under a particular negotiation tournament. Similarly, we may ask whether any agent adopting other strategies be willing to switch to s under certain negotiation domains? Because of its importance, the *robustness* criterion has been given much attention recently [14, 133].

Behavior Analysis in Dispersion Game Multi-agent learning is an important research area which has been applied in a wide range of practical domains [97, 139, 46]. One important scenario in multi-learner system is modeled as dispersion games [120]. Dispersion games are the generalization of anti-coordination games to an arbitrary number of players and actions. This class of games has received wide attentions and they have been applied to model a variety of practical applications, e.g., load balancing problems [139], and niche selection in economics such as Santa Fe bar problem [21] and minority games [43]. We focus on two novel strategies designed for dispersion games: *basic simple strategy* (BSS) and *extended simple strategy* (ESS). Previous work [120, 7] has investigated the performance of both strategies through extensive simulations and shown the convergence to an Maximal Dispersion Outcome (MDO). However, only preliminary analytical results have been provided for the analysis of both strategies, and it is particularly difficult to give very accurate analytical results.

In this chapter, we investigate how model checking techniques can be used to analyze the robustness of negotiation strategies and the behaviors of the agents under the two strategies (BSS and ESS) in the context of dispersion games in an accurate and automatic way.

However, there are still several challenges to successfully apply model checking in MAS:

- Accurate formal models are needed to represent the systems and the desired properties under investigation. The more complicated the system is, the more difficult it is to build the model.
- Dedicated model checking algorithms are needed for specific properties, e.g., robustness analysis, as existing model checking algorithms are designed for different purposes.
- The existence of multiple agents and multiple strategies in the system can easily result in state space explosion problem, which makes the verification inefficient, and thus appropriate state space reduction technique is required.

In this work, we tackle the above challenges as follows. First, we use PAT to model the complex dynamics of agents' possible behaviors. Note that according to the dynamics of the target systems, LTS is used to describe the negotiation scenarios while DTMC is used for dispersion games. Since LTS and DTMC can be viewed as specific PA, PCSP# module is suitable for both cases. Second, corresponding model checking algorithms are developed to exactly cover the properties needed for the specific properties, such as robustness analysis.

Third, since the agents always adopt the same strategy and thus exhibit similar behaviors, we propose to adopt counter abstraction technique [99, 117] to reduce the state space of the model. Counter abstraction is a special kind of symmetry reduction where the properties to be proved are irrelevant with the process identifiers. It has been proven suitable for both concurrent and probabilistic models [99, 117, 84].

For experiments, we first analyze the *robustness* of the top-eight strategies participated in the last year's ANAC competition, and rankings among them are given under different settings. The results show that our approach can greatly facilitate the robustness analysis process. Next, for dispersion games, we focus on checking two important properties of the system: convergence and convergence rate. We are able to automatically prove that the outcome is guaranteed to converge to an MDO when the agents adopt BSS while the convergence property is lost in ESS. For ESS, with probabilistic model checking we can also obtain the exact probability that the outcome deviates from an MDO by checking the corresponding property. For the property of convergence rate, the exact average number of rounds for the outcome to converge to an MDO is automatically obtained.

Compared with previous work [14, 133, 120, 38, 129, 55, 127], our contributions are threefold, as summarized below.

1. We propose a general framework to formally analyze the behaviors of MAS using model checking approach, which we believe can greatly contribute to the advancement of the MAS analysis.
2. We apply counter abstraction technique to reduce the state space of the model due to the symmetric property existing in the systems, thus making the analysis using model checking techniques both feasible and efficient.
3. We have implemented some dedicated verification algorithms in PAT, especially for robustness analysis. Several bundles of experiments are conducted to show the effectiveness and efficiency of our approach.

The remainder of this chapter is organized as follows. Section 4.2 presents relevant technical definitions. Section 4.3 investigates the formal modeling of related MASs using counter abstraction technique. Section 4.4 introduces the formal specification of desired properties. Section 4.5 evaluates our method. Section 4.6 gives an overview of related work. Section 4.7 summarizes this chapter.

4.2 Preliminaries

In this section, we recall some basic concepts and backgrounds used throughout the rest of this chapter.

4.2.1 Negotiation Model

ANAC competition is the annual competition which brings together researchers from the automated negotiation community [15, 14, 3]. Following the settings adopted in ANAC competitions, the basic negotiation form is bilateral negotiation, i.e., negotiations between two agents, under the alternating-offers protocol in which the agents take turns to exchange proposals. For each negotiation scenario, both agents can negotiate over multiple issues (items), and each item can have a number of different values. Let us denote the set of items as $\mathcal{M}$, and the set of values for each item $m_i \in \mathcal{M}$ as $\mathcal{V}_i$[1]. We define a negotiation outcome ω as a mapping from every item $m_i \in \mathcal{M}$ to a value $v \in \mathcal{V}_i$, and the negotiation domain is defined as the set Ω of all possible negotiation outcomes. We assume that the knowledge of the negotiation domain is known to both agents beforehand, and is not changed during the whole negotiation session.

For each negotiation outcome ω, different agents may have different preferences. Here we assume that each agent i's preference can be modeled by a utility function u_i such that $\forall\, \omega \in \Omega$, it is mapped into a real-valued number in the range of [0,1], i.e., $u_i(\omega) \in [0, 1]$. In practical negotiation environments, there is usually a certain cost associated with each negotiation. To take this factor into consideration, a real-time deadline is imposed on the negotiation process and each agent's actual utilities over the negotiation outcomes are decreased by a discounting factor δ over time. Following the setting adopted in ANAC'12, each negotiation session is allocated 3 minutes, which is normalized into the range of [0,1], i.e., $0 \le t \le 1$. Formally, if an agreement is reached at time t before the deadline, each agent i's actual utility function $U_i^t(\omega)$ over this mutually agreed negotiation outcome ω is defined as follows,

$$U_i^t(\omega) = u_i(\omega)\delta^t \tag{4.1}$$

If no agreement is reached by the deadline, each agent i will obtain a utility of $ru_i^0\delta$, where ru_i^0 is agent i's private reservation value in the negotiation scenario. The agents will

[1]Here $\mathcal{V}_i$ can be either discrete values or continuous real values.

also obtain their corresponding reservation values if the negotiation is terminated before the deadline. Note that the agents' actual utilities over their reservation values are also discounted by the discounting factor δ over time t. The agents' preference information and their reservation values are private and can not be accessed by their negotiating partners.

The interaction between the negotiation agents is regulated by the alternating-offers protocol, in which the agents are allowed to take turns to exchange proposals. During each encounter, if it is agent i's turn to make a proposal, agent i chooses one of the following three options: accept the offer from its negotiating partner, reject & propose a counter-offer to its negotiating partner, and terminate the negotiation.

4.2.2 Robustness Analysis using Empirical Game Theoretic Approach

Under the ANAC competitions [3, 2], the winner is the agent who receives the highest average payoffs in the specific competition tournament where each participation agent adopts a different strategy designed by different parties. Each agent's strategy in the competition cannot be changed during the tournaments. However, in practical negotiation scenarios, agents are free to choose any strategy available, thus it is equally important for us to analyze the *robustness* of different strategies. For example, we may be interested in knowing which strategy that most agents would have the incentive to adopt if the agents are free to choose any strategy.

In robustness analysis of negotiation strategies, since there exists an infinite number of possible strategies that the agents may take, the standard game-theoretic approach is not applicable because it explicitly considers all possible strategies. Empirical Game Theoretic (EGT) analysis [90] is a game-theoretic analysis approach based on a set of empirical results and can be used to investigate the robustness of the strategies. EGT handles the existence of infinite possible strategies by assuming that each agent only selects its strategy from a fixed set of strategies and the outcomes for each strategy profile can be determined through empirical simulations. This technique has been successfully applied in addressing questions about robustness of different strategies in previous years' trading agent competitions [90] and negotiation strategies in ANAC 2011 [14].

In EGT analysis, a fixed set of negotiation strategies, say S, is given first. Each agent in the system is free to select any strategy from S as its negotiation strategy. For each bilateral negotiation (p, p'), the corresponding payoff $U_p(p, p')$ received by agent p is determined as its average payoff over all possible domains against its current opponent p', which can be obtained through empirical simulations (available from ANAC competition website [3]). The average payoff of an agent in any given tournament can be determined by averaging its payoff obtained in all bilateral negotiations against all other agents participated in the tournament. Specifically, for a given tournament involving a set $\mathcal{P}$ of agents, the payoff $U_p(\mathcal{P})$ obtained by agent p can be calculated as follows,

$$U_p(\mathcal{P}) = \frac{\sum_{p' \in \mathcal{P}, p' \neq p} U_p(p, p')}{|\mathcal{P}| - 1} \tag{4.2}$$

where $U_p(p, p')$ represents the corresponding average payoff of agent p in the bilateral negotiation against agent p'. Note that agent p and p' can use either the same or different strategies. Based on Equation 4.2, now we can easily determine the corresponding payoff profile for any given tournament. An agent has the incentive to deviate its current strategy to another one if and only if its payoff after deviation can be statistically significantly improved, provided that all the other agents keep their strategies unchanged. This strategy deviation is known as *single-agent deviation*. There may exist multiple candidate strategies that an agent has the incentive to deviate to (i.e., multiple single-agent deviations exist), but here we only consider the best deviation available to that agent in terms of maximizing its deviation benefit following previous work [14], which is called *single-agent best deviation*.

Before formally defining the *robustness* of a strategy, first we need to adopt concepts from game theory to define the *stability* of a strategy profile[2]. Given a strategy profile under a negotiation tournament, if no agent has the incentive to unilaterally deviate from its current strategy, then this strategy profile is called a *empirical pure strategy Nash equilibrium*[3]. It is also possible for the agents to adopt mixed strategies and thus we can easily define the concept of *empirical mixed strategy Nash equilibrium* accordingly. However, in practical negotiations people are risk-averse and usually would like to be represented by a strategy with predictable behaviors instead of a probabilistic one [101]. Therefore we only consider *empirical pure strategy Nash equilibria* in our analysis following previous work [14]. Notice that in general a game may have no empirical pure strategy Nash equilibrium.

Another useful concept for analyzing the *stability* of the strategy profiles is *best reply cycle* [140], which is a subset of strategy profiles. For any strategy profile within this subset, there is no single-agent best deviation path leading to any profile outside the cycle. In

[2]A strategy profile refers to the combination of all participating agents' strategies.

[3]This concept is similar to the concept of *pure strategy Nash equilibrium* in classical game theory, but it is called *empirical pure strategy Nash equilibrium* since the analysis is based on empirical results.

other words, in a best reply cycle, all single-agent best deviation paths starting from any strategy profile within itself must lead to another strategy profile inside the cycle.

Both *empirical pure strategy Nash equilibrium* and *best reply cycle* can be considered as two different interpretations of empirical stable sets to evaluate the *stability* of different strategy profiles. Based on these two concepts, we are ready to evaluate the *robustness* of a strategy using the concept of *basin of attraction* of a stable set [130]. The *basin of attraction* of a stable set is defined as the percentage of strategy profiles which can lead to this stable set through a series of single-agent best deviations. Accordingly, a negotiation strategy s is considered to be *robust* if it belongs to a stable set with a large *basin of attraction* [130, 14]. In other words, if there exists a large proportion of initial strategy profiles which, through a series of single-agent best deviations, can eventually lead to a stable set containing strategy s, then strategy s is highly robust in the long run, since the strategy s can always have the opportunity of being adopted eventually if the tournament is sufficiently repeated due to single-agent best deviations.

In summary, we apply the EGT analysis to identify both *empirical pure strategy Nash equilibrium* and *best reply cycle* and evaluate the *robustness* of negotiation strategies via the concept of *basin of attraction*.

4.2.3 Dispersion Game and Strategies Definition

Dispersion games (DGs) [120] generalize the anti-coordination games by allowing arbitrary number of players and actions. In this class of games, the agents prefer the outcomes in which their action choices are as dispersed as possible over all possible actions. Formally a N-player dispersion game is a tuple $\langle N, (A_i), (u_i)\rangle$ where

- $N = \{1, 2, \ldots, n\}$ is the set of agents.
- A_i is the set of actions available to agent i.
- u_i is the utility function of each agent i, where $u_i(O)$ corresponds to the payoff agent i receives when the outcome O is achieved.

We assume that all agents have the same set of actions, that is, $A_1 = A_2 = \ldots = A_n$, and also the game is both agent symmetric and action symmetric. That is, each agent's utility over a particular outcome is only determined by the number of agents choosing the same action as itself.

When the agents are interacting with one another in DG-like environments, the most desirable outcomes would be the case that all agents' action choices are as dispersed as possible, from both individual agent's and the overall system's perspectives. This kind of outcomes is called maximal dispersion outcomes (MDOs) [120]. Formally, an MDO can be defined as follows.

Definition 9 *Given a DG, an outcome $O = \{a_1, \ldots, a_i, \ldots, a_n\}$ is* ***maximal dispersion outcome*** *iff for each agent $i \in N$ and each outcome $O' = \{a_1, \ldots, a_i', \ldots, a_n\}$ such that $a_i \neq a_i'$, we have $n_{a_i}^{O} \leq n_{a_i'}^{O'}$. Here $n_{a_i}^{O}$ and $n_{a_i'}^{O'}$ are the number of agents choosing action a_i and a_i' under outcome O and O' respectively.*

The strategies we consider here are *basic simple strategy* (BSS) and *extended simple strategy* (ESS). *Basic simple strategy* is a novel strategy for agents to make decisions in repeated DGs proposed by Alpern [7]. This strategy is specifically designed for the case when the number of agents n is equal to the number of actions $k (k = |A_i|)$. According to BSS, initially each agent i chooses a random action. If no other agent chooses the same action, agent i will still choose the same action next round. If there exist other agents choosing the same action, agent i will randomly choose an action from the set $A' = \{a' \in A_i \mid n_{a'}^{O} \neq 1\}$ of actions in the next round. Note that this strategy only requires that the agents know which actions are chosen by only one agent in previous round.

Another strategy we consider is *extended simple strategy*, which extends BSS for the general case when $n \neq k$. In each round t, each agent i chooses the same action a_i as previous round if $n_{a_i}^{O_t} \leq \lfloor n/k \rfloor$, where $n_{a_i}^{O_t}$ is the number of agents choosing a_i in round t. Otherwise, agent i chooses action a_i with probability $\frac{n/k}{n_{a_i}^{O_t}}$ and with probability $1 - \frac{n/k}{n_{a_i}^{O_t}}$ randomly chooses an action over the action set $\{a' \in A_i \mid n_{a'}^{O_t} < \lceil n/k \rceil\}$.

Unlike BSS, ESS does not assign equal probability to those actions that are not chosen by only one agent. For example, consider the case when there are 4 agents and the action set $A_1 = A_2 = \ldots = A_4 = \{a_1, a_2, a_3, a_4\}$, and the outcome in the current round t is $O_t = \{a_1, a_1, a_2, a_2\}$. In ESS, the agents choosing action a_1 in current round t will choose action a_1 with probability 0.5 and either action a_3 or a_4 with probability 0.25 in round $t + 1$. In contrast, the agents will randomly select one action to perform according to strategy BSS.

4.2.4 Counter Abstraction Technique

Counter abstraction is a special kind of symmetry reduction where the properties to be proved are irrelevant with the process identifiers. If a system is composed of a large number of behaviorally similar or even equal processes, we can abstract its state space by grouping the processes based on which local state they reside in. For example, suppose there are 3 behaviorally identical processes residing in a system. Instead of saying "process 1 is in state s, process 2 is in state t and process 3 is in state s", we simply say "two processes are in state s and one process is in state t". In this way, the state space can be reduced by exploiting a powerful state space symmetry.

4.3 Modeling with Counter Abstraction

4.3.1 Modeling Negotiation Systems

Given a set of agents denoted as N and a set of negotiation strategies denoted as S, the n-agent negotiation problem can be naturally modeled as a strategic form game. Formally it can be represented as a tuple $\langle N, (S_i), (U_i)\rangle$ where

- $N = \{a_1, a_2, \ldots, a_n\}$ is the set of agents.
- S_i is the set of negotiation strategies available to a_i, and we assume that all agents have the same set of strategies, that is, $S_1 = S_2 = \ldots = S_n = S$.
- U_i is the utility function of agent i, where $U_i(\mathcal{P})$ corresponds to the average payoff a_i receives, given the set of agents involved in the current negotiation is $\mathcal{P}$, which can be calculated according to Equation 4.2.

Each agent (process) a_i has its own negotiation strategy $s_i \in S_i$, and each global state $s = (v, \langle s_1, \ldots, s_n\rangle) = (v, O_i)$, which is the combination of the valuations of the global variables v[4] and the chosen strategies of all agents (or the game outcome O_i). If global variables are ignored, then each global state represents a unique strategy profile in the system. The transition relation T is built based on the single-agent best deviation, i.e., an

[4]Here the global variables refer to all variables defined in the model apart from the local variables $(s_1, \ldots, s_n)$ storing the strategy choices for each agent.

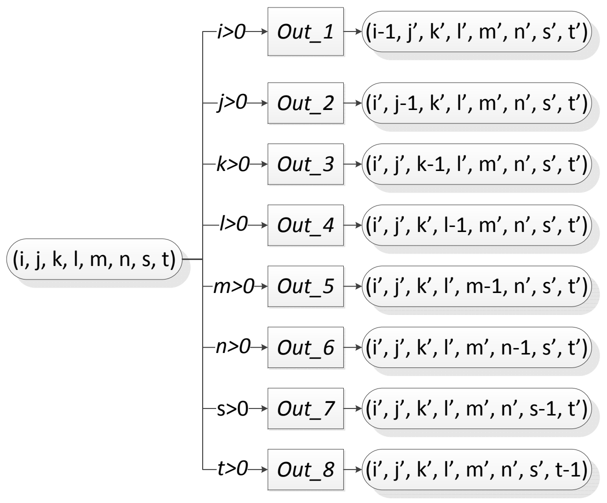

(i, j, k, l, m, n, s, t)
i>0
Out_1
(i-1, j', k', l', m', n', s', t')
j>0
Out_2
(i', j-1, k', l', m', n', s', t')
k>0
Out_3
(i', j', k-1, l', m', n', s', t')
l>0
Out_4
(i', j', k', l-1, m', n', s', t')
m>0
Out_5
(i', j', k', l', m-1, n', s', t')
n>0
Out_6
(i', j', k', l', m', n-1, s', t')
s>0
Out_7
(i', j', k', l', m', n', s-1, t')
t>0
Out_8
(i', j', k', l', m', n', s', t-1)

Figure 4.1: One Step of the Negotiations

agent may change its current strategy to another one according to the maximal deviation benefit. Thus the formal model representing the dynamics of the negotiations can be automatically constructed and is uniquely determined.

However, in EGT analysis, the agents always choose strategies from the same strategy group and the identities are not important, therefore this abstraction technique can be naturally applied here. Specifically, we only need to consider how many agents choose each strategy at the same time. Previously, given two states in which the number of agents choosing each strategy are the same, but the identities of the agents choosing the same strategy are different, they are defined as different states. But now they can be merged as the same one. For example, consider an EGT analysis with 5 agents and 3 strategies and two possible global states $s = (v, \langle s_1, s_1, s_2, s_2, s_3 \rangle)$ and $s' = (v, \langle s_1, s_2, s_2, s_3, s_1 \rangle)$. We only need to keep track of the number of agents choosing each strategy, i.e., we have $f(s_1) = 2, f(s_2) = 2, f(s_3) = 1$, where $f(s)$ records the number of agents choosing strategy s, and thus the two original global states are reduced to a single one (v, f). Following the above idea, we can reduce the state space of multi-agent negotiation models.

Next, we take the 8-agent 8-strategy negotiations as an example, which is the setting adopted in the final round competition of ANAC, to show the formal modeling of the system. The group of strategies is denoted as $S = \{s_1, s_2, \cdots, s_8\}$. Since the counter abstraction is used, we do not keep record of individual agent's strategy. Instead, states representing the overall strategies distribution between agents are defined. We take one step of the dynamic behaviors of the system as an example, which is demonstrated in Fig. 4.1.

In Fig. 4.1, the arrows indicate the direction of the states transition. (i, j, k, l, m, n, s, t) is used to represent a strategy profile in the system. i means currently there are i agents choosing s_1 and m means there are m agents choosing s_5. Obviously the sum of these integers should be 8. This combination is actually a state in the corresponding LTS. For the whole system, given a strategy profile, there are at most 8 enabled outgoing transitions. This is because each transition corresponds to the single-agent best deviation for each individual strategy. Therefore, there are 8 outgoing arrows from state (i, j, k, l, m, n, s, t) displayed in the figure.

let us take the uppermost arrow in the figure as an example. This is a potential transition which means one agent choosing s_1 tries to deviate to another strategy according to the best payoff it can achieve. If this transition can happen, i must be positive, otherwise no agent can abandon s_1. A label $i > 0$ is used on the arrow to represent this constrain. Assume this condition is true, then there are some agents currently choosing s_1, and one of them may have the incentive to change s_1 to another strategy which can mostly increase its payoff. Which strategy will be the agent's new choice? The answer is decided via the negotiation procedure represented by Out_1, which means there will be an agent replacing its strategy s_1 to another. This algorithm can be implemented by imperative programming languages and imported to model checker. Afterwards, a new state *(i-1, j', k', l', m', n', s', t')* will be generated since a new strategy profile is obtained. Because one agent abandons s_1, then i becomes *i-1*. One of the other 7 integers will increase by 1, but we are not sure which one it is.

After defining the transition rules of an individual state, one critical question is that what the initial distribution of the strategies is among the agents, i.e., the initial strategy profile. This is very important since it affects the following executions. For example, (8, 0, 0, 0, 0, 0, 0, 0) and (0, 0, 0, 0, 0, 0, 0, 8) have totally different behaviors in their future executions because of the different payoff of different strategies. For robustness analysis, we should consider all possible scenarios of the system, i.e., it is better that the initial states cover all strategies profiles. Different model checkers may have different solutions to handle this issue. For example, if a model checker supports the declaration of multiple initial states, then all possible strategy profiles can be defined as initial states and the system behaviors can be analyzed afterwards. Or in some cases, the model checker just supports one initial state, then some tricks should be used to guarantee that this single initial state can transit to all possible strategy profiles.

4.3.2 Modeling BSS and ESS in Dispersion Games

For both BSS and ESS in DGs, in each round, the agents simultaneously choose their actions in a probabilistic manner based on the outcome of the previous round. The natural way of modeling the agents' dynamics in DGs is to represent each agent's learning dynamics as a process. The overall system exhibits highly stochastic behaviors and non-determinism because of the coexistence of multiple probabilistic learners. However, since each agent makes its decision independently each round, the concurrent behaviors among agents can be equivalently modeled as a series of sequential behaviors. In this way, the non-determinism in the system is eliminated and thus the system can be naturally modeled as a DTMC. On the other hand, similar to the negotiation system, agents in DG also have identical behaviors since they share the common actions. It is unnecessary to model each individual agent. Therefore, counter abstraction technique is suitable again.

We model each action instead of each agent as a process in model implementation, and only record the number of agents choosing this action. Each action process's behavior is determined by the stochastic behaviors of all agents previously choosing it. The current local state of each action (process) is represented by the number of agents currently choosing it, which will be updated accordingly based on the stochastic behaviors of the relevant agents. If there is a new agent choosing action a_i, then the variable recording the local state of action a_i will be increased by 1. Each global state of the system is determined by the local states of all the action processes (the game outcome) together with all global variables. Fig. 4.2 shows the behaviors of the model for ESS with $\mid A_i \mid= 2$ and any number of agents. In this model, two processes, Action 1 and Action 2, are executing in parallel, and are also synchronized at the end of each round. For each action process i, its current local state is represented by the number n_i of agents choosing it in the current round. The execution path of each process is determined by its current local state and the behaviors of the agents choosing it. Specifically, each process i repeatedly checks whether there is any agent that takes action i in current round but has not made its next round decision yet. If yes, the process proceeds by allowing this agent to make its decision in the way as specified by ESS and makes update accordingly; if not, the process waits, updates its local state and starts the next round after the other process also finishes this round. The behaviors of the model for BSS are similar and we omit it here.

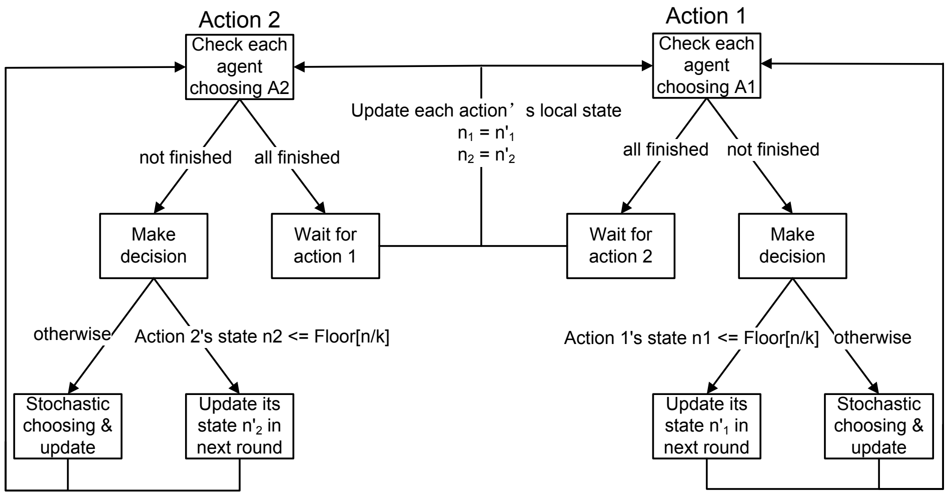

Action 2
Check each agent choosing A2
Action 1
Check each agent choosing A1
Update each action’s local state
n1 = n'1
n2 = n'2
not finished
all finished
all finished
not finished
Make decision
Wait for action 1
Wait for action 2
Make decision
otherwise
Action 2's state n2 <= Floor[n/k]
Action 1's state n1 <= Floor[n/k]
otherwise
Stochastic choosing & update
Update its state n'2 in next round
Update its state n'1 in next round
Stochastic choosing & update

Figure 4.2: Finite state automaton of the model of ESS with $| A_i |= 2$

Besides, by using two sets of variables to record the local states of the action processes (e.g, n_1 and n_1' in Fig. 4.2), the behavior of each process does not have any side effect on the behaviors of other processes. Therefore the updating of each action process can be performed in a sequential way without involving any non-determinism, and thus it is sufficient to model the system as DTMC instead of PA as previously mentioned.

4.4 Properties Specification

After building the model, another important issue is formally verifying suitable properties, which can be used to analyze the behaviors of MAS.

4.4.1 Properties in Negotiation Systems

From the analysis of Section 4.2.2, *empirical pure strategy Nash equilibrium*, *best reply cycle* and the resulting *basin of attraction* should be verified. In the following we describe the corresponding model checking algorithms used to check these properties.

Empirical Pure Strategy Nash Equilibrium The existence of *empirical pure strategy Nash equilibrium* means there exists some states that all agents in the negotiation setting will keep their strategies, so that no outgoing transition exists from these states. From the viewpoint of model checking, these states are *deadlock* states in the system. On the other hand, each state in the system is corresponding to a strategy profile, therefore each deadlock state existing in the model indicates the agents will not change their mind, and this state should satisfy *empirical pure strategy Nash equilibrium*. As a result, deadlock checking can be used to check whether *empirical pure strategy Nash equilibrium* exists in the system. In traditional deadlock checking, the verification algorithm stops whenever a deadlock state is found, and returns a counterexample. Otherwise there is no deadlock state existing. In our setting, we require to find all deadlock states instead of one in order to completely analyze the robustness. So we improve the traditional deadlock checking algorithm to capture all deadlock states, if there is any.

Best Reply Cycle *Best reply cycle* describes the scenario that there are several states which compose a loop, and these states do not have outgoing transitions to states outside this loop. In this case, states in the loop cannot reach states which correspond to empirical pure strategy Nash equilibrium since they can always transit to other states. From the viewpoint of model checking, the desired loops are actually nontrivial BSCCs in the state space. And on the other hand, it is trivial that all nontrivial BSCCs should be *best reply cycle*. Therefore our target is to find out all nontrivial BSCCs. SCC searching is widely used in model checking techniques, especially for LTL verification. Tarjan's SCC searching algorithm [122] can be applied here to find all nontrivial SCCs, and BSCCs are restored as our targets.

Basin of Attraction Based on the above two properties, the stability of different strategy profiles can be decided. All *empirical pure strategy Nash equilibria* and *best reply cycles* compose the stable sets of the negotiation system. Accordingly, *basin of attraction* of a stable set is defined as the percentage of strategy profiles which can lead to itself through a series of single-agent best deviations, i.e., the percentage of the states in the system reaching the deadlock states or BSCCs. It can be calculated based on the results of the above two properties, since the total number of states reaching each stable set can be recorded. One strategy is robust iff it belongs to stable sets with large *basin of attraction*.

4.4.2 Properties in Dispersion Games

In DGs, We mainly concentrate on the following three properties: convergence, deviation and convergence rate.

Convergence Given a strategy in DG, convergence indicates whether the agents adopting this strategy are guaranteed to converge to an MDO at last. This is an important property to analyse in DG and in the literature of learning in games as well [120, 7]. If the answer is positive, it indicates that the strategy can let the agents to stabilize on the most efficient outcome eventually. To verify this property, we can check the probability that the system satisfies the LTL formula as follows:

$$\mathcal{P}_r(System \models \Diamond\Box\, MDO); \tag{4.3}$$

where *System* models the overall system. *MDO* is a combination of atomic propositions, and states satisfying *MDO* represent the outcome of the game in these states is an MDO. The combination of $\Diamond$ and $\Box$ operators captures the meaning of convergence, i.e., the system will eventually reach MDO and will always be in that state thereafter. Through verifying property 4.3, the probability that the system converges to an MDO is obtained.

Deviation In some scenarios, convergence does not exist in DG. This means the outcome is an MDO in round t, but the outcome in round $t+1$ is not an MDO. In other words, the outcome deviates from an MDO. For a given strategy, if this average probability of deviation is very low, we can say that the outcome approximately converges to an MDO under this strategy. To achieve this goal, we can check the probability that the system satisfies the LTL formula as follows.

$$\mathcal{P}_r(System \models \Box(MDO \rightarrow \mathbf{X}\neg MDO); \tag{4.4}$$

This formulae represents the probability from an MDO to an output which is not an MDO in the 'next' round. Through verifying property 4.4, the probability that the system deviates from an MDO is obtained.

Convergence Rate Next we consider analyzing another important property of the strategies: the convergence rate. For any strategy that has been proven to (approximately) converge to desirable outcomes, the natural following up question would be how fast the convergence could be. Here we illustrate how we can analyze this property of both strategies using **reward checking** [18] techniques. Here we skip the formal definition of reward checking since we just use the simplest setting: we use transition reward to calculate the average rewards from the initial state to an MDO. Specifically, we set the reward of finishing each round is 1, and increase the accumulated reward by one after each round. Using the iterative method in [18], we can calculate the average rounds (rewards) from the initial state to an MDO. The property can be expressed as follows.

$$\mathcal{R}(System \models \diamond MDO); \tag{4.5}$$

where R indicates the rewards of reaching some target states. Through checking the above property, the exact average number of rounds the system takes to converge to an MDO can be obtained.

4.5 Evaluation

All proposed algorithms in this chapter have been implemented in PAT. In this section, we evaluate our approach in negotiations and dispersion games via several sets of experiments.

4.5.1 Negotiation Systems

We perform robustness analysis on the top eight strategies participated in ANAC 2012 using PAT, and our own strategy *CUHKAgent* is included. *CUHKAgent* is an implementation of our adaptive negotiating strategy *ABiNeS* [61] for bilateral negotiations participated in ANAC 2012 [3], and wins the first place of ANAC 2012. The set of top agents are represented as $\mathcal{S} = \{\mathbf{C, L, O, R, B, M, I, A}\}$[5]. We use the notation $\mathcal{P}$ to represent the set of agents participating in the negotiation.

The detailed payoff matrix for all possible bilateral negotiations is given in Table 4.1 which is available from `http://anac2012.ecs.soton.ac.uk/` and will be used as the basis for performing the robustness analysis. Next, multiple experiments are conducted under

[5]The bold letters are the abbreviations for each strategy as follows: **C** - CUHKAgent, **L** - AgentLG, **O** - OMACAgent, **R** - TheNegotiatorReloaded, **B** - BRAMAgent2, **M** - Meta-agent, **I** - IAMHaggler2012, **A** - AgentMR. These abbreviations will be used in the following descriptions.

Table 4.1: Payoff matrix for the top eight negotiation strategies in ANAC 2012 average over all domains (For each strategy profile, only the row agent's payoff is given since the game is symmetric.)

$U(p, p')$	**C**	**L**	**O**	**R**	**B**	**M**	**I**	**A**
C	0.596	0.465	0.491	0.669	0.548	0.618	0.832	0.437
L	0.541	0.421	0.439	0.673	0.462	0.640	0.832	0
O	0.533	0.38	0.423	0.648	0.433	0.562	0.815	0
R	0.546	0.522	0.502	0.576	0.509	0.596	0.773	0.425
B	0.523	0.357	0.414	0.657	0.463	0.648	0.757	0.207
M	0.501	0.486	0.472	0.623	0.484	0.456	0.76	0.079
I	0.559	0.567	0.55	0.578	0.531	0.592	0.819	0
A	0.471	0	0	0.615	0.163	0.12	0	0

different negotiation settings to analyze the robustness of different strategies and illustrate the effectiveness of our approach. All these experiments will be shown to be finished within seconds, and thus the efficiency of this method is satisfiable.

4.5.1.1 Bilateral Negotiations among Eight Possible Strategies

In the context of bilateral negotiations, there exists 2 agents, i.e., $|\mathcal{P}| = 2$, and the strategy set $\mathcal{S}$ is defined as above. Each agent can choose any strategy from the set $\mathcal{S}$ during negotiation. Through the counter abstraction approach, the state space of the negotiation model is reduced from $|\mathcal{S}|^{|\mathcal{P}|} = 8^2 = 64$ to $\binom{|\mathcal{P}|+|\mathcal{S}|-1}{|\mathcal{S}|-1} = 36$. This reduction can also help to reduce the verification time of PAT. The verification results are listed as follows.

- Verification time: 0.1 second.
- Deadlock states: there is no such states.
- BSCCs: there is only one nontrivial BSCC existing in the system: $(1, 1, 0, 0, 0, 0, 0, 0) \rightarrow (0, 1, 0, 1, 0, 0, 0, 0) \rightarrow (1, 0, 0, 1, 0, 0, 0, 0) \rightarrow (1, 1, 0, 0, 0, 0, 0, 0)$.
- States reaching deadlock or BSCCs: 36 states reach the BSCC; no state reaches deadlock states.

According to the verification results, we find that under bilateral negotiations, there is no *empirical pure strategy Nash equilibrium* and there exists only one *best reply cycle*, i.e.,

Table 4.2: The robustness ranking of strategies in bilateral negotiations.

Strategy	**C**	**L**	**O**	**R**	**B**	**M**	**I**	**A**
Ranking	1^{st}	1^{st}	4^{th}	1^{st}	6^{th}	5^{th}	6^{th}	8^{th}

$(1,1,0,0,0,0,0,0) \rightarrow (0,1,0,1,0,0,0,0) \rightarrow (1,0,0,1,0,0,0,0) \rightarrow (1,1,0,0,0,0,0,0)$. Here 0 and 1 indicate the number of agents choosing each strategy, and the order of these numbers in one state is consistent with the strategies order listed in $\mathcal{S}$. Besides, the *basin of attraction* of this cycle is 100%, i.e., for all possible initial strategy profiles, there always exists a single-agent best deviation path which can lead to one of the strategy profiles within this cycle. Our strategy *CUHKAgent* (**C**) is contained in two strategy profiles ($(1,1,0,0,0,0,0,0)$ and $(1,0,0,1,0,0,0,0)$) in this cycle. This indicates that **C** is very robust against other strategies since it is always possible that the agents will be willing to adopt **C** no matter what their initial strategy is in the long run. Obviously, strategies **L** and **R** are also robust in current environment while others may be less competitive in the experiments.

Next, it is meaningful to investigate the overall ranking of these top strategies in current setting according to their robustness. To achieve this goal, we adopt the elimination mechanism used in [27]. Different from [27] which eliminates the worst player, we gradually eliminate the most robust strategies available to the agents in the experiment, and try to find the most robust strategies in the remaining ones. Note that the robustness of one strategy may be related with some opponents' performance. The ranking of a given strategy may be affected by the presence, or absence, of other strategies, and may depend on the number of agents in the system. Therefore this kind of ranking can only be used as a reference.

According to the existence of *best reply cycle*, we can conclude that strategies **C**, **L**, and **R** rank the 1^{st} in all strategies in current system. Afterwards, these three strategies are removed from $\mathcal{S}$, and the robustness analysis will be conducted in the remaining five strategies. Step by step, the overall robustness ranking of these top strategies in bilateral negotiations will be obtained, as listed in table 4.2. We can see that **O** and **M** have the average robustness, while **B**, **I** and **A** are relatively not so robust. Therefore, the agents having the last three strategies will try to change their choices to get better payoff.

The analysis within bilateral negotiation does not give us much information about the robustness of our strategy within a tournament setting involving more than two agents. Therefore, more cases are discussed in the following part.

4.5.1.2 Eight-agent Negotiation Tournaments

We start with a simple case in which each agent is only allowed to choose one strategy from the top four strategies[6], and then we perform the robustness analysis by taking all top eight strategies into consideration later, following the setting of ANAC competition. For both cases, each participating agent negotiates with all the other participates, and the average payoff of each agent can be determined using Equation 4.2. However, different from the setting in ANAC competition, the agents are free to choose any strategy from the set of strategies available. Different agents may select the same strategy during the same tournament in our EGT analysis. The setting of ANAC competition can be considered as a specific tournament in which each agent chooses a unique strategy among the eight strategies.

One natural way is to perform robustness analysis over all domains. However, this can hide a lot of detailed information due to the averaging effects. Besides, most of the domains are relatively small and thus easier for the agents to negotiate to get a high utility under the limited negotiation time (3 mins). Therefore, we conduct the robustness analysis under the tournament setting over one challenging domain: *Travel* domain, similar to previous work [133]. The *Travel* domain is one of the largest and most complex domains in the competition, which thus can better reflect the practical negotiation scenarios which usually involve a large number of possible proposals to consider. In addition, different from [133], we set the discounting factor of this domain to the low value of 0.5 instead of 1 (without discounting). Under the setting with high discounting effect, it requires the agents to delicately and adaptively trade off between concession to the opponent (be fear of obtaining lower payoff due to large discounting effect) and staying tough (hope to get higher payoff by letting the opponent concede first) against different types of opponents. Therefore we believe that this setting can better reflect the behavior differences between different strategies.

Negotiation Tournament Analysis over Top Four Strategies In this section, we consider the eight-player negotiation tournament over the set $\mathcal{S}' = \{\mathbf{C}, \mathbf{L}, \mathbf{O}, \mathbf{R}\}$ of the top tour strategies. Similar with the analysis in bilateral negotiation, the total number of strategy profiles considered can be reduced from $|\ \mathcal{S}'\ |^{|\mathcal{P}|} = 4^8 = 65536$ to $\binom{|\mathcal{P}|+|\mathcal{S}'|-1}{|\mathcal{S}'|-1} = 165$ considering the symmetry of the negotiation. The verification results from PAT is listed as follows.

[6]The reason that we choose the top-four strategies instead of the top-three is that both *OMACagent* and *TheNegotiatorReloaded* rank the third place.

- Verification time: 1.2 seconds.
- Deadlock states: there is only one deadlock state in the system: $(8, 0, 0, 0)$.
- BSCCs: there is no nontrivial BSCC in the system.
- States reaching deadlock or BSCCs: no stat reaches nontrivial BSCC; 165 states reach the deadlock state.

Based on the verification results, we observe that there is only one *empirical pure strategy Nash equilibrium*, $(8, 0, 0, 0)$, in which all agents adopt our strategy *CUHKAgent*. From the percentage of the states reaching this deadlock state, we can see that *basin of attraction* of *CUHKAgent* is 100%. In other words, for all non-equilibrium tournaments, there always exists a single-agent best deviation path leading to this equilibrium. This result indicates that our strategy *CUHKAgent* is very robust under the top four-agent negotiation tournament, even though the agents' average payoff under this equilibrium tournament is lower compared with some other tournaments (e.g., all agents adopting the strategy *IAMHaggler2012*). Similar phenomena (the inefficiency of Nash equilibrium from the social perspective) are commonly observed in non-cooperative game theory. For example, in the prisoner's dilemma game, mutual defection is the only pure strategy Nash equilibrium, but there exists another Pareto-optimal outcome of mutual cooperation under which all agents' payoffs can be significantly increased.

We give a specific illustration of how the agents adjust their strategy choices between different tournaments under the EGT analysis in Fig. 4.3. This deviation analysis graph is generated automatically using the model checker PAT's simulator. In Fig. 4.3, each node is associated with a unique number representing different tournaments. The strategy profile for each state is given by the label of its outgoing transitions. Each transition indicates a single-agent best deviation for one particular type of strategy. For example, node 2 represents the tournament in which the number of agents choosing the four strategies **C, L, O, R** are 3, 1, 2, 2 respectively, which is specified by the last 4 digits of the labels of its outgoing transitions, i.e., LC.**3.1.2.2**, RC.**3.1.2.2** and OC.**3.1.2.2**. The transition from node 2 to node 27 indicates that there exists a single-agent best deviation for an agent choosing strategy **L** to deviate to strategy **C**, as indicated by its label **LC**.3.1.2.2. From Fig. 4.3, we can clearly observe the overall trend that all agents choosing strategies different from **C** have the incentive to deviate from their current strategies to strategy **C**. This deviation analysis graph can be viewed as a graph with six levels, because the shortest pathes from state 1 to state 13 has six transitions, such as the paths $1 \to 2 \to 5 \to 8 \to 10 \to 12 \to 13$

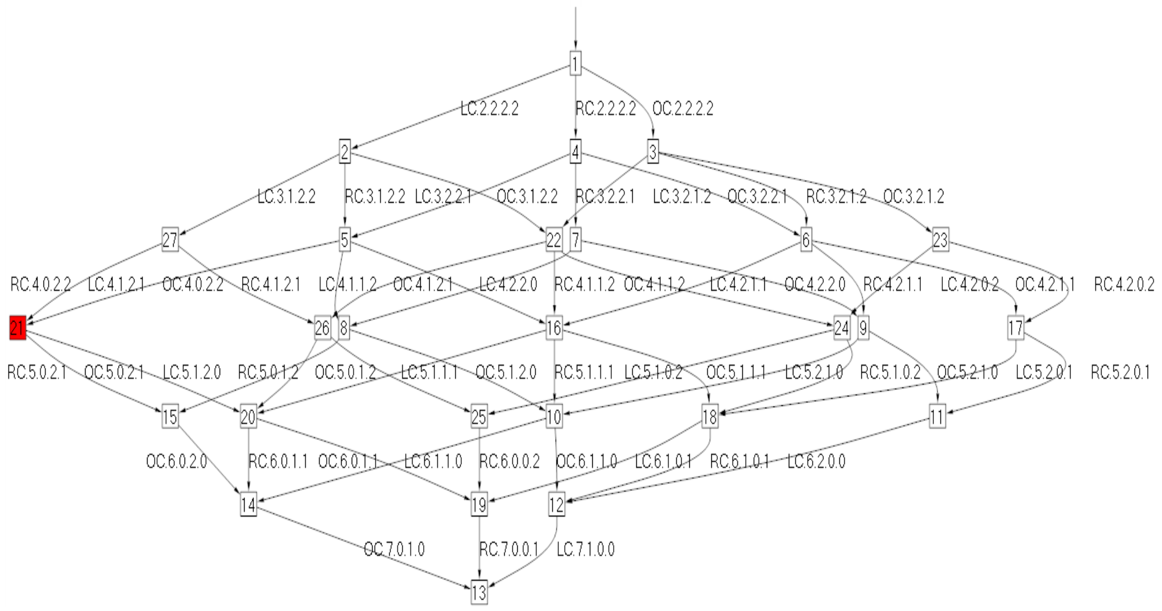


Figure 4.3: Deviation analysis graph with initial state (node 1) in which each strategy is chosen by two agents

and $1 \rightarrow 4 \rightarrow 7 \rightarrow 8 \rightarrow 15 \rightarrow 14 \rightarrow 13$. The transitions between every adjacent levels i and j correspond to single-agent best deviations from states in level i to states in level j. At the last level, all agents originally choosing strategies different from **C** have switched to strategy **C**, thus converging to the terminating node 13, which corresponds to the pure strategy Nash equilibrium $(8, 0, 0, 0)$.

Negotiation Tournament Analysis over Top-eight Strategies In this section, we turn to the analysis of the complete eight-player tournament over the top eight strategies following the setting of ANAC 2012. In this setting, the total number of strategy profiles considered can be reduced from $| \mathcal{S} |^{|\mathcal{P}|} = 16777216$ to $\binom{|\mathcal{P}|+|\mathcal{S}|-1}{|\mathcal{S}|-1} = 6435$ due to symmetry of the negotiation. The verification results are listed as follows.

- Verification time: 10.2 seconds.
- Deadlock states: there is only one deadlock state in the system: $(8, 0, 0, 0, 0, 0, 0, 0)$.
- BSCCs: there is no nontrivial BSCC in the system.
- States reaching deadlock or BSCCs: no state reaches nontrivial BSCC; all states reach the deadlock state.

Table 4.3: The robustness ranking of strategies in eight-agent negotiations.

Strategy	**C**	**L**	**O**	**R**	**B**	**M**	**I**	**A**
Ranking	1^{st}	2^{st}	6^{th}	4^{st}	4^{th}	6^{th}	2^{th}	8^{th}

The verification results are similar to results in the previous case. There also exists only one *empirical pure strategy Nash equilibrium*, $(8, 0, 0, 0, 0, 0, 0, 0)$, where all agents adopt our strategy *CUHKAgent*, and also the *basin of attraction* of this equilibrium is 100%. This indicates that our strategy *CUHKAgent* is also very robust under the full negotiation tournament setting over top eight strategies. Similar deviation trends as the previous case can also be found here: for any initial tournament, those agents choosing strategies other than strategy **C** have the incentive to switch to strategy **C** to maximally increase their individual average payoffs, and thus after a series of single-agent best deviations, finally the strategy profile will converge to the pure strategy Nash equilibrium $(8, 0, 0, 0, 0, 0, 0, 0)$. The deviation analysis graph is omitted here due to space limitation.

Again, we want to get the overall robustness ranking of these eight strategies in current eight-agent negotiations. Because *CUHKAgent* is the most robust one, it will be eliminated and robustness analysis will be conducted in the remaining 7 strategies. Gradually, the overall robustness ranking of these top strategies in eight-agent negotiations will be obtained, as listed in table 4.3. We can see that different from the ranking in Table 4.2, strategy **I** is quite robust; **L** still has excellent robustness; **R** and **B** have the average robustness; and **O**, **M** and **A** have relatively bad robustness in current setting, therefore agents having these three strategies may try to abandon them if they have other choices.

We notice that the robustness rankings in the settings of bilateral negotiation (Section 4.5.1.1) and tournament negotiation (Section 4.5.1.2) are different. This difference can be explained as follows: in the bilateral negotiation setting, the possible payoff increase from deviation is caused by the negotiation efficiency difference between every pair of strategies over each other. However, in the tournament negotiation setting, the possible payoff increase from deviation is from the efficiency difference between each pair of strategies over the rest of strategies being considered. Therefore the best deviation strategy in the first setting may not be the best choice when it comes to the second setting.

Table 4.4: Probability of Convergence to an MDO of ESS

$P_c(n,k)$	n=2	n=3	n=4	n=5	n=6	n=7	n=8	n=9	n=10
k=2	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0
k=3	1.0	1.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0
k=4	1.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0

4.5.2 BSS and ESS in Dispersion Games

In this part, we analyze the performance of both strategies BSS and ESS with probabilistic model checking.

4.5.2.1 Convergence

We first consider the property whether the agents adopting the strategies BSS and ESS are guaranteed to converge to an MDO.

For the first strategy BSS, it is required that the number n of players is always equal to the number of actions $\mid A_i \mid$. Due to the state space explosion with the increase of problem size, we only consider three simple cases with $n = 3, 4$ and 5. By automatically verifying the previous property, the model checker PAT returns that the probability that the outcome converges to an MDO is 1 for all cases.

For the second strategy ESS, the number of players can be different from the number of actions. We consider the following cases here: the number of available actions $k = 2, 3, 4$, and the number of players n varies from 2 to 10. Table 4.4 shows the convergence probability $P_c(n,k)$ for all these cases by automatically verifying property 4.3.

From Table 4.4, we can see that the strategy ESS cannot guarantee that the outcome will always converge to an MDO, and it depends on the relation between the number of agents and actions available to them. Specifically, for $k = 2, 3, 4$ and $n = 1, ..., 10$, the outcome is guaranteed to converge to an MDO when $k \geq n$ or $n\%k = 0$, and the convergence property does not hold otherwise. Intuitively, the underlying reason is as follows: when $n\%k \neq 0$ and $n > k$, for any MDO, there always exists an unstable action such that the agents choosing this action would always have certain probability to choose other actions next round. If all agents choose another action next round simultaneously, the resulting outcome will not be an MDO any more.

Table 4.5: Probability of Deviation after reaching an MDO

$P_c(n,k)$	n=3	n=4	n=5	n=6	n=7	n=8	n=9	n=10
k=2	0.0625	0.0	0.0699	0.0	0.0746	0.0	0.0686	0.0
k=3	0.0	0.0725	0.125	0.0	0.0912	0.1368	0.0	0.092
k=4	0.0	0.0	0.122	0.1524	0.1583	0.0	0.1374	0.1573

4.5.2.2 Deviation

For the cases when the convergence property is lost in ESS, it is interesting and useful to consider the corresponding deviation probability. By checking property 4.4, we can automatically obtain the exact probability that the deviation happens. This property is verified for all previous cases that the convergence property is lost in ESS, and the results are shown in Table 4.5.

In Table 4.5, the cases with value 0 indicate that the outcome will always converge to an MDO, which is in accordance with previous results. For the rest of cases that the convergence property is lost, the exact probability that the outcome will deviate from an MDO is obtained. These results give us a better understanding and more accurate prediction of the dynamics of the agents' behaviors.

4.5.2.3 Convergence Rate

To analyze the convergence rate of the strategy BSS, we calculate the average number of rounds it takes for the outcome to converge to an MDO. Since we have previously shown that the outcome will not deviate once an MDO is achieved in BSS, it is equal to check the average number of rounds it takes for the outcome to reach an MDO for the first time. For the strategy ESS, it does not always guarantee the convergence to an MDO. For those cases that the convergence property is lost, here we only check the average number of rounds it takes until an MDO is reached for the first time.

Here we only provide the results for ESS in different cases, shown in Table 4.6. For those cases that ESS guarantees the agents to converge to MDO (denoted in black), we can see that the average number of rounds required to converge to MDO is gradually increased when the number of agents n becomes larger. This implies that the convergence rate is gradually decreased with the increasing of the number of agents n. The intuitive reason is that the overall system becomes more dynamic due to the increase in the number of

Table 4.6: Average Number of Rounds to Converge to (Reach) an MDO in ESS

$\bar{R}$(n,k)	n = 3	n = 4	n=5	n=6	n=7	n=8	n=9	n=10
k=2	1.33	**2.44**	1.55	**2.69**	1.70	**2.87**	1.81	**3.00**
k=3	**2.63**	1.48	2.11	**3.20**	1.81	2.45	**3.52**	2.04
k=4	2.15	**3.08**	1.58	2.15	2.90	**3.73**	2.04	2.59

Table 4.7: The Number of States and Verification Time for Checking the Convergence Probability of ESS with and without Abstraction

	n=2	n=3	n=4	n=5	n=6	n=7	n=8	n=9	n=10
k=2 (state (W))	42	99	158	284	400	622	814	1159	1446
k=2 (state (W/O))	62	532	3482	35758	242602	-	-	-	-
k=2 (time (W))	0.015	0.017	0.027	0.027	0.061	0.043	0.168	0.064	0.276
k=2 (time (W/O))	0.063	0.073	0.631	28.77	126.7	-	-	-	-
k=3 (state (W))	99	349	1155	2413	3598	8009	13323	17614	32105
k=3 (state (W/O))	352	1985	38729	-	-	-	-	-	-
k=3 (time (W))	0.021	0.035	0.065	0.117	0.421	0.457	0.556	5.376	1.492
k=3 (time (W/O))	0.093	0.196	35.51	-	-	-	-	-	-
k=4 (state (W))	512	1885	3588	14478	34668	72848	103108	257861	462256
k=4 (state (W/O))	386	6662	148898	-	-	-	-	-	-
k=4 (time (W))	0.049	0.069	0.266	1.171	1.938	3.683	37.09	20.77	27.951
k=4 (time (W/O))	0.127	0.709	32.58	-	-	-	-	-	-

stochastic agents, thus making it more difficult for the agents to coordinate their actions. Another interesting observation is that the average number of rounds before convergence are always locally maximal (i.e., larger than that for both cases of $n-1$ and $n+1$) for those cases when the condition $n\%k = 0$ is satisfied, i.e., when the convergence property holds.

By applying counter abstraction technique in our modeling process, both the state space and the verification time cost are greatly reduced. To show this, here we list the details of verification time cost and state space cost of checking the convergence probability of ESS strategy in Table 4.7. "W" indicates counter abstraction is used while "W/O" means the opposite. Meanwhile, "-" means that the verification takes more than 10 minutes. We can see that our counter abstraction approach can significantly reduce the state space and the verification time, and more than half cases are not able to verify within a reasonable amount of time without abstraction.

4.6 Related Work

Ballarini et al. [19] apply probabilistic model checking to automatically analyse the uncertainty existing in a two-agent negotiation game. In the negotiation game, there exist one seller and one buyer bargaining over a single item, and both players exhibit probabilistic behaviors based on the opponent's previous behavior. They model the dynamics of the two-player system as a discrete-time Markov chain (DTMC). They mainly illustrate how to use the probabilistic model checker PRISM [68] to automatically analyse the probability that the players reach an agreement within each round of the game. This property is specified in probabilistic computation tree logic (PCTL) [59]. Their work is similar to our DG's analysis in that both work apply the probabilistic model checking technique to automatically analyse the dynamics of a multi-agent system in a game-like scenario. However, we study more complex scenarios involving an arbitrary number of players, actions and strategies, and we propose using the abstraction technique to reduce the model's state space.

Tadjouddine et al. [121] investigate the problem of automatically verifying game-theoretical property of strategy-proofness for auction protocols. They consider the case of Vickrey auction protocol and check the property of strategy-proofness using the model checker SPIN [54]. To solve the state space explosion problem, they apply two types of abstraction approaches to solve it, i.e., program slicing technique and abstract interpretation. Program slicing is a technique to remove portion of codes in the model which is irrelevant with respect to the property checked. The basic idea behind abstract interpretation is to map the original strategy domain onto an abstract and less complex domain, and then perform model checking on the abstract model. By using these two abstraction methods, the authors show that strategy-proofness of Vickrey auction can be automatically verified in SPIN for any number of players. However, in their work, there does not involve any probabilistic element within the protocol and the agents' behaviors, while dispersion games we consider exhibits highly stochastic behaviors.

Bordini et al. [25] review the problem of verifying multi-agent system implemented in language AgentSpeak using model checking techniques. They aim at automatically verifying whether certain specifications are satisfied using existing model checkers. For this purpose, the original multi-agent system implemented in a BDI language AgentSpeak [102] need to be transformed into the formal language supported by current model checkers first. They introduce a variant of language AgentSpeak, AgentSpeak(F), which can be automatically transformed into Promela, the model specification language of SPIN [54]. They also describe another approach based on the translation of the system in AgentSpeak into a system in Java, which then can be checked by another model checker JPF [138]. Additionally, they adopt a simplified form of BDI logic to specify the properties to be checked, which can be transformed into LTL, supported by previous model checkers. With the combination of these two techniques, the properties of a multi-agent system implemented in AgentSpeak can be automatically checked with existing model checkers. There also exists other similar work [135] that transforms other agent-based languages such as Mable [135] into Promela and use SPIN to perform model checking. However, in our work, the model is implemented directly in the modeling language supported by the model checker PAT, which avoids the additional language transformation cost. Besides, probabilistic property checking, which is important in analysing multi-agent system dynamics, is not supported in their work.

4.7 Summary

In this chapter, we proposed to automatically analyze the dynamics of MAS using model checking approach. Two representing scenarios are studied: *robustness* of negotiation strategies in a general multi-agent system and *basic simple strategy* and *extended simple strategy* in dispersion games. This approach guarantees the automaton, efficiency and completeness of the analysis procedure. In order to reduce the state space, counter abstraction technique is used in the modeling.

In robustness analysis, EGT analysis is used to check the strategy's robustness, and properties representing *empirical pure strategy Nash equilibrium*, *best reply cycle* and the corresponding *basin of attraction* are formally verified by PAT. Through the experimental results, we can see that our negotiation strategy *CUHKAgent* is very robust compared with other top strategies, and the overall robustness rankings of the strategies are given.

In BSS and ESS analysis, properties representing the system's convergence, deviation and convergence rate are studied. Experiments indicate that while BSS always guarantees convergence, ESS may have deviation. Using probabilistic model checking, better insights of the dynamics of these strategies were obtained compared with previous empirical evaluations.

Chapter 5

Improved Reachability Analysis in DTMC via Divide and Conquer

5.1 Introduction

Discrete Time Markov Chain (DTMC) is widely used in probabilistic model checking, such as the models of DGs in Chapter 4. The difference between DTMC and traditional Labeled Transition System (LTS) is that non-determinism in LTS is replaced by probabilistic choices in DTMC. Reachability analysis plays a key role in DTMC verification. Verification of properties such as Probabilistic Computational Tree Logic (PCTL) and Linear Temporal Logic (LTL) can be reduced to the reachability analysis problem [18]. Therefore in this chapter we focus on improving reachability analysis in DTMC verification.

Similar to PA, the transition probability matrix of a DTMC can be built from its transition relation. After the target states are decided, each state in the matrix can be represented by a variable, which means the probability of reaching the target states from this state. Next, there are mainly two approaches to calculate the probability from initial states to the targets. One is solving linear equations directly. In this method, variables representing intermediate states (which are not target or initial) are eliminated gradually through equations operation, and finally variables representing the initial states' probability of reaching targets can be solved. The other approach is using value iteration method, which works by finding a better approximation iteratively until certain stopping criteria are satisfied. The approach based on solving linear equations is straightforward to understand and it guarantees to deliver accurate result. However, since we need one variable for each state in the system, a lot of variables are needed for large systems whereas state-of-the-art linear solvers are limited to thousands of variables only. Therefore the applicability of this approach is limited to small-scale systems. On the other hand, the value iteration method tries to find fix-points iteratively, and it has relatively better performance in handling systems with a large number of states. Therefore it is more popular in probabilistic model checkers such as PRISM [80] and MRMC [73, 74]. However, this approach also has its drawback: slow convergence, i.e., it may take a large number of iterations before the approximations converge to a certain value. The phenomenon exists when there are complicated loops existing in the probabilistic systems, although the state space of such systems may not be very huge. The number of iterations is related to the subdominant eigenvalue of the probability transition matrix [141].

To tackle the above-mentioned problems, in this chapter we propose a new approach to verify DTMC models, especially for the ones with loops using a divide-and-conquer strategy. Instead of directly calculating the probability from initial states to targets, we divide the whole state space into several partitions, and solve them individually to eliminate loops. Afterwards, the remaining acyclic DTMC can be solved efficiently via value iteration method.

As we mentioned above, the slow convergence problem in value iteration comes from loops. Therefore, the first step of our approach is finding Strongly Connected Components (SCCs) in the given DTMC. This SCC-based approach is similar to previous work such as [12, 36, 5, 85]. However, instead of using SCC's topology order [36, 85], we solve each SCC independently by calculating the new transition probability from input states to output states of the SCC, which is similar to work [12, 5]. These new transitions are denoted as *abstract* transitions since SCCs are abstracted by transitions from input states to output states. However, [5] focuses on counterexample generation and abstracts SCCs via iteratively finding the smallest SCCs. On the contrary, we divide each SCC having a large number of states to several smaller partitions. For each partition, abstract transitions from its input to output are calculated via solving linear equations. Here we use Gauss-Jordan elimination [8]. Further, the states in each partition which are not input states will be removed, and thus the states in the SCC can be reduced. Afterwards, the new SCC is ready for next iteration of divide and conquer. This procedure for each SCC will be done iteratively until any of the following three criteria is satisfied. First, there is no more loop in the reduced SCC. Then this part will be left alone since it is already acyclic. Second, the number of remaining states in reduced SCC is small enough to be solved via a linear solver.

Third, the last iteration does not reduce any states. In the second and third scenarios, the final SCC will be solved via linear equation again, and final abstract transitions will be generated. After all loops in SCCs are resolved, the whole DTMC becomes acyclic, and value iteration is used to calculate the probability from initial states to targets. Since the abstract transitions from each partition's input states to output states are determined by the partition itself and independent to other partitions, multi-cores or distributed computers can be straightforwardly used here to solve each partition simultaneously, which makes the verification faster.

Contributions Compared with previous work, our contribution is threefold, as we summarize below.

1. A new divide-and-conquer approach for DTMC reachability analysis is proposed, which combines solving linear equations and value iteration methods together and tackles the problem that huge loops make the DTMC verification inefficient.
2. Based on the fact that each SCC and even each group in one SCC is independent from others, we use parallel computation to further speed up the verification.
3. The new approach has been implemented into our model checking framework PAT, and several representative experiments are conducted to show the effectiveness of our approach.

Organization The remainder of this chapter is structured as follows. Section 5.2 recalls relative background. In Section 5.3, we introduce our algorithm in details. The evaluation is reported in Section 5.4. Section 5.5 surveys related work and summarizes this chapter.

5.2 Preliminaries

In this section, we recall some background knowledge, which is relevant in the rest of this paper.

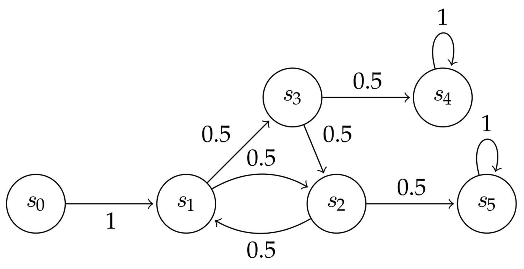


Figure 5.1: An Example of SCC

5.2.1 Discrete Time Markov Chains

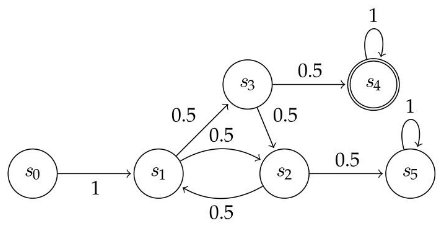
In Chapter 2, we have defined SCCs of a DTMC, and a DTMC is *acyclic* iff it only has *trivial* SCCs. Here we define an *adjacent group* (AG) $D \subseteq S$ such that $\exists\, s \in D, \forall\, s' \in D \wedge s' \neq s$, there is a finite path $\pi = \langle s_0, s_1, \cdots, s_n\rangle$ satisfying $s_0 = s \wedge s_n = s' \wedge \forall\, i \in [0, n], s_i \in D$, and s is called *root* state in D. In the following, we refer to adjacent groups simply as groups. The difference between SCC and AG is illustrated by the example in Fig. 5.1. Note the actions in DTMCs are not critical for the content of this chapter, so we ignore them for simplicity.

In Fig. 5.1, $\{s_1, s_2\}, \{s_1, s_2, s_3\}$ are *connected*; $\{s_0\}$, $\{s_4\}$, $\{s_5\}$ and $\{s_1, s_2, s_3\}$ are the *SCCs* in the model; AGs are more complex, for example, $\{s_0, s_1, s_2\}$ and $\{s_1, s_2, s_5\}$ are AGs and there are other possible combinations. Note that a set of states like $\{s_0, s_1, s_4\}$ is not a valid AG because there is no root state. *Connected* subgraphs are AGs but the reverse is not always true, e.g., $\{s_0, s_1, s_2\}$ is an AG but not a *connected* subgraph.

Similar to [12, 5], in a DTMC $\mathcal{M} = (S, s_{init}, Act, Pr, AP, L)$, given a group of states $\mathcal{D} \subseteq S$, the input states of $\mathcal{D}$ are defined as the states in $\mathcal{D}$ having incoming transitions from states outside $\mathcal{D}$; the output states of $\mathcal{D}$ are defined as states outside $\mathcal{D}$ which have incoming transitions from states in $\mathcal{D}$. Formal definitions are as follows.

$$Inp(\mathcal{D}) = \{s' \in \mathcal{D} \mid \exists\, s \in S\backslash\mathcal{D}.\ P(s, s') > 0\}^1$$
$$Out(\mathcal{D}) = \{s' \in S\backslash\mathcal{D} \mid \exists\, s \in \mathcal{D}.\ P(s, s') > 0\}$$

[1]If $s_{init} \in \mathcal{D}$, then $s_{init} \in Inp(\mathcal{D})$.



$$\begin{cases} p_0 = p_1 \\ p_1 = 0.5 \times p_2 + 0.5 \times p_3 \\ p_2 = 0.5 \times p_1 + 0.5 \times p_5 \\ p_3 = 0.5 \times p_2 + 0.5 \times p_4 \\ p_4 = 1 \\ p_5 = 0 \end{cases}$$

Figure 5.2: Reachability Analysis

5.2.2 Reachability Analysis in DTMC

One critical question for quantitative analysis of DMTC models is to compute the probability of reaching a certain set of target states G from the initial state. Here $\Diamond G$ is used to denote the event of reaching G, and $\mathcal{P}_{\mathcal{M}}(s_{init} \models \Diamond G)$ represents the probability that G can be reached from initial state in a DTMC $\mathcal{M}$. Here $\mathcal{P}_{\mathcal{M}}$ can be written as $\mathcal{P}$ if $\mathcal{M}$ is clear. Let $\pi = \langle s_0, s_1, \cdots, s_n \rangle$ represent any finite path in $\mathcal{M}$. Then we have

$$\mathcal{P}(s_{init} \models \Diamond G) = \mathcal{P}(\{\pi \mid s_0 = s_{init} \wedge \exists\, i \in [0..n], s_i \in G \wedge \forall\, j \in [0..i-1], s_j \notin G\})$$

Given the transition relation Tr of $\mathcal{M}$, there are two approaches to calculate $\mathcal{P}(s_{init} \models \Diamond G)$. One is solving linear equations, while the other is using value iteration. We use p_i to represent the probability from state s_i to the targets. In the following we use the example in Fig. 5.2 to show how these two approaches work. Note that state s_4 is the only target state, denoted by double cycles.

Solving Linear Equations From the model, the transition matrix between states can be built. For example, $p_1 = 0.5 \times p_2 + 0.5 \times p_3$ and $p_0 = p_1$. Since s_4 is target, $p_4 = 1$. s_5 cannot reach target obviously, therefore $p_5 = 0$. From these equations, each p_i can be solved through matrix operations. Although this approach can achieve accurate results, there are drawbacks. Because each state is represented by a variable, there may be a huge number of variables in large scale systems. The state-of-the-art linear solvers are limited to handle thousands of variables, therefore linear equation approach may not be scalable.

Using Value Iterations In this approach, p_i is calculated iteratively. This method is similar to the value iteration in PA introduced in Chapter 2; the only difference is that for each state in DTMC, there is one unique equation representing its transition relation. Therefore the maximal/minimal comparison is ignored. This approach has better scalability than the linear equations method, so it is more popular in existing model checkers. However, the existence of loops may make the convergence slow. The probability of each state in SCCs will be updated many times, which means a large number of iterations may be needed before the results satisfy the terminating criteria.

5.2.3 States Abstraction and Gauss-Jordan Elimination

Here we follow the idea of [5]. Given a DTMC $\mathcal{M} = (S, s_{init}, Tr, AP, L)$ and a group of states $\mathcal{D} \subseteq S$, $\mathcal{D}$ can be abstracted by calculating the transition probability from $Inp(\mathcal{D})$ to $Out(\mathcal{D})$. According to the proof in [5], the abstraction of any arbitrary set of states is independent from others, and the abstract transitions **do not affect** the probability of reaching target states G.

One example of the abstraction is in Fig. 6.4. Fig. 6.4 (a) is the original DTMC, which has one SCC $\mathcal{D} = \{s_1, s_2, s_3\}$. $Inp(\mathcal{D}) = \{s_1\}$ and $Out(\mathcal{D}) = \{s_4, s_5\}$. In order to abstract $\mathcal{D}$[2], the probability from $Inp(\mathcal{D})$ to each state $s_{out} \in Out(\mathcal{D})$ should be calculated. Theoretically, the calculation from an SCC's inputs to outputs can be solved via linear equations or value iteration approaches[3]. However, for value iteration approach, since there could be several output states in $Out(\mathcal{D})$, we have to separately calculate the probability from input states to each output state. If there are many output states, this method could be inefficient. In addition, the existence of loops still causes slow convergence issue. Furthermore, using value iteration, there will be some errors because of the user-defined precision, but there is no way to know the error bounds. Therefore, we use a specific linear equation solving technique: Gauss-Jordan elimination [8] to do the abstraction.

Gauss-Jordan elimination is an algorithm for getting matrices in reduced row echelon form that placing zeros above and below each pivot [8]. Here, we briefly introduce how it works in our setting.

Assume there are m states in a set of states, say $\mathcal{D}$, and $|Out(\mathcal{D})| = n$. Then two matrices A and B, containing linear equations information of all transitions in $\mathcal{D}$, are first introduced

[2]Here we take an SCC as an example. Actually this abstraction can be applied to arbitrary set of states, according to [5].

[3]Different from our previous discussion which focuses the calculation from the initial state to targets, here we discuss the probability from input states to every output state of an SCC.

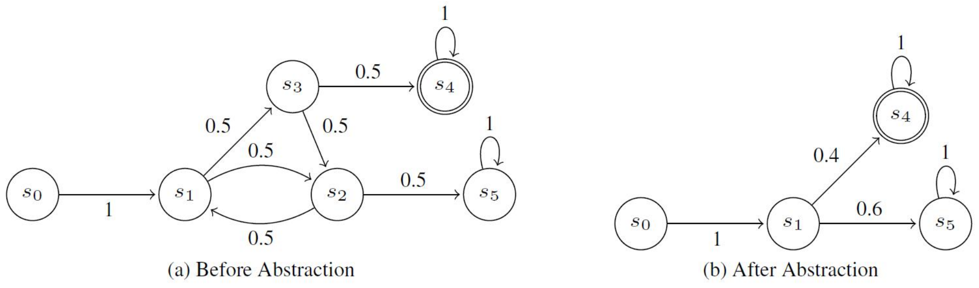


Figure 5.3: States Abstraction via Gauss-Jordan Elimination

as follows.

$$A(i,j) = \begin{cases} 1, & \text{if } i = j; \\ -Tr(i,j), & \text{otherwise.} \end{cases} \qquad B(i,k) = -Tr(i,k).$$

Here, A is an $m \times m$ square matrix. $A(i,j)$ is a negative value of probability of transition from i^{th} state to j^{th} state in $\mathcal{D}$ if $i \neq j$. The diagonal elements of A are filled by 1. This records the transition relationship within $\mathcal{D}$. B is an $m \times n$ matrix to record the transition relationship from $\mathcal{D}$ to $Out(\mathcal{D})$. k represents the k^{th} state in $Out(\mathcal{D})$.

Next, augmenting the square matrix A with matrix B, we will have $[A \mid B]$. Gauss-Jordan elimination on $[A \mid B]$ will then produces $[I \mid C]$. Here, I is the identity matrix with 1s on the main diagonal and 0s elsewhere. The new transition probability e.g., $Tr'(i,k)$, stores the transition probability from i^{th} state in $\mathcal{D}$ and k^{th} state in $Out(\mathcal{D})$, which is actually $-C(i,k)$. Now take Fig. 6.4 (a) as an example. Its $[A \mid B]$ and resulting $[I \mid C]$ are listed as follows. In this example, $A(i,j)$ corresponds to $Tr(s_{i+1}, s_{j+1})$ and $B(i,k)$ indicates $Tr(s_{i+1}, s_{k+4})$.

$$[A \mid B] = \left[\begin{array}{ccc|cc} 1 & -0.5 & -0,5 & 0 & 0 \\ 0 & 1 & -0.5 & 0 & -0.5 \\ 0 & -0.5 & 1 & -0.5 & 0 \end{array}\right]; \quad [I \mid C] = \left[\begin{array}{ccc|cc} 1 & 0 & 0 & -0.4 & -0.6 \\ 0 & 1 & 0 & -0.2 & -0.8 \\ 0 & 0 & 1 & -0.6 & -0.4 \end{array}\right]$$

Here the transitions from all the states in $\mathcal{D}$ to $Out(\mathcal{D})$ are obtained. Note that those states which are not in $Inp(\mathcal{D})$ will be removed. Therefore we are just interested in the new transitions from $Inp(\mathcal{D})$ to $Out(\mathcal{D})$, which are

$Tr'(s_1, s_4) = 0.4; \quad Tr'(s_1, s_5) = 0.6;$

We can obtain that $p_1 = 0.4 \times p_4 + 0.6 \times p_5$ in the abstracted DTMC, which is shown in Fig. 6.4 (b). Given a group of states $\mathcal{D}$, this abstraction procedure is defined as a method $Abs(\mathcal{D})$.

Note that in practice, most transition matrices in probabilistic model checking have a very sparse structure that contains a large number of zeros. We adopt a compressed-row representation [111] as a data structure for matrices in Gauss-Jordan elimination.

5.3 Divide and Conquer Approach

From the analysis in Section 5.2, for a large DTMC with complicated loop structure, both linear equations and value iteration method are ineffective, even unworkable. In this section, we propose a divide and conquer approach which tackles the above-mentioned problem. Our main idea is similar to work [12, 5], which transfers the original DTMC to an acyclic one by abstracting SCCs recursively so as to reduce the number of state and loops.

Intuitively, our approach divides large SCCs into smaller partitions, each of which will be solved via Gauss-Jordan elimination independently. Through this approach, loops will be eliminated. Afterwards, value iteration method is used to decide the final probability of reaching targets. In the following, we introduce our algorithm in details.

5.3.1 Overall Algorithm

Given a DTMC $\mathcal{M}$ $(S, s_{init}, Act, Pr, AP, L)$ and target states $G \subseteq S$, the probability of reaching G, denoted as $\mathcal{P}(s_{init} \models \Diamond G)$, can be solved by Algorithm 4. Note that B is an input parameter, which indicates SCCs having more than B states should be divided. $Abs(K)$ is defined in Section 5.2.3. $VI(\mathcal{M}, G)$ indicates calculating the probability of reaching G via value iteration. The procedure of the algorithm is explained in the following.

- The first step is to find all SCCs C in $\mathcal{M}$ by Tarjan's approach [122], and their input and output states are recorded as well. This is captured by Line 1.
- For each SCC $\mathcal{D} \in C$, we will first check whether $|\mathcal{D}|$ exceeds B or whether $|Out(\mathcal{D})| > 1$. If not, $Abs(\mathcal{D})$ will be executed directly. States in $\mathcal{D}$ but not in $Inp(\mathcal{D})$ will be removed. Afterwards $\mathcal{D}$ will be removed from C, as shown in Lines 4-5. The reason why we directly abstract cases $|Out(\mathcal{D})| \leq 1$ is as follows.

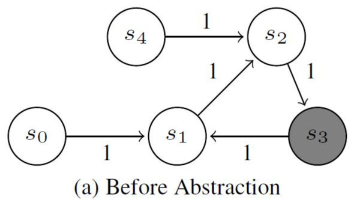
Algorithm 3 Divide and Conquer Approach

```
1: Let $C$ be the set of all nontrivial SCCs in $\mathcal{M}$;
2: while $|C| > 0$ do
3:     Let $\mathcal{D} \in C$;
4:     if $|\mathcal{D} \leq B| \vee Out(\mathcal{D}) \leq 1$ then
5:         $Abs(\mathcal{D})$ and $C \leftarrow C \backslash \mathcal{D}$;
6:     else
7:         Divide $\mathcal{D}$ into a set of AGs denoted as $\mathcal{A}$;
8:         for each $\mathcal{E} \in \mathcal{A}$ do
9:             $Abs(\mathcal{E})$;
10:        end for
11:        Let $\mathcal{D}'$ be the set of remaining states in $\mathcal{D}$;
12:        if $|\mathcal{D}'| \leq B \vee |\mathcal{D}'| = |\mathcal{D}|$ then
13:            $Abs(\mathcal{D}')$ and $C \leftarrow C \backslash \mathcal{D}$
14:        else
15:            Let $C_{\mathcal{D}'}$ be the set of all nontrivial SCCs in $\mathcal{D}'$;
16:            $C \leftarrow (C \backslash \mathcal{D}) \cup C_{\mathcal{D}'}$;
17:        end if
18:    end if
19: end while
20: return $VI(\mathcal{M}, G)$;
```

  - If $|Out(\mathcal{D})| = 0$, $\mathcal{D}$ has no outgoing transitions, then no matter whether $\mathcal{D}$ has target states or not, we do not need to solve $\mathcal{D}$. If $\mathcal{D} \cap G = \phi$, it is obvious that all states in $\mathcal{D}$ has probability 0 to reach G; otherwise, it is trivial to show that all states in $\mathcal{D}$ has probability 1 to reach G.
  - If $|Out(\mathcal{D})| = 1$, assume s_{out} is the output state. All paths entering $\mathcal{D}$ will leave it eventually. Therefore, for every $s_i \in Inp(\mathcal{D})$, the probability of paths entering $\mathcal{D}$ via s_i, staying in $\mathcal{D}$ and exiting $\mathcal{D}$ to s_{out} should be 1. So $\mathcal{D}$ can be abstracted directly.

- Lines 7-17 describe the case when $\mathcal{D}$ needs to be divided, i.e., when the SCC has more than B states. First we divide $\mathcal{D}$ into several groups based on some heuristics, each of which has a reasonably small number of state, i.e., less than B. Therefore, for each group $\mathcal{E}$ we use $Abs(\mathcal{E})$ to get the abstraction. Here we choose AG as the structure of each partition, because the existence of the root state, say s_r, may remove the most states after abstraction. In the extreme case where $Inp(\mathcal{E}) = \{s_r\}$, all states in $\mathcal{E}$ except s_r can be removed.

- By removing the states which are not input states of any $\mathcal{E}$, the number of states in



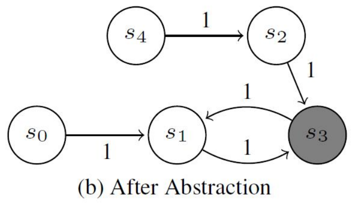


Figure 5.4: Destruction of SCC during Abstraction

$\mathcal{D}$ is often (not always) reduced. Line 12 checks two situations. 1) the size of $\mathcal{D}'$ is smaller than or equal to B, and 2) there is no reduction for $\mathcal{D}$ in this iteration. If 1) is true, then there is no need to divide $\mathcal{D}'$ again, and $Abs(\mathcal{D}')$ is executed directly. If 2) is true, i.e., no state is reduced after divide and conquer, the main reason should be that each state in $\mathcal{D}$ has a lot of pre-states. Therefore every state in one group is an input state and cannot be removed. In this case, $\mathcal{D}'$ should also be abstracted. Afterwards, $\mathcal{D}$ is removed from C. If 1) and 2) are both false, Lines 15-16 will be executed.

- Because of the abstraction, $\mathcal{D}$ may not be an SCC now. An example is shown in Fig. 5.4. On the left hand side, $\mathcal{D} = \{s_1, s_2, s_3\}$; if we group s_1 and s_2 together, then s_3 is this group's output. It is easy to get the abstract transitions between them, as shown in right hand side. Because both s_1 and s_2 are input states, no state is removed. However, it is obvious that $\mathcal{D}' = \{s_1, s_2, s_3\}$ is not an SCC anymore. Tarjan's algorithm is used again to find new SCCs in the $\mathcal{D}'$, captured by Line 15. New SCCs will be added to C for another iteration.

- When the iteration terminates, there is only trivial SCCs in $\mathcal{M}$ now; in other words, $\mathcal{M}$ is acyclic. Value iteration approach can be used to calculate the probability from the initial state to targets efficiently, and this is captured by Line 20.

As we mentioned in Section 5.2.3, the iterative abstraction will not affect the final result of the probability calculation. The following theorem establishes that the algorithm is always terminating.

Theorem 5.3.1 *Given a finite state DTMC $\mathcal{M}$, Algorithm 4 always terminates.*

Proof We assume $\hat{S} = \Sigma_{\mathcal{D}\in C} |\mathcal{D}|$, in other words, $\hat{S}$ is the total number of states in C. Then the theorem can be proved by showing (1) $\hat{S}$ is finite at the beginning, and (2) $\hat{S}$

monotonically decreases after each iteration.

(1) is obviously true because $\mathcal{M}$ has finite number of states, and $\hat{S} \leq |S|$ where S is the set of states of $\mathcal{M}$.

Given an SCC $\mathcal{D} \in C$, if it satisfies the condition in Line 4, then $\mathcal{D}$ will be removed from C, thus $\hat{S}$ is reduced. Otherwise, from Line 6, there are two possible outputs. (i) $\exists \mathcal{E} \in \mathcal{A}$, $Abs(\mathcal{E})$ reduces its number of states, or (ii) $\forall \mathcal{E} \in \mathcal{A}$, $Abs(\mathcal{E})$ does not reduce its number of states. If (i) is true, then $\hat{S}$ is also reduced. If (ii) is true, then $|\mathcal{D}'| = |\mathcal{D}|$. According to Line 8, $\mathcal{D}$ will be abstracted directly and be removed from C. Thus $\hat{S}$ is still reduced. Therefore (2) is true, and the theorem holds. □

5.3.2 Dividing Strategies

Although the divide-and-conquer approach is correct and terminating, its efficiency is highly dependent on how an SCC is divided. Assume $\mathcal{A}$ is the set of partitions after dividing an SCC, then a suitable partition, say $\mathcal{E} \in \mathcal{A}$, should satisfy the following conditions.

1. $\mathcal{E}$ should not have too many states, since each partition is abstracted using Gauss-Jordan elimination which is limited to a relatively small number of states;
2. $\mathcal{E}$ should not have too few states as well, otherwise there will be too many partitions to be solved, and the states reduction for $\mathcal{E}$ is inefficient;
3. The smaller $|Out(\mathcal{E})|$ is, the better reduction is achieved. Too many output states will make the input states of $\mathcal{E}$ have too many abstract transitions, which makes the remaining structure complicated, and affects the efficiency of the following abstraction.

As a result, the remaining issue is that given an SCC $\mathcal{D}$, is there any *optimal* strategy to divide it into *suitable* AGs? In practice, the structure of $\mathcal{D}$ could be arbitrary. This increases the difficulty of finding a general strategy for all cases.

The simplest division method is to try to set each AG to have the same number of states. Assume each AG should have N states. Then starting from one input state of $\mathcal{D}$, depth first search (DFS) or breadth first search (BFS) can be used to group every N states together. Afterwards, each AG can be abstracted, and the remaining states are combined together to do the next iteration. The advantage of this strategy is that the number of states in each partition is easily controlled. It can be very efficient in cases where the states in $\mathcal{D}$ has few transitions. However, this method cannot control the number of output states of each partition, and a predefined N may not be suitable for $\mathcal{D}$'s structure.

Therefore, another improved strategy is used to automatically decide the number of states in each AG. Instead of picking a constant N in the beginning, we set a lower bound B_L and an upper bound B_U for each partition. Thus the number of states in each partition should be between B_L and B_U. At first, B_L states will be grouped into $\mathcal{E}$, and $|Out(\mathcal{E})|$ is recorded. Afterwards, some states in $Out(\mathcal{E})$ are added into $\mathcal{E}$, and $|Out(\mathcal{E})|$ is updated. If $|Out(\mathcal{E})|$ keeps unchanged or even becomes smaller after the update, we will try to add more states into $\mathcal{E}$ again. If $|Out(\mathcal{E})|$ is increased but the increase is not significant, a few states will be added into $\mathcal{E}$ but the number should be small. Otherwise $\mathcal{E}$ is confirmed and ready for $Abs(\mathcal{E})$. Note the number of states in $\mathcal{E}$ should be always below B_U. This strategy guarantees

1. the number of states in $\mathcal{E}$ is under control. B_L and B_U guarantee that the size of $\mathcal{E}$ should not be too large or too small.
2. the outputs of $\mathcal{E}$ are also manageable. This guarantees the states structure after abstraction is not too complicated, and is suitable for next iteration.

Parameters B, N, B_L and B_U can be adjusted according to the specific DTMC to get the optimal efficiency.

5.3.3 Parallel Computation

Previous work such as [36, 85] depends on the topological order between different SCCs. Therefore, parallel computation is not so easy to use in their setting. On the contrary, our algorithm eliminates loops via abstracting every SCC one by one, without considering their order. The independence between different SCCs can be proved following the proof in [5]. What is more, even each AG in one SCC is also independent from others, and the proof actually follows the same idea of SCC's independence. Thus, parallelization is suitable in our setting in order to solve different AGs simultaneously.

In details, after finding all SCCs, they are stored with their input and output states. For each SCC, a spare thread can be used to solve it. Therefore, Lines 2-14 in Algorithm 4 can be

System	PAT (w)			PAT (w/o)		
	Prob	Time (s)	Memory (MB)	Prob	Time (s)	Memory (MB)
N = 500	0.5	0.03	71	0.49987	0.5	24
N = 5000	0.5	0.3	83	0.49987	5.5	63
N = 50000	0.5	2.6	151	0.49987	125.2	111
N = 500000	0.5	29.7	885	0.49987	1612.8	838

Table 5.1: Experiments: A Simple Example

solved via parallel computation. In addition, whenever an AG is grouped, another spare thread, if there is any, can be used to abstract it. Thus Line 8 in Algorithm 4 can also be handled in parallel.

5.4 Implementation and Evaluation

We have implemented the algorithm into PAT. In the following, several experiments are conducted to show the efficiency of our new approach. Note that we show the improvement via comparing to PAT itself, which was based on value iteration method previously. Since the only difference between these two versions is the algorithm of reachability analysis, it is fair to check the effectiveness of the new method. Besides, several cases used in our experiment have dynamically updated probabilistic distributions, and the modeling of them by other model checkers is highly nontrivial.

In these experiments, we use the **improved** dividing strategy, and B, B_L, B_U are set to be 300, 100, 150 respectively. In other words, an SCC with more than 300 states should be divided; each group has states between 100 and 150. These parameters are manually selected based on our experimental experience, i.e., generally these parameters have better performance compared with others. The testbed is a server running Windows Server 2008 64 Bit with Intel Xeon 4-Core CPU×2 and 32 GB memory.

First, we use a simple example to show that our approach gets accurate results, resolves the slow convergence problem and results in huge speedup. Assume there are $N + 2$ states $\{s_0, s_1, ..., s_{N-1}, s_u, s_f\}$ existing in this example. Each state $s_i, i \in [0..N - 1]$, has probability 0.99 to reach $s_{(i+1)\%n}$, and also has probability 0.005 to reach s_u and s_f separately. The case $N = 3$ is shown in Fig. 5.5. Obviously, all states $s_i, i \in [0..N - 1]$ compose an SCC, and s_u and s_f are this SCC's outputs. We check the probability from s_0 to s_u, and several experiments are executed based on different value of N as listed in Table 5.1.

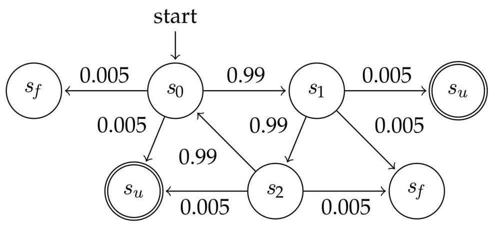


Figure 5.5: A Simple Example: N = 3. s_u and s_f are copied for better demonstration.

In Table 5.1, columns $Prob$ represents the probability returned by the model checking algorithms. Columns $PAT(w)$ ($PAT(w/o)$) show the experimental information taken with (without) the new approach. Columns $Time$ represent the total time cost in the verification. For these cases, our new approach outperforms value iteration approach dramatically by reducing the verification time to less than 10%. On the other hand, the memory used in new approach is higher than that used in the previous method, which is reasonable since solving linear equations consumes more memory than value iteration approach. Through the manual analysis, we know that 0.5 is the accurate result while 0.4998 is only an approximation.

Next, we apply our approach to several more meaningful systems and demonstrate that our approach can still improve the efficiency significantly.

Dispersion games [58] are used again for the experiments since the models of both BSS and ESS are DTMCs. Their convergence probabilities are calculated respectively. Another case used in our experiments is coin flipping protocol for polynomial randomized consensus [13] (CS). This case focuses on modeling and verifying the shared coin protocol of the randomized consensus algorithm. Here we use a safety property in the system as our target.

The experiments based on these three models are listed in Table 5.2. $BSS(N)$ indicates there are N players (also N actions) in the game; $ESS(N, K)$ means there are N players and K actions; $CS(N, K)$ indicates there are N processes and K is a constant used in the model. Here we are interested in the ratio of model building (*BM*) time to the total time, which is denoted as *BMR* in the table. In $PAT(w)$, *BM* means the time for building **acyclic** DTMC, i.e., the overall time consumed by eliminating loops in DTMC; in $PAT(w/o)$, it indicates the time for building the whole system. In both PAT versions, value iteration is used to get the final result after building the model. '-' indicates the verification takes more than 1 hour thus the result is not taken into consideration. From the table, we have several observations.

System	States	Prob	PAT (w)			PAT (w/o)		
			Time	BMR	Memory	Time	BMR	Memory
BSS (4)	4196	1	1.3	92.3%	39	0.2	50%	35
BSS (5)	49572	1	3.5	94.3%	297	4.4	11.4%	142
BSS (6)	605890	1	41.4	72.7%	1297	105.3	6.7%	417
BSS (7)	7462639	1	1671	30.1%	6350	2073.1	4.1%	5039
ESS (6, 4)	32662	1	1.4	92.8%	16.3	2.7	14.8%	5.6
ESS (6, 5)	162945	1	6.7	91.1%	48.5	11.4	16.7%	13.9
ESS (7, 5)	463460	1	27.9	84.9%	310	75.8	7.1%	292
ESS (8, 5)	1114480	1	70.5	74.7%	619	278.5	6.1%	643
ESS (8, 6)	6476524	1	438.0	68.5%	4209	1168.1	7.5%	3904
CS (4, 3)	4966	0.023	0.8	87.5%	45	2.4	8.3%	35
CS (6, 3)	34529	0.023	15.7	81.5%	214	124.1	0.9%	108
CS (6, 4)	45281	0.015	24.8	86.7%	324	243.8	0.6%	81
CS (6, 5)	56033	0.012	38.6	91.2%	312	432.1	0.4%	104
CS (7, 4)	99265	0.014	102.3	87.6%	1062	983.1	0.4%	97
CS (7, 5)	122785	0.011	161.7	92.1%	1145	1384.8	0.3%	97
CS (7, 6)	146305	0.01	245.5	94.9%	1404	2409.5	0.2%	156
CS (8, 4)	200083	0.013	585.1	93.4%	1974	-	-	-

Table 5.2: Experiments: Benchmark Systems

1. For some small examples such as $BSS(4)$, our new approach is slower. This is due to the overhead taken by the SCC searching algorithm, and value iteration approach is efficient when loops are small.

2. As the examples become larger, the verification speed is increased by our proposed approach. This improvement is obvious especially in large-scale systems such as $ESS(8,5)$, $ESS(8,6)$ and $CS(8,4)$.

3. CS consumes more resource than BSS and ESS when they have similar size of state space, such as $CS(7,6)$ and $ESS(6,5)$. The reason is that CS has more complicated SCCs, and both our new approach and traditional value iteration method have to use more time and memory to solve it. As a result, the SCCs' structure affects the verification efficiency to a large extent.

4. According to BMR, we can see that in the previous version of PAT, building the model costs small portion of the overall verification time compared with the value iteration procedure. The average value of BMR is less than 10%, which means slow convergence indeed exists in systems having large SCCs. CS has very small BMR and this is consistent with the fact that CS has complicated SCCs. In the new approach,

time is mainly used by abstractions, as average *BMR* is more than 80%. It indicates that the efficiency of the divide and conquer strategy is critical in the whole verification now, and optimal dividing strategy is worthy to explore.

On the other hand, we want to share some limitations of our approach according to the experimental information. The efficiency of this approach is dependent on whether large SCCs exist in the system. During our experiment, the new approach performs slower than value iteration method in several cases. The main two reasons include 1) there is no loops in the system, thus the SCC searching algorithm makes the whole verification slow; 2) the system just has small SCCs while the whole state space is large, thus the gain of the abstraction is limited.

5.5 Related Work and Summary

SCCs are an important structure in both concurrent and probabilistic verification. For probability calculation, those loops in SCCs are one of the key factors affecting the efficiency. Some previous work has been done based on SCC decomposition for probabilistic systems, including DTMCs and Markov Decision Processes (MDPs) [18], and we are mainly inspired by this work.

To speed up the verification of MDP, the authors of [36] have proposed to decide the topological order of all SCCs in the MDP, and value iteration method is used to solve the SCCs from the bottom upwards. Based on this work, the authors of [85] have used SCC decomposition to handle the incremental quantitative verification of MDP. The topological order between SCCs guarantees that some changes in one SCC will not affect those SCCs after it. Compared to their work, ours does not consider the orders of SCCs via treating each SCC independently. This makes parallel computation approach feasible. In addition, Gaussian-Jordan elimination is used to remove loops. Different from value iteration, which needs a user defined precision, our approach generates accurate result.

Besides, there are several work based on SCC focusing on probabilistic counter-example generation, such as [12, 5]. Their idea of abstracting each SCC from its input to output is the biggest inspiration of our work. Compared with these work, ours is more focusing on improving reachability analysis in DTMC. Therefore, we divide SCCs into smaller partitions and solve them directly.

Summary In this chapter, we proposed a divide-and-conquer approach to speed up reachability analysis of DTMCs. Because SCCs are one of main reasons that the probability calculation is slow, we focus on abstracting SCCs via calculating the transition probability from their inputs to outputs. We divide every SCC, whose states exceed some specific bound, into several AGs having reasonable number of states, and can be solved efficiently via Gauss-Jordan elimination. We have implemented our approach in PAT, and some benchmark systems are used to show its effectiveness and efficiency.

Chapter 6

Modeling and Verifying Probabilistic Real-Time Systems using PRTS

6.1 Introduction

PCSP# is expressive since it captures both concurrency and probability. However, real-life systems might have more complicated behaviors, such as quantitative timing requirements (e.g., time out). Two examples are shown below to demonstrate systems which cannot be handled via PCSP# due to their real-time characteristics.

Example 1 IPv4 Zeroconf protocol [34] is a dynamic configuration protocol for IPv4 addresses; whenever an individual device is connected to a network, the device is allowed to manage its own address configuration. First, it is required to randomly choose an IP address from a pool of 65024 available addresses. Afterwards, the device sends notifications to other devices in the network to check whether this chosen IP is occupied or not. This device will wait 20 seconds to obtain others' responses. If no reply is received after the devices resend such messages three more times, the device starts using the IP address. Otherwise, it repeats by choosing a new IP address and sending notifications. In the whole procedure, messages transferring in the system may get lost due to some errors. Thus, probabilistic behavior modeling is needed in this case to specify the IP picking behavior and the messages transmission; meanwhile, real-time constrains are needed to handle the time limits of the message communication. In this system, the aim of each device is trying to get an IP address, therefore one important issue is: *given a new device, what is its probability of getting an IP (might be used or fresh) with a specific deadline*?

Example 2 Multi-lift systems are widely used in high-level buildings. These systems are heavily relying on control software, which has various requirements. A multi-lift system consists of a hierarchy of components, e.g., the system contains multiple lifts, floors, users, etc.; a lift contains a panel of buttons, a door and a lift controller; a lift controller may contain multiple control units. It is complex in control logic as behavior of different components must be coordinated through a software controller. Further, there is a degree of randomness in the system. For example, whenever a user enters a lift and wants to go up, his target floor is probabilistically distributed. On the other hand, if a user wants to go down, then the probability of choosing ground floor is probably the biggest. What is more, the responding and serving time of the lifts should also be taken into consideration, therefore quantitative timing is also needed in multi-lift systems. In terms of correctness, one property from a user point of view may be: *if a user has requested to travel in certain direction, a lift should* not *pass by, i.e., traveling in the same direction without letting the user in*. However, designing multi-lift systems which guarantee the property is extremely challenging and thus lift systems in practice often "violates" this property. Typically, once a user presses a button on the external panel at certain floor, the controller assigns the request to the '*nearest*' lift. If the '*nearest*' lift is not the first reaching the floor in the same traveling direction, the property is violated. One counterexample that could be returned by a standard model checker is that the lift is held by some user for a long time so that other lifts pass by the floor in the same direction first. One potential remedy is to re-assign all external requests every time a lift travels to a different floor. Due to high complexity, many existing lift systems do not support re-assigning requests. The question is then: *what is the probability of violating the property, with typical randomized arrival of user requests from different floors or from the button panels inside the lifts, or even to quality how much better*? If the probability is sufficiently low, then the design may be considered as acceptable. Further, can we prove that choosing the 'nearest' lift is actually better than assigning an external request to a random lift?

Ideally, these systems shall be formally verified in order to avoid undesired failures in operation. However, model checking probabilistic real-time systems like above is nontrivial. In particular, there are two challenges. First, a modeling language which is more expressive than PCSP# to support features like real-time, hierarchy, concurrency, data structures as well as probability, is required. Second, the models must be efficiently model checkable for widely used properties, such as deadlock checking, reachability checking and Linear Temporal Logic (LTL) checking [37].

One line of work on modeling complicated systems is based on integrated formal specification languages [30, 91]. These proposals suffer from one limitation, i.e., there are few supporting tools for system simulation or verification. Existing model checkers are limited because they do not support one or many of the required system modeling features. For instance, SPIN [70] supports complex data operations and concurrency, but not real-time or probability. Uppaal [23] supports real-time, concurrency and recently data operations as well as statistical model checking (in the extension named [39]), but lacks support for hierarchical control flow. PRISM [80] supports the verification of Probabilistic Timed Automata (PTA) [82], which has the concurrency, probability and real-time characteristics. However, it does not support hierarchical systems, but rather networks of flat finite state systems. In addition, most of the tools support only simple data operations, which could be insufficient in modeling systems which have complicated structures and complex data operations.

In this chapter, we propose a new formal modeling language Probabilistic Real-time System (PRTS) which covers a range of features to model the above-mentioned systems. The following summarizes how PRTS supports different system features.

- Hierarchy: PRTS supports a rich set of compositional operators to compose system components in different ways, e.g., conditional or unconditional choice, interrupt, parallel composition, sequential composition, etc. Many are adopted from process algebras with amended operational semantics [69]. As a result, PRTS models are fully hierarchical as a system component can be composed by other components.
- Concurrency: PRTS supports parallel composition of system components. Components running in parallel can communicate through shared variables, synchronous/asynchronous channels or multi-party barrier synchronization.
- Real-time: PRTS supports a set of timed operators like *deadline, time out and timed interrupt* to capture real-time requirements. Therefore, real-time system behavior can be captured concisely and intuitively.
- Probability: PRTS supports probabilistic system modeling through probabilistic choices. One example is that a pacemaker model has a choice of functioning correctly with probability 99.54% or 0.46% probability of malfunctioning [20].

- Data structures and operations: PRTS also supports primitive data types like Boolean, Integer, array and arbitrary user-defined data types as PCSP# does. User-defined data types and associated data operations can be defined in imperative programming languages such as Java and C#. Models which depend on the data types must import the corresponding classes as an external library. The only constraint on these classes is that they must implement a pre-defined interface so that they can be manipulated by the model checking algorithms.

Due to quantitative timing, the state space of the systems might be infinite. Therefore, we develop an abstraction technique based on zone abstraction [44] so as to generate a finite-state model, which is subject to model checking. For system verification, two widely used properties are supported in PRTS: reachability checking and LTL checking.

Compared with previous work, we make the following technical contributions.

1. We define an expressive modeling language PRTS, which is a combination of data structures/operations, hierarchy, real-time, probability, concurrency, etc. Expressiveness often comes at the price of decidability. PRTS is carefully designed so that it is expressive and at the same time model checkable.

2. Our second contribution is a fully automatic zone abstraction method. We show that the infinite states caused by real-time transitions could be reduced by our abstraction to a finite set of zones. As a result, we obtain a finite-state abstract PA which is subject to probabilistic model checking.

3. We develop an algorithm to model check PRTS models against LTL properties with non-Zenoness assumption. Intuitively, the non-Zenoness assumption guarantees that infinitely many steps always take infinite time. It is necessary as Zeno traces (traces which take infinitely many steps with finite time units) are unrealistic and should not be regarded as counterexamples for the property.

4. We implement a dedicated model checker, which supports editing, visualized simulation and verifying PRTS models. The model checker employs state-of-the-art probabilistic model checking techniques for checking PRTS models against temporal logic properties or refinement relationship. The tool has been applied to the multi-lift system and benchmark systems. We compare it with existing model checkers to show its efficiency and scalability.

The remainder of the chapter is structured as follows. Section 6.2 recalls some background knowledge. Section 6.3 introduces PRTS in detail, including its syntax and operational semantics. Section 6.4 describes the zone-based abstraction technique. Section 6.5 describes model checking with the non-Zenoness assumption. The evaluation is reported in Section 6.6. Section 6.7 surveys related work. Section 6.8 summarizes this chapter.

6.2 Preliminaries

In this section, we recall basic concepts and definitions relevant to model checking techniques [37, 18].

6.2.1 Probabilistic Formalisms for Real-time Systems

In Chapter 2, we have introduced probabilistic formalisms such as PA and DTMC. In this part, we add timed transitions to these formalisms so that they can capture real-time behaviors. Besides the visible actions Act and invisible action τ, $\epsilon \in \mathbb{R}_+$ denotes the event of idling for exactly ϵ time units and $\mathbb{R}_+$ denotes the set of positive real numbers. Transition $s \overset{x,\mu}{\to} s'$ in PA and transition $s \overset{x,p}{\to} s'$ now satisfy $x \in Act_\tau \cup \mathbb{R}_+$. Other definitions are consistent with Chapter 2.

6.2.2 LTL-X

SE-LTL is also introduced in Chapter 2. However, in real time systems, it is difficult to define "next". Therefore in this work we focus on LTL-X property, i.e., SE-LTL without X operator.

6.2.3 Non-Zenoness

In reality, infinite steps must take infinite time. This requirement is known as non-Zenoness, which is formally defined as follows.

Definition 10 *Given a path $\pi = \langle s_0, x_0, \mu_1\, s_1, x_1, \mu_2 \cdots \rangle$ in a PA, π is non-Zeno if and only if π is infinite, and $\Sigma_{x_i \in R_+} x_i$ is unbounded.*

$$\begin{array}{lll}
P ::= & Wait[d] & \text{– time delay} \\
\mid & P\ timeout[d]\ Q & \text{– time out} \\
\mid & P\ interrupt[d]\ Q & \text{– timed interrupt} \\
\mid & P\ within[d] & \text{– timed responsiveness} \\
\mid & P\ deadline[d] & \text{– deadline} \\
\mid & pcase\ \{pr_0 : P_0;\ pr_1 : P_1;\ \cdots;\ pr_k : P_k\} & \text{– probabilistic multi-choices}
\end{array}$$

Figure 6.1: Process constructs

A path is Zeno if and only if it is not non-Zeno. In other words, a path is Zeno if and only if it contains infinitely many steps taken in a finite time interval. For obvious reasons, Zeno paths are unrealistic, and should be ruled out during verification.

In probabilistic systems, an infinite run may have probability 0 and therefore it is irrelevant whether it is Zeno or not. Thus, we follow the idea of [82] and focus on non-Zeno schedulers, as defined below.

Definition 11 *A scheduler δ of a PA $\mathcal{D}$ is non-Zeno if and only if:*

$$Prob\{\pi \mid \pi \in Paths(\mathcal{D}^{\delta}) \text{ and } \pi \text{ is non-Zeno}\} = 1.$$

Let $\delta_{nz}(\mathcal{D})$ denote the set of non-Zeno schedulers in $\mathcal{D}$.

A scheduler which is not in $\delta_{nz}(\mathcal{D})$ is a Zeno scheduler. The DTMC generated from a Zeno scheduler will have Zeno paths with non-0 probability. Zeno schedulers should be discarded during the verification since the corresponding DTMCs have positive probability to execute unrealistic behaviors.

6.3 PRTS

In this section, we introduce our language in details, including its syntax and operational semantics.

6.3.1 Language Syntax

An elegant modeling language not only increases the aesthetic of the model, but also supplies convenience to users. Meanwhile, the language should cover several facets of the requirements and the model should reflect a system exactly (up to abstraction of irrelevant details). Based on PCSP# and Timed CSP [107], we draw upon existing approaches [82, 10, 116] and create the single notation PRTS to cover a variety of system features.

A PRTS model (hereafter model) is a 3-tuple (Var, V_i, P) where Var is a finite set of finite-domain global variables; V_i is the initial valuation of Var and P is a process which captures the control logic of the system. A process is defined in form of $Proc(\overline{para}) = PExpr$ where $Proc$ is a process name; $\overline{para}$ is a vector of parameters and $PExpr$ is a process expression. All process constructs of PCSP# can be used to construct $PExpr$; besides, a number of timed process constructs are added to capture common real-time system behavior patterns, as shown in Fig. 6.1.

Let d denote an integer constant. Process $Wait[d]$ idles for exactly d time units. In process $P\ timeout[d]\ Q$, the first observable event of P shall occur before d time units elapse (since process $P\ timeout[d]\ Q$ is activated). Otherwise, Q takes over control after exactly d time units. Process $P\ interrupt[d]\ Q$ behaves exactly as P (so that it may engage in multiple observable events) until d time units, and then Q takes over. Process $P\ within[d]$ must react within d time units, i.e., an observable event must be engaged by process P within d time units. Note that $P\ within[d]$ puts a constraint on P. Urgent event prefixing [40], written as $e \overset{!}{\rightarrow} P$, is defined as $(e \rightarrow P)\ within[0]$, i.e., e must occur as soon as it is enabled. Process $P\ deadline[d]$ constrains P to terminate (possibly engaging in multiple observable events) before d time units. In the following, d is referred to as the parameter of the timed process. Given a model, the maximum parameter of the timed processes is called the clock ceiling. Probabilistic choice is recalled in the form of $pcase\ \{pr_0 : P_0;\ pr_1 : P_1;\ \cdots;\ pr_k : P_k\}$, where pr_i is a positive integer to express the probability weight. We remark that ***pcase* is also a timed process construct in our setting, because the τ-transition generated by the *pcase* process must fire immediately**, i.e., it must occur when it is enabled.

In the following, we use a model of the *multi-lift system* to demonstrate system modeling using PRTS. The system is chosen because it is widely used in daily life with real-time (service time constraint), stochastic behaviors (randomized user behaviors) and various data (direction and level status of each lift), which is suitable to demonstrate language features of PRTS. The PRTS model of this system is shown in Fig. 6.2.

```
1. #define NoOfFloors 4;
2. #define NoOfLifts 2;
3. #define upwards 1;
4. #import "PAT.Lib.Lift";
5. var<LiftControl> ctrl = new LiftControl(NoOfFloors,NoOfLifts);
\\**************user's behaviors**************
6. Users() = pcase {
7.            1 : (extreq.0.1{ctrl.AssignExternalRequest(0,1)} -> Skip)within[5]
8.            1 : (intreq.0.0{ctrl.AddInternalRequest(0,0)} -> Skip)within[1]
9.            1 : (intreq.1.0{ctrl.AddInternalRequest(1,0)} -> Skip)within[1]
10.           1 : (extreq.1.0{ctrl.AssignExternalRequest(1,0)} -> Skip)within[5]
11.           1 : (extreq.1.1{ctrl.AssignExternalRequest(1,1)} -> Skip)within[5]
12.           1 : (intreq.0.1{ctrl.AddInternalRequest(0,1)} -> Skip)within[1]
13.           1 : (intreq.1.1{ctrl.AddInternalRequest(1,1)} -> Skip)within[1]
14.           1 : (extreq.2.0{ctrl.AssignExternalRequest(2,0)} -> Skip)within[5]
15.           1 : (extreq.2.1{ctrl.AssignExternalRequest(2,1)} -> Skip)within[5]
16.           1 : (intreq.0.2{ctrl.AddInternalRequest(0,2)} -> Skip)within[1]
17.           1 : (intreq.1.2{ctrl.AddInternalRequest(1,2)} -> Skip)within[1]
18.           1 : (extreq.3.0{ctrl.AssignExternalRequest(3,0)} -> Skip)within[5]
19.           1 : (intreq.0.3{ctrl.AddInternalRequest(0,3)} -> Skip)within[1]
20.           1 : (intreq.1.3{ctrl.AddInternalRequest(1,3)} -> Skip)within[1]
21.       }; Users();
\\*************lift's operations***************
22. Lift(i, level, direction) = case {
23. ctrl.Open(i,level)==1:(serve.level.direction{ctrl.ClearRequests(i,level,direction)}->Lift(i,level,direction))
24. ctrl.KeepMoving(i,level,direction)==1:(reach.(level+direction).direction->Lift(i,level+direction,direction))
25. ctrl.HasAssignment(i)== 1:changedirection.i{ctrl.ChangeDirection(i)}->Lift(i,level,-1*direction)
26. default : idle.i -> Lift(i, level, direction)
27. } within[2];
28. System = (||| x:{0..NoOfLifts-1} @ Lift(x, 0, upwards)) ||| Users();
```

Figure 6.2: A lift system model

Multi-lift System The model has 4 parts. First, some global variable and constants are defined. Lines 1-3 define three constants which denote the number of floors, the number of lifts and the traveling direction of lifts. Here "downwards" is ignored since it can be obtained by $-1 \times upwards$. In this example there are two lifts with 4 floors. Line 4 imports a C# library, which defines a data type *LiftControl* encapsulating all data components and operations of the lift system. Note that it is a design decision whether to maintain the data externally in the C# library or in the model itself. A *LiftControl* object contains multiple data structures, e.g., an integer array for user requests from external button panels, a two dimensional array for requests for internal button panels, etc. Interested readers can refer to PAT (version 3.0 or later, open it with PAT's C# editor and compiler) for its details. The *LiftControl* class also defines multiple data operations, such as assigning an external request for traveling upwards/downwards to a lift, and responding to an internal request for traveling upwards/downwards. The complete C# code for this library can be found in [1]. Next, line 5 of the lift model creates a *LiftControl* object named *ctrl*.

Lines 6 to 21 define a process *Users*(), which models behavior of the users. In this simple modeling, user requests arrive uniformly and repeatedly[1]. There are 14 different requests with 4 floors and 2 lifts (two of which are external requests). Each is given $\frac{1}{14}$ probability, as modeled using $pcase$. For instance, event $extreq$.0.1 models an external request at 0-floor for traveling upwards and event $intreq$.0.0 indicates a user in lift 0 has an internal request to go to level 0. For external requests from middle levels such as level 1 and 2, traveling upwards and downwards are both possible. These events are associated with programs which invoke the methods $AssignExternalRequest$ and $AddInternalRequest$ defined in the external library for assigning requests to lifts through object $ctrl$. Note that user behaviors are subject to real-time constraint. For simplicity, we assume that an external request arrives within 5 time units and an internal request arrives within 1 time unit.

Line 22 to 27 define the process $Lift$ which models an individual lift. The parameters (i, $level$, $direction$) indicate the lift ID, the current floor this lift stays and its traveling direction. Multiple conditional choices $case$ is used to describe the operations of a lift. At line 23, the lift checks whether it should open its door at current level without changing its direction. If the answer is *yes*, all internal requests of reaching this floor, and external requests at this floor with the same traveling direction are removed. At line 24, the lift continues moving without any interruption. Afterwards, level is set to be the next floor. At line 25, if there are requests on the opposite direction, the lift changes its direction to serve requests. The keyword $default$ in line 26 means if no boolean condition above is true, then this choice will be executed, which states that the lift simply idles, waiting to pick up a request some time later. Again, each possible action is constrained by $within[2]$ so that the lift will actively serve requests.

At the top level, the system is modeled as the interleaving of users and lifts at line 28. Here the indexed interleave $||| \ x : \{0..k\}@P(x)$ is a syntactic sugar for $P(0) \ ||| \ P(1) \ ||| \cdots ||| \ P(k)$. All lifts are assumed to travel upwards at the beginning.

6.3.2 Concrete Operational Semantics

In the following we present the operational semantics for PRTS.

A concrete system configuration is a pair (V, P) where V is a variable valuation and $P \in \mathcal{P}$ is a process. For simplicity, a valuation for no variables is written as $\emptyset$. A transition of the system is written in the form $(V, P) \xrightarrow{x} (V', P')$ such that $x \in Act_\tau \cup \mathbb{R}_+$, if the corresponding

[1]A realistic user model can be obtained by mining data of actual lift systems.

distributions are trivial. The operational semantics is defined by associating a set of firing rules with each and every process construct. The firing rules associated with probability and real-time processes are presented as follows. The rest of the rules can be found in Appendix A.

$$\frac{\epsilon \leq d}{(V, Wait[d]) \xrightarrow{\epsilon} (V, Wait[d - \epsilon])} \quad [\ wait1\ ]$$

$$\frac{}{(V, Wait[0]) \xrightarrow{\tau} (V, Skip)} \quad [\ wait2\ ]$$

- The semantics of $Wait[d]$ is captured by rule $wait1$ and $wait2$. Rule $wait1$ states that the process may idle for an arbitrary amount of time ϵ as long as $\epsilon \leq d$. As a result, $Wait[d]$ becomes $Wait[d - \epsilon]$ and the valuation of the variables is unchanged. Rule $wait2$ states that when d is 0, the process becomes $Skip$ via a τ transition.

$$\frac{(V, P) \xrightarrow{e} (V', P'), e \in Act}{(V, P\ timeout[d]\ Q) \xrightarrow{e} (V', P')} \quad [\ to1\ ]$$

$$\frac{(V, P) \xrightarrow{\tau} (V', P')}{(V, P\ timeout[d]\ Q) \xrightarrow{\tau} (V', P'\ timeout[d]\ Q)} \quad [\ to2\ ]$$

$$\frac{(V, P) \xrightarrow{\epsilon} (V', P'), \epsilon \leq d}{(V, P\ timeout[d]\ Q) \xrightarrow{\epsilon} (V', P'\ timeout[d - \epsilon]\ Q)} \quad [\ to3\ ]$$

$$\frac{}{(V, P\ timeout[0]\ Q) \xrightarrow{\tau} (V, Q)} \quad [\ to4\ ]$$

- The semantics of $P\ timeout[d]\ Q$ is captured by rules $to1$ to $to4$. Rule $to1$ states that if P executes an observable event e and becomes P', while changing V to V', then $(V, P\ timeout[d]\ Q)$ becomes (V', P'). In other words, P has performed an observable event before timeout occurs so that Q will not be activated. Rule $to2$

states that if P instead performs a τ transition, then Q and timeout operator remain (since an observable event has not been performed). Rule $to3$ states that if P may idle for less than or equal to d time units, so does $P\ timeout[d]\ Q$. Rule $to4$ states that if d is 0, Q takes over control through a τ transition.

$$\frac{(V, P) \xrightarrow{a} (V', P'), a \neq \checkmark}{(V, P\ interrupt[d]\ Q) \xrightarrow{a} (V', P'\ interrupt[d]\ Q)} \quad [\ ti1\ ]$$

$$\frac{(V, P) \xrightarrow{\epsilon} (V', P'), \epsilon \leq d}{(V, P\ interrupt[d]\ Q) \xrightarrow{\epsilon} (V', P'\ interrupt[d - \epsilon]\ Q)} \quad [\ ti2\ ]$$

$$\frac{(V, P) \xrightarrow{\checkmark} (V', P')}{(V, P\ interrupt[d]\ Q) \xrightarrow{\checkmark} (V',\ P')} \quad [\ ti3\ ]$$

$$\frac{}{(V, P\ interrupt[0]\ Q) \xrightarrow{\tau} (V,\ Q)} \quad [\ ti4\ ]$$

- The semantics of $P\ interrupt[d]\ Q$ is defined by rules $ti1$ to $ti4$. Rule $ti1$ says if P can execute event a, no matter observable or not (which cannot be termination event, which is marked as ✓), then $P\ interrupt[d]\ Q$ can also perform a and Q and d are unchanged. Rule $ti2$ states that if a timed transition ϵ can be performed by P as long as $\epsilon \leq d$, so does $P\ interrupt[d]\ Q$. Rule $ti3$ states that if P terminates, then $interrupt$ operator and Q will be discharged. Rule $ti4$ means when d is 0, Q will take control of the process via a τ transition.

$$\frac{(V, P) \xrightarrow{a} (V', P')}{(V, P\ within[d]) \xrightarrow{a} (V', P')} \quad [\ wi1\ ]$$

$$\frac{(V, P) \xrightarrow{\tau} (V', P')}{(V, P\ within[d]) \xrightarrow{\tau} (V', P'\ within[d])} \quad [\ wi2\ ]$$

$$\frac{(V,P) \xrightarrow{\epsilon} (V',P'), \epsilon \leq d}{(V, P\ within[d]) \xrightarrow{\epsilon} (V,\ P\ within[d-\epsilon])} \quad [\ wi3\ ]$$

- The semantics of $P\ within[d]$ is captured by rules $wi1$ to $wi3$. Rule $wi1$ indicates that if P can execute an observable event a and changes to P', while updating V to V', then $(V, P\ within[d])$ can also execute a and update to (V', P'). In the contrast, if an invisible event is engaged, then $within$ operator remains, which is captured by rule $wi2$. Rule $wi3$ states that the process can idle for ϵ time units as long as P can do so and $\epsilon \leq d$.

$$\frac{(V,P) \xrightarrow{a} (V',P'),\ a \neq \checkmark}{(V, P\ deadline[d]) \xrightarrow{a} (V', P'\ deadline[d])} \quad [\ dl1\ ]$$

$$\frac{(V,P) \xrightarrow{\checkmark} (V',P')}{(V, P\ deadline[d]) \xrightarrow{\checkmark} (V', P')} \quad [\ dl2\ ]$$

$$\frac{(V,P) \xrightarrow{\epsilon} (V',P'), \epsilon \leq d}{(V, P\ deadline[d]) \xrightarrow{\epsilon} (V,\ P\ deadline[d-\epsilon])} \quad [\ dl3\ ]$$

- Rules $dl1$ to $dl3$ define the semantics of $P\ deadline[d]$. Rule $dl1$ states that if P can execute a non-termination event, whether observable or not, so does $P\ deadline[d]$. $dl2$ indicates that when P executes termination event, then $deadline$ operator is discharged. $dl3$ states that the process can idle for ϵ time units as long as P can do so and $\epsilon \leq d$.

$$\frac{}{\begin{array}{l}(V, pcase\ \{pr_0 : P_0;\ \cdots;\ pr_k : P_k\}) \xrightarrow{\tau} \mu \\ \text{where } \mu((V, P_i)) = \frac{pr_i}{pr_0 + \cdots + pr_k} \text{ for all } i \in [0, k]\end{array}} \quad [\ pb\ ]$$

- Rule pb states that if $pcase$ is activated, then it transmits to a distribution μ via action τ. The probability of reaching the successive states follows the probability weight. Note the valuation of variables keeps unchanged, and the missing of timed transition rule indicates a probabilistic choice (i.e., pcase) must be resolved immediately without delay.

Given a PRTS model, a PA can be generated following these rules.

Definition 12 *Let $M = (Var, V_i, P)$ be a PRTS model and AP be a set of propositions on Var. $\mathcal{D}_M$ is a PA $(S, s_{init}, Act, Pr, AP, L)$ such that S is a set of system configurations; $s_{init} = (V_i, P)$; and $Pr \subseteq S \times (Act_\tau \cup \mathbb{R}_+) \times Distr(S)$ is defined by the firing rules; $L((V, P)) = \{ap \mid ap \in AP$ such that $V \vDash ap\}$.*

$\mathcal{D}_M$ is referred to as the *concrete semantics* of M. Because PRTS has a dense-time semantics, the corresponding PA have infinitely many states. In order to apply model checking techniques, a finite-state abstract PA is required.

6.4 Dynamic Zone Abstraction

In this section, we present a fully automated approach to generate a finite-state abstract PA from a model. Without loss of generality, we have the following two assumptions.

- We assume that all variables have finite domains.
- We assume that every process reachable from the initial configuration is finite (as required in [96]), i.e., a process expression has only finitely many process constructs.

As a result, the only source of infinity is timing, or equivalently, the infinitely many possible values for parameters of timed process constructs. For instance, given process $Wait[1]$, there are infinitely many processes that can be reached by a time-transition, e.g., $Wait[0.9]$, $Wait[0.99]$, $Wait[0.999]$, etc. One observation is that for certain properties, the exact value of the parameters are not important, i.e., they can be grouped into equivalent classes. This leads to the idea of using a constraint to capture the value of the parameters. In the following, we summarize *dynamic zone abstraction* proposed in [116], and apply it to PRTS models.

In order to distinguish parameters associated with different process constructs, the first step of the abstraction is to associate timed process constructs (including *pcase*) with clocks. Constraints on the clocks are then used to capture values of the respective parameters. For instance, let $P\ timeout[d]_c\ Q$ denote that process $P\ timeout[d]\ Q$ is associated with clock c. $P\ timeout[d]_c\ Q$ with a constraint $c \leq 5$ represents any process $P\ timeout[d']\ Q$ with

$d' \leq 5$. We assume that each timed process construct is associated with a unique clock. In the following, we define the notion of abstract configuration.

Definition 13 *An abstract system configuration is a triple (V, P, D) such that V is a variable valuation; P is a PRTS process; and D is a zone over the clocks.*

There are usually multiple timed process constructs in a process P. Nonetheless, at one moment not all of the processes are activated, i.e., only some of them are ready to take over control and perform a transition. We write $cl(P)$ to denote the set of clocks **activated** in P and $X = 0$ where X is a set of clocks to denote the conjunction of $c = 0$ for all $c \in X$. A zone D is the conjunction of multiple primitive constraints over a set of **activated** clocks. A primitive constraint is of the form $t \sim d$ or $t_i - t_j \sim d$ where t, t_i, t_j are clocks, d is a constant and $\sim$ is either, $\geq$, $=$ or $\leq$. Note in our setting, the clock constraints are always closed. Intuitively, a zone is the maximal set of clock valuations satisfying the constraint. A zone is empty if and only if the constraint is unsatisfiable. An abstraction configuration (V, P, D) is valid if and only if D is not empty. The following zone operations are relevant. Let D denote a zone. $D^{\uparrow}$ denotes the zone obtained by delaying for an arbitrary amount of time. Note that all clocks proceed at the same rate. For instance, let c be a clock, $(c \leq 5)^{\uparrow}$ is $c \leq \infty$. Given a set of clocks X, $D[X]$ denotes the set of valuations of clocks in X which satisfy D. Further, we write $C(D)$ to denote the clocks of D. Zones can be equivalently represented as Difference Bound Matrices (DBMs) and zone operations can be translated into DBMs manipulation [44, 24].

In order to define the abstract PA, we define a set of *abstract* firing rules. The abstract firing rules eliminate timed transitions all together and use zones to ensure a process behaves correctly with respect to timing requirements. To distinguish from concrete transitions, an abstract transition is written in the form: $(V, P, D) \overset{e}{\rightsquigarrow} (V', P', D')$. Given a process P_T, $idle(P)$ is defined to be the maximum zone such that P can idle before performing an event-transition. For instance, $idle(P\ deadline[5]_c) = idle(P) \land c \leq 5$, i.e., $P\ deadline[5]_c$ can idle as long as P can idle and the reading of c is no bigger than 5. Fig. 6.3 shows the detailed definition.

Rules $idle1$ to $idle5$ state that if the process is un-timed and none of its sub-processes is activated, then function $idle$ returns true, which means that the process may idle for an arbitrary amount of time. Rules $idle6$ to $idle9$ state that if sub-processes of the process are activated, then function $idle$ is applied to the sub-processes. For instance, if the process

$idle(Stop)$	$= true$	$- idle1$
$idle(Skip)$	$= true$	$- idle2$
$idle(e \rightarrow P)$	$= true$	$- idle3$
$idle(a\{program\} \rightarrow P)$	$= true$	$- idle4$
$idle(\textbf{if } (b)\ \{P\} \textbf{ else } \{Q\})$	$= true$	$- idle5$
$idle(P \mid Q)$	$= idle(P) \wedge idle(Q)$	$- idle6$
$idle(P \setminus X)$	$= idle(P)$	$- idle7$
$idle(P;\ Q)$	$= idle(P)$	$- idle8$
$idle(P \parallel Q)$	$= idle(P) \wedge idle(Q)$	$- idle9$
$idle(P)$	$= idle(Q) \quad \text{if } P \mathrel{\widehat{=}} Q$	$- idle10$
$idle(Wait[d]_c)$	$= c \leq d$	$- idle11$
$idle(P\ timeout[d]_c\ Q)$	$= c \leq d \wedge idle(P)$	$- idle12$
$idle(P\ interrupt[d]_c\ Q)$	$= c \leq d \wedge idle(P)$	$- idle13$
$idle(P\ within[d]_c)$	$= c \leq d \wedge idle(P)$	$- idle14$
$idle(P\ deadline[d]_c)$	$= c \leq d \wedge idle(P)$	$- idle15$
$idle(pcase_c\{pr_0 : P_0; \cdots; pr_k : P_k\})$	$= c = 0$	$- idle16$

Figure 6.3: Idling calculation

is a choice (rule $idle6$) or a parallel composition (rule $idle9$) of P and Q, then the result is $idle(P) \wedge idle(Q)$. Intuitively, this means that process $P \mid Q$ (or $P \parallel Q$) may idle as long as both P and Q can idle. $idle10$ defines the case for process referencing. Rules $idle11$ to $idle15$ define the cases when the process is timed. For instance, process $Wait[d]_c$ may idle as long as c is less or equal to d. Lastly, since there is no idling allowed for $pcase_c$, the value of c must always be 0, which is captured by $idle16$.

The following shows the abstract firing rules for real-time and probability processes. Same as the concrete rules, if the probability of a transition is 1, the probability will be removed from the labeling for simplicity. Other rules are listed in Appendix B.

$$\frac{}{(V, Wait[d]_c, D) \overset{\tau}{\rightsquigarrow} (V, Skip, D^{\uparrow} \wedge c = d)} \quad [\ ade\ ]$$

- Rule ade states that process $Wait[d]$ idles for exactly d time units and then engages in event τ and the process transforms to $Skip$. Note that the zone of the target configuration is $D^{\uparrow} \wedge c = d$. Intuitively, it means that the transition occurs sometime in the future (captured by $D^{\uparrow}$) when c reads d (captured by $c = d$). It should be intuitively clear that this is 'equivalent' to the concrete firing rules.

$$\frac{}{(V, P\ timeout[d]_c\ Q, D) \overset{\tau}{\rightsquigarrow} (V, Q, D^{\uparrow} \wedge c = d \wedge idle(P))} \quad [\ ato1\ ]$$

$$\frac{(V, P, D) \overset{\tau}{\rightsquigarrow} (V', P', D')}{(V, P\ timeout[d]_c\ Q, D) \overset{\tau}{\rightsquigarrow} (V', P'\ timeout[d]_c\ Q, D^{\uparrow} \wedge D' \wedge c \leq d)}\ [\ ato2\ ]$$

$$\frac{(V, P, D) \overset{e}{\rightsquigarrow} (V', P', D'), e \neq \tau}{(V, P\ timeout[d]_c\ Q, D) \overset{e}{\rightsquigarrow} (V', P', D^{\uparrow} \wedge D' \wedge c \leq d)}\ [\ ato3\ ]$$

- Rules $ato1$, $ato2$ and $ato3$ capture the abstract semantics of $P\ timeout[d]\ Q$. Depending on when the first event of P takes place and whether it is observable, process $P\ timeout[d]\ Q$ behaves differently in three ways. Rule $ato1$ states that if P generates a τ transition, the $timeout$ construct remains. Furthermore, the target zone $D' \wedge c \leq d$ constrains that the transition must take place no later than d time units. In the contrast, rule $ato2$ states that if P generates an observable transition, then the $timeout$ construct is removed. Similarly, it is constrained that the transition must occur no later than d time units. Rule $ato3$ captures the case when timeout occurs. Namely, timeout occurs if and only if the reading of c is exactly d and, further, P must be able to idle until c reads d.

$$\frac{(V, P, D) \overset{a}{\rightsquigarrow} (V', P', D'), a \neq \checkmark}{(V, P\ interrupt[d]_c\ Q, D) \overset{a}{\rightsquigarrow} (V', P'\ interrupt[d]_c\ Q, D^{\uparrow} \wedge D' \wedge c \leq d)}\ [\ ait1\ ]$$

$$\frac{}{(V, P\ interrupt[d]_c\ Q, D) \overset{\tau}{\rightsquigarrow} (V, Q, D^{\uparrow} \wedge c = d \wedge idle(P))}\ [\ ait2\ ]$$

$$\frac{(V, P, D) \overset{\checkmark}{\rightsquigarrow} (V', P', D')}{(V, P\ interrupt[d]_c\ Q, D) \overset{\checkmark}{\rightsquigarrow} (V', P', D^{\uparrow} \wedge D' \wedge c \leq d)}\ [\ ait3\ ]$$

- Rules $ait1$, $ait2$, $ait3$ describe the abstract semantics of $P\ interrupt[d]\ Q$. $ait1$ states that P can execute an observable event a (not the termination event ✓) before d time units, and the $interrupt$ operator remains. $ait2$ indicates that when $t = d$, $interrupt$ will happen via a τ transition and Q will take over afterwards. $ait3$ states if ✓ happens before d time units, then the $interrupt$ operator will be ruled out.

$$\frac{(V, P, D) \overset{\tau}{\rightsquigarrow} (V', P', D')}{(V, P\ within[d]_c, D) \overset{\tau}{\rightsquigarrow} (V', P'\ within[d]_c, D^{\uparrow} \wedge D' \wedge c \leq d)} \quad [\ awi1\ ]$$

$$\frac{(V, P, D) \overset{e}{\rightsquigarrow} (V', P', D')}{(V, P\ within[d]_c, D) \overset{e}{\rightsquigarrow} (V', P', D^{\uparrow} \wedge D' \wedge c \leq d)} \quad [\ awi2\ ]$$

- Rules $awi1$ and $awi2$ define the abstract semantics of $P\ within[d]$. Rule $awi1$ states that if a τ transition occurs within d time units, then the resultant process is of the form $P'\ within[d]$, which means that it is yet to perform some observable event before d time units. Rule $awi2$ states that once an observable event occurs, the $within$ construct is removed.

$$\frac{(V, P, D) \overset{a}{\rightsquigarrow} (V', P', D'), a \neq \checkmark}{(V, P\ deadline[d]_c, D) \overset{a}{\rightsquigarrow} (V', P'\ deadline[d]_c, D^{\uparrow} \wedge D' \wedge c \leq d)} \quad [\ adl1\ ]$$

$$\frac{(V, P, D) \overset{\checkmark}{\rightsquigarrow} (V', P', D')}{(V, P\ deadline[d]_c, D) \overset{\checkmark}{\rightsquigarrow} (V', P', D^{\uparrow} \wedge D' \wedge c \leq d)} \quad [\ adl2\ ]$$

- Rules $adl1$ and $adl2$ define the abstract semantics of $P\ deadline[d]$. Rule $adl1$ ensures that all transitions of P must occur within d time units. Rule $adl2$ states that if P terminates, then $deadline$ is removed.

$$\frac{}{(V, pcase_c\ \{pr_0 : P_0;\ pr_1 : P_1;\ \cdots;\ pr_k : P_k\}, D) \overset{\tau}{\rightsquigarrow} \mu} \quad [\ apcase\ ]$$

where $\mu(V, P_i, D^{\uparrow} \wedge c = 0 \wedge cl(P_i) = 0) = \frac{pr_i}{pr_0 + \cdots + pr_k}$ for all $i \in [0, k]$

- Rule $apcase$ captures the abstract semantics of $pcase$. The new zone after i^{th} transition is $D \wedge c = 0 \wedge cl(P_i) = 0$, which means c is still 0 after the transition, and all new activated clocks in P_i should be 0. This indicates that the corresponding τ transition is instantaneous.

Definition 14 *Let $M = (Var, V_i, P)$ be a model and AP be a set of propositions on Var. $\mathcal{D}^a_M = (S_a, init_a, Act, Pr_a, AP, L_a)$ is the abstract PA such that S_a is a set of valid abstract system configurations; $init_a = (V_i, P, D_{init})$ is the initial abstract configuration where D_{init} is $cl(P) = 0$; $Pr_a \subseteq S \times Act_\tau \times Distr(S)$ representing the transition relation, which is defined by the abstract firing rules; $L_a((V, P, D)) = \{ap \mid ap \in AP$ such that $V \models ap\}$. If $(V, P, D) \xrightarrow{x,\mu} (V', Q, D')$, then $D' = D[cl(Q)] \wedge cl(Q) - cl(P) = 0$.*

Informally, given $(V, P, D) \in S_a$, if (V', Q, D') is an abstract configuration obtained by applying an abstract firing rule to the given state, D' is obtained by firstly pruning all clocks which are not in $cl(Q)$ and then setting clocks associated with newly activated processes (i.e., $cl(Q) - cl(P)$) to be 0. The construct of $\mathcal{D}^a_M$ is illustrated in the following example.

Example Assume a model $M = (\emptyset, \emptyset, S)$ such that process S is defined as follows.

$$
\begin{aligned}
P = (a \rightarrow pcase_{c_0}\{ & \\
&1 : Wait[1]_{c_1};\ (b \rightarrow Skip) within[0]_{c_3} \\
&1 : Wait[2]_{c_2};\ b \rightarrow Skip \\
\})\ deadline[2]_{c_4}; &
\end{aligned}
$$

$$Q = (Wait[2]_{c_5};\ b \rightarrow Skip) within[2]_{c_6};$$

$$S = P \parallel Q;$$

Note that $c_i, 0 \leq i \leq 6$ are clocks associated to timed constructors. Using the abstract firing rules, the abstract PA can be generated and is shown in Fig. 6.4.

In Fig. 6.4, we omit the self-looping transition on deadlocking states for readability. The initial configuration state s_{init} is $(\emptyset, S, c_4 = c_5 = c_6 = 0)$, where clock c_4 is associated with $deadline[2]$ in P, and c_5 and c_6 are associated with $Wait[2]$ and $within[2]$ in Q respectively. Other clocks are not activated in the initial state since their corresponding timed process constructs are not taking control of current process. Applying rule $adl2$, we get the transitions from state s_{init} to state s_1 via event a, and $0 \leq c_4 = c_5 = c_6 \leq 2$ after this transition. $c_0 = 0$ is also in zone of s_1 since $pcase_{c_0}$ is activated. Alternatively, applying rule ade, another τ transition from s_{init} to s_2 is generated because of $Wait[2]_{c_5}$ in Q, which makes $c_4 = c_5 = c_6 = 2$. Note zone in s_2 is $c_4 = c_6 = 2$, because c_5 is expired and no longer activated after this transition. The $pcase_{c_0}$ takes control of the process in s_1, and the probabilistic transitions

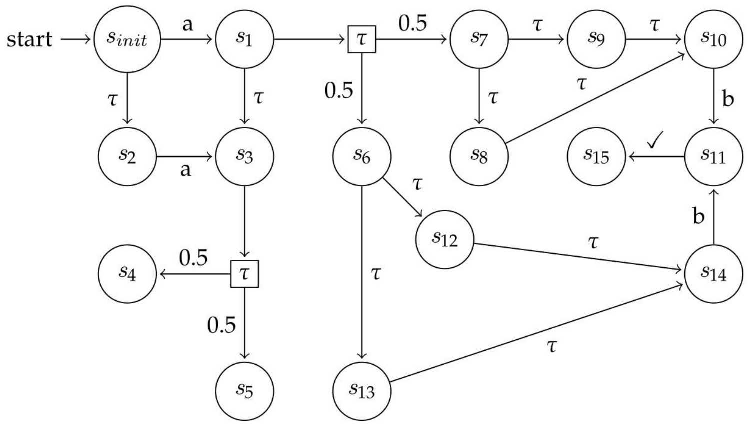


Figure 6.4: An Abstract Model

should happen immediately. Therefore, we have a distribution labeled by τ from s_1 to s_6 and s_7 according to rule $apcase$, and the transition probabilities are both 0.5. c_0 is still 0 after these transitions, and is ruled out from the generated zone since $pcase_{c_0}$ expires. In s_6, P has become $(Wait[1]_{c_1};\ (b \to Skip)within[0]_{c_3})\ deadline[2]_{c_4}$ and the corresponding zone is $0 \leq c_4 = c_5 = c_6 \leq 2 \wedge c_1 = 0$. Applying rule ade again, $Wait[1]_{c_1}$ in P generates a τ transition from s_6 to s_{12}. Process $S = P \parallel Q$ in s_{12} is now

$$((b \to Skip)within[0]_{c_3})\ deadline[2]_{c_4} \parallel$$

$$(Wait[2]_{c_5};\ b \to Skip)within[2]_{c_6}$$

$c_3 = 0$ and $1 \leq c_4 = c_5 = c_6 \leq 2$ in zone of s_{12}. Because b must execute simultaneously in both P and Q, so applying ade, $Wait[2]_{c_5}$ in Q generates a τ transition from s_{12} to s_{14}. Note that this transition must take no time due to $within[0]$ in P. Therefore $c_4 = c_5 = c_6 = 2$ should be true both before and after this transition. Afterwards, b is enabled in both P and Q, so a transition labeled with b is generated from s_{14} to s_{11}. Meanwhile $within[0]_{c_3}$ in P and $within[2]_{c_6}$ in Q are removed since an observable event occurs. $deadline[2]_{c_4}$ in P remains and $c_4 = 2$ is still active. Applying rule $adl2$, a transition from s_{11} to s_{15} labeled with ✓ is generated because of $Skip$ in both P and Q. Afterwards, $deadline[2]_{c_4}$ expires. Other transitions and states are similarly obtained. □

Correspondingly, we can define abstract DTMC, whose formal definition is as follows.

Definition 15 *An abstract DTMC is a tuple $(S_a, init_a, Act, Pr_a, AP, L_a)$, where S_a is a set of valid abstract system configurations; $init_a = (V_i, P, D_{init})$ is the initial abstract configuration* *where D_{init} is $cl(P) = 0$; Pr_a is a function:$S_a \rightarrow Act_\tau \times Distr(S)$ representing the transition relation; AP is a set of propositions on Var; $L_a((V, P, D)) = \{ap \mid ap \in AP$ such that $V \models ap\}$.*

PRTS vs. PTA Probabilistic Timed Automata (PTA) are an extension of Timed automata with discrete probabilistic distribution, and they are widely used in probabilistic real-time domain. Therefore, it is meaningful to investigate the relationship between PRTS and PTA.

Since PRTS is based on Timed CSP [107], the timed constraints in PRTS are closed while PTA may contain both closed and open timing constraints. Different from PTA, PRTS assumes probabilistic choices are instantaneous without time delay. These constraints make PRTS (assuming variables are finite-domained and process expressions are always finite) less expressive than PTA; however, PRTS is still useful in modeling probabilistic real-time systems, which is shown in PRTS models mentioned above and those used in Section 6.6.

On the other hand, PRTS allows probabilistic choices whose probability distributions are decided at run-time, which makes it flexible to model real life systems, since the probability of transitions in some scenario may change from time to time. Further, the external library increases the convenience of modeling for systems with complicated calculations. In addition, it can be shown that every clock is bounded from above in PRTS (see the definition of $idle$ in Fig. 6.3 and abstract firing rules defined above), which implies zone normalization [103] is not essential in our setting. Lastly, model checking with non-Zenoness based on the zone graphs is feasible in PRTS, as we show in the following section.

6.5 Verification of Abstract PA

In the section, we show that the abstract PA $\mathcal{D}_M^a$ is subject to probabilistic model checking.

6.5.1 Finiteness

We first show that all abstract PAs generated via our dynamic zone abstraction are finite, which is captured by the following theorem.

Theorem 6.5.1 *$\mathcal{D}_M^a$ is finite for any PRTS model M.*

Proof By definition, the size of $\mathcal{D}^a_M$ is bounded by $|V| \times |P| \times |D|$, where $|V|$ denotes the number of variable valuations; $|P|$ denotes the number of processes; and $|D|$ denotes the number of zones. By assumption in Section 6.4, all variables have finite domains and therefore $|V|$ is finite. Similarly, all reachable processes are finite by assumption. Thus, $|P|$ is finite if and only if values for parameters of the timed processes are finite and the number of clocks in $cl(P)$ is finite. It can be shown that all abstract firing rules preserve the parameters and therefore values for parameters are finite. By assumption, there can be only finitely many process constructs in any reachable process. Thus, $cl(P)$ is always finite. Therefore, $|P|$ is finite. Lastly it is known that the number of zones is finite given finitely many clocks and only bounded integer constants [44]. As a result, we conclude that $|D|$ is finite and then $\mathcal{D}^a_M$ is finite. □

6.5.2 Over-approximation

In this part, we show that given an LTL-X property ϕ, $\mathcal{D}^a_M$ is over-approximation of $\mathcal{D}_M$, i.e., for any scheduler δ in $\mathcal{D}_M$, there exists a scheduler η in $\mathcal{D}^a_M$ satisfying $\mathcal{P}^{\eta}_{\mathcal{D}^a_M}(\phi) \geq \mathcal{P}^{\delta}_{\mathcal{D}_M}(\phi)$.

First, in a concrete DTMC, there are two kinds of transitions: timed transitions $s \xrightarrow{\epsilon} s'$, which always leads to a trivial distribution according to the semantics, and event transitions $s \xrightarrow{a,p} s'$ such that a can be both observable and un-observable. For simplicity, we write $s \xrightarrow{\epsilon,a,p} s'$ to denote that there exists s_0 such that $s \xrightarrow{\epsilon} s_0 \xrightarrow{a,p} s'$. Here p is unchanged since timed transition always has probability 1. Note in this case we assume ϵ could be 0, which indicates a is not delayed. On the other hand, in an abstract PA, abstract DTMC without timed transitions can be obtained, i.e, for any $s \overset{a,p}{\rightsquigarrow} s'$ in an abstract DTMC, $a \notin \mathbb{R}_+$.

Next, we define *probabilistic time-abstract simulation* of DTMC as follows.

Definition 16 *A* probabilistic time-abstract simulation relation *between a concrete DTMC $C = (S_c, init_c, Act, Pr_c, AP, L_c)$ and an abstract DTMC $C_a = (S_a, init_a, Act, Pr_a, AP, L_a)$ is a relation $\mathcal{R} \subseteq S_c \times S_a$ such that the following condition.*

C1: If $(s_c, s_a) \in \mathcal{R}$ and $s_c \xrightarrow{\epsilon,a,p} s'_c$ for $\epsilon \geq 0$, $a \in Act_\tau$ and $p \in (0, 1]$, then there exists s'_a such that $s_a \overset{a,p}{\rightsquigarrow} s'_a$ and $(s'_c, s'_a) \in \mathcal{R}$;

C2: $(init_c, init_a) \in \mathcal{R}$.

Proposition 6.5.2 *If abstract DTMC C_a time-abstract simulates concrete DTMC C, then given a certain LTL-X property ϕ, $\mathcal{P}_{C_a}(\phi) \geq \mathcal{P}_C(\phi)$.* $\square$

Theorem 6.5.3 *For each scheduler σ of $\mathcal{D}_M$, there exists a scheduler η of $\mathcal{D}^a_M$ such that $\mathcal{D}^\sigma_M$ probabilistic time-abstract simulates $(\mathcal{D}^a_M)^\eta$.*

Proof Given $\mathcal{D}^\sigma_M = (S_c, init_c, Act, Pr_c, AP, L_c)$, we first show that a sequence of timed transitions can be treated as one timed transition in verifying LTL-X property ϕ. In other words, if we have $s_1 \overset{\epsilon 1}{\to} s_2 \overset{\epsilon 2}{\to} s_3$ as a part of a path in $\mathcal{D}^\sigma_M$, one transition $s_1 \xrightarrow{\epsilon 1 + \epsilon 2} s_3$ is equivalent to the previous sequence. This is trivial since timed transitions always have probability 1 and do not affect the variable valuation in our setting. So that the probability of ϕ on this path keeps the same. Therefore, here we just consider $\mathcal{D}^\sigma_M$ in which no successive timed transitions exist.

By definition, we need to find a scheduler η on $\mathcal{D}^a_M$ such that $(\mathcal{D}^a_M)^\eta = (S_a, init_a, Act, Pr_a, AP, L_a)$, and there exists a probabilistic time-abstract simulation relation $\mathcal{R}$ between S_c and S_a. According to Pr_c defined by σ, η is constructed to choose corresponding actions and distributions in each state s of $\mathcal{D}^a_M$.

We define $\mathcal{R}$ as follows: $\forall\, s_c = (V_c, P_c) \in S_c; \forall\, s_a = (V_a, P_a, D) \in S_a$, $(s_c, s_a) \in \mathcal{R}$ if and only if $V_c = V_a$, i.e., $L_c(s_c) = L_a(s_a)$, and P_c is abstracted by P_a with D. P_c is abstracted by P_a with D if and only if the following two conditions are satisfied.

- P_c differs from P_a only by the parameters of the timed process constructs.
- For every timed process construct of P_c, let d be the associated parameter; let d' be the constant associated with the corresponding construct in P_a. If the construct is not activated in P, then $d = d'$. If the construct is activated with a clock c in P_a, then $c = d' - d$ satisfies $D[c]$.

For instance, if $P_c = Wait[3];\ Wait[5]$ and $P_a = Wait[4]_{c_1};\ Wait[5]_{c_2}$, then P_a with zone $c_1 \leq 4$ abstracts P_c. Next, we show that $C1$ and $C2$ of Definition 16 are satisfied by $\mathcal{R}$.

$C2$ is proved straightforwardly. Therefore we focus on the proof of $C1$, which is done by structural induction. Four cases are exemplified to show the correctness of $C1$, where P_c is set to be $Wait[d]$, $P\ timeout[d]\ Q$ and $pcase\ \{pr_0 : P_0;\ \cdots;\ pr_k : P_k\}$.

- If P_c is $Wait[d]$ and $((V_c, P_c), (V_a, P_a, D_a)) \in \mathcal{R}$, then P_a is $Wait[d']_c$ such that $c = d'$ satisfies $D'_a[c]$. By rule $wait1$ and $wait2$, $(V_c, Wait[d]) \xrightarrow{d} (V_c, Wait[0]) \xrightarrow{\tau} (V_c, Skip)$. Therefore $\sigma((V_c, P_c)) = (d, \mu_1)$ and $\sigma((V_c, Wait[0])) = (\tau, \mu_2)$ where μ_1 and μ_2 are trivial. Then in the generated concrete DTMC, we have $(V_c, Wait[d]) \xrightarrow{d,\tau,1} (V_c, Skip)$. By rule ade, $(V_a, P_a, D_a) \overset{\tau}{\rightsquigarrow} (V_a, Skip, D_a^{\uparrow} \wedge c = d)$ and thus $(V_a, P_a, D_a) \overset{\tau}{\rightsquigarrow} (V_a, Skip, D')$ where $D' = true$ (since c is removed due to its expiration and $cl(Skip) = \emptyset$). Then let $\eta((V_a, P_a, D_a)) = (\tau, \mu)$ such that $\mu((V_a, Skip, D')) = 1$, i.e., $(V_a, P_a, D_a) \overset{\tau,1}{\rightsquigarrow} (V_a, Skip, true)$. It is trivial to show $((V_c, Skip), (V_a, Skip, true)) \in \mathcal{R}$. Therefore $C1$ is true.

- If P_c is $P\ timeout[d]\ Q$ and $((V_c, P_c), (V_a, P_a, D_a)) \in \mathcal{R}$, then P_a is $P'\ timeout[d]_t\ Q$ such that P is abstracted by P' with D_a. Assume $(V_c, P) \xrightarrow{\epsilon} (V_c, P') \xrightarrow{e} (V_1, P_1)$ for some $\epsilon \leq d$. By rule $to3$ and $to1$, $(V_c, P_c) \xrightarrow{\epsilon} (V_c, P'_c) \xrightarrow{e} (V_1, P_1)$. Therefore $\sigma((V_c, P_c)) = (\epsilon, \mu_1)$ and $\sigma((V_c, P'_c)) = (e, \mu_2)$ where μ_1 and μ_2 are trivial. Then in the generated concrete DTMC, we have $(V_c, P_c) \xrightarrow{\epsilon,e,1} (V_1, P_1)$. By induction hypothesis, $(V_a, P', D_a) \overset{e,1}{\rightsquigarrow} (V_1, P'_1, D'_a)$ such that $((V_1, P_1), (V_1, P'_1, D'_a)) \in \mathcal{R}$. By rule $ato2$, $(V_a, P_a, D_a) \overset{e}{\rightsquigarrow} (V_1, P'_1, D'_a \wedge D_a^{\uparrow} \wedge t \leq d)$. Then let $\eta((V_a, P_a, D_a)) = (e, \mu)$ such that $\mu((V_1, P'_1, D'_a \wedge D_a^{\uparrow} \wedge t \leq d)) = 1$. By assumption, $\epsilon \leq d$ and thus it is easy to show that P'_1 with $D'_a \wedge D_a^{\uparrow} \wedge t \leq d$ abstracts P_1. Thus, $C1$ is satisfied. Similarly, we build η in the case where $(V_c, P\ timeout[d]\ Q) \xrightarrow{\epsilon,\tau} (V_1, P_1\ timeout[d]\ Q)$ for some $\epsilon \leq d$ or $(V_c, P\ timeout[d]\ Q) \xrightarrow{d,\tau} (V_1, Q)$.

- If P_c is $pcase\ \{pr_0 : P_0;\ \cdots;\ pr_k : P_k\}$ and $((V_c, P_c), (V_a, P_a, D_a)) \in \mathcal{R}$, then P_a is $pcase_c\ \{pr_0 : P_0;\ \cdots;\ pr_k : P_k\}$ such that $c = 0$. According to rule pb, $(V_c, P_c) \xrightarrow{\tau} \mu$ such that $\mu(V_c, P_i) = p_i = \frac{pr_i}{pr_0 + pr_1 + \cdots + pr_k}$. Therefore $\sigma((V_c, P_c)) = (\tau, \mu)$, and in the generated concrete DTMC, $(V_c, P_c) \xrightarrow{0,\tau,p_i} (V_c, P_i)$ such that 0 means $pcase$ cannot be delayed. By rule $apcase$, $(V_a, P_a, D_a) \overset{\tau}{\rightsquigarrow} \mu$, and $\mu((V_a, P_i, D_a^{\uparrow} \wedge c = 0 \wedge cl(P_i) = 0)) = p_i$. After ruling out expired clock c, the generated zone D' is $D_a^{\uparrow} \wedge cl(P_i) = 0$. Then let $\eta((V_a, P_a, D_a)) = (\tau, \mu)$ such that $\mu((V_a, P_i, D')) = p_i$, therefore $(V_a, P_a, D_a) \overset{\tau,p_i}{\rightsquigarrow} (V_a, P_i, D')$. It is trivial to show $((V_c, P_i), (V_a, P_i, D')) \in \mathcal{R}$. Therefore $C1$ is true.

Other cases can be proved to satisfy $C1$ similarly. We thus conclude that for any scheduler σ of $\mathcal{D}_M$, there is a scheduler η of $\mathcal{D}_M^a$ such that a probabilistic time-abstract simulation relation between $\mathcal{D}_M^{\sigma}$ and $(\mathcal{D}_M^a)^{\eta}$ exists. $\square$

Based on the above theorem, we relate the verification results of abstract model and the concrete model, which is captured by the following theorem.

Theorem 6.5.4 *Given an LTL-X property ϕ, $\mathcal{P}^{max}_{\mathcal{D}_M}(\phi) \leq \mathcal{P}^{max}_{\mathcal{D}^a_M}(\phi)$.*

Proof This theorem holds by proposition 6.5.2 and Theorem 6.5.3. □

In the following, we show, with an example, that the probability may not be exact, i.e., $\mathcal{P}^{max}_{\mathcal{D}_M}(\phi) < \mathcal{P}^{max}_{\mathcal{D}^a_M}(\phi)$. Let us take the model in Fig. 6.4 as an example, which is inspired by examples in [82]. Suppose that ϕ is $\Diamond b$. In Fig. 6.4, the maximal probability in the abstract model should be 1 since two probabilistic choices from s_1 can eventually execute b and reach s_{11}. Now let us analyze the original PRTS model in Fig. 6.4.

Note that b is a common event in P and Q, therefore it can happen only when it is enabled by both processes simultaneously. In process Q, b occurs exactly after 2 time units due to $Wait$[2] and $within$[2]. As a result, b must also happen at 2 time units sharp in P. For P, the whole process should finish within 2 time units because of *deadline*[2]. Event a can happen between 0 to 2 time units. Since probabilistic choices do not take any time, and b should happen after 2 time units sharp from the beginning, we can find that the first branch of probabilistic choice, $Wait$[1]; $(b \rightarrow Stop)within$[0], must be activated after 1 time unit; and the second branch of probabilistic choice, $Wait$[2]; $b \rightarrow Stop$, should be activated without any delay. Therefore, event a should occur either after exactly 1 time unit, or immediately. There are two schedulers corresponding to these two scenarios. In both DTMC generated via these two schedulers, there will be only one branch satisfies that b can eventually happen. Therefore, the maximal probability of S executing b should be 0.5 instead of 1 in the abstract model. Informally speaking, a scheduler in the abstract model may not be feasible in the concrete model, as it may be the result of collapsing multiple schedulers at different time points.

Given that the probability of satisfying ϕ in the abstract model is an over-approximation, it can be used to check whether the probability of a system satisfying a property is under some threshold λ, denoted as $\leq \lambda$. For example, in the multi-lift system, a meaningful property could be "the probability that users are passed by should not exceed 0.1". Assume the maximal probability for some property in an abstract model is p, then if $p \leq \lambda$, the original model is also under the threshold. Otherwise, the original model may or may not satisfy its maximal probability for this property $\leq \lambda$.

6.5.3 Non-Zenoness

In this section, we discuss model checking with non-Zenoness in PRTS. Zeno schedulers may exist in PRTS models. One simple PRTS example representing this scenario is demonstrated as follows.

$$\begin{aligned}
&P = Q;\ b \rightarrow Skip;\\
&Q = a \rightarrow Q \Box Skip;\\
&R = (b \rightarrow Skip) within[1];\\
&System = P \parallel R;
\end{aligned}$$

In this example, process $System$ is composed by P and R, which are running in parallel. Event b is required to happen within 1 time unit whenever process R is activated. Meanwhile, process Q has two choices, both of which have probability 1. One is to engage in event a and then behave as Q, and the other is to terminate. If a scheduler of $System$, say δ, always chooses to engage event a, b cannot happen and the timed constraint $within[1]$ always exists. Thus an infinite path happens within 1 time unit, which is unrealistic. Furthermore, this path has probability 1, indicating δ is a Zeno scheduler. The existence of Zeno schedulers may affect the results of the properties we want to check, which is shown in Section 6.6. Therefore, we need a method to check whether a scheduler is Zeno or not, and Zeno schedulers should be removed during the verification.

Before introducing the detailed verification algorithms, we first investigate the relation between the Zeno schedulers in concrete PA and its abstraction. When we rule out a Zeno scheduler in the abstract model, *is it possible that some non-Zeno schedulers in the concrete model are also removed?* If so, the verification under non-Zeno assumption is not reliable. To answer this question, we rely on identifying BSCCs in abstract DTMC. First, A transition is called *instantaneous* if it cannot be delayed, such as firing the abstract rule for $pcase$ or transition labeled with a in $(a \rightarrow Skip)within[0]$. Given a BSCC $\mathcal{B}$ composed by set of states T in an abstract DTMC, let $loopCLK(T)$ denote the set $\{x \mid \forall (V, P, D) \in T.\ x \in C(D)\}$, i.e., the clocks which are present in every configuration of the $\mathcal{B}$. We define $\mathcal{B}$ is *non-Zeno* if it satisfies

1. $loopCLK(T) = \varnothing$.
2. Not all transitions in $\mathcal{B}$ are instantaneous.

A BSCC is $Zeno$ if it is not *non-Zeno*. Now we have the following theorem.

Theorem 6.5.5 *Given an abstract PA $\mathcal{D}$ and its scheduler δ, δ is Zeno iff DTMC $\mathcal{D}^{\delta}$ has at least one reachable Zeno BSCC.*

Proof (If) Assume $\mathcal{B}$ is a Zeno BSCC of $\mathcal{D}^{\delta}$, composed by states set T. Then either 1) $loopCLK(T) \neq \emptyset$ or 2) all transitions in $\mathcal{B}$ are instantaneous. If 1) is true, assume $c \in loopCLK(T)$. For any infinite path reaching $\mathcal{B}$, say π, it will stay in $\mathcal{B}$ forever, and c never expires on π. According to our setting, all timed constraints are bounded, so does the parameter of c. Therefore, infinitely many steps happen in a bounded time interval. Thus π is Zeno. If 2) is true, for any infinite path reaching $\mathcal{B}$, say π, it will has infinitely many continuous instantaneous steps without leaving $\mathcal{B}$. In other words, infinitely many steps happen in 0 time unit, therefore π is Zeno. As a result, as long as $\mathcal{B}$ is Zeno, all infinite paths reaching $\mathcal{B}$ are Zeno. Because $\mathcal{D}^{\delta}$ is reachable, the probability of reaching $\mathcal{B}$ is positive. Therefore δ is Zeno.

(Only-If) Since $\mathcal{D}^{\delta}$ is Zeno, there must be some Zeno paths in $\mathcal{D}^{\delta}$ satisfying the sum of their probability is positive. Furthermore, all infinite paths with positive probability must reach certain BSCC. Therefore, there exists at least one BSCC that Zeno paths can reach, which can be denoted as $\mathcal{B}$. Moreover, it is known that any path reaching a BSCC can visit all states and transitions in this BSCC infinitely often. Therefore, any Zeno path reaching $\mathcal{B}$, say π, visits all states and transitions in $\mathcal{B}$ infinitely often. Since π is Zeno, there must be a finite bound for its infinite execution. Therefore, all transitions in $\mathcal{B}$ must be instantaneous. Therefore $\mathcal{B}$ is a Zeno BSCC. So the theorem holds. □

The following theorem then answers the question about the relation between the Zeno schedulers in concrete PA and its abstraction.

Theorem 6.5.6 *Let $\mathcal{D}$ be a PRTS model. For each non-Zeno scheduler σ of $\mathcal{D}_M$, there exists a non-Zeno scheduler η of $\mathcal{D}_M^a$ satisfying $\mathcal{D}_M^{\sigma}$ simulates $(\mathcal{D}_M^a)^{\eta}$.*

Proof According to Theorem 6.5.3, for non-Zeno scheduler σ of $\mathcal{D}_M$, there must be a scheduler η of $\mathcal{D}_M^a$ satisfying $\mathcal{D}_M^{\sigma}$ simulates $(\mathcal{D}_M^a)^{\eta}$ following a relation $\mathcal{R}$. Therefore, we simply need to prove η is a non-Zeno scheduler in the abstract model.

Because σ is a non-Zeno scheduler of $\mathcal{D}_M$, according to Theorem 6.5.5, all BSCCs in $\mathcal{D}_M^{\sigma}$ are non-Zeno. For each BSCC $\mathcal{B}$ in $\mathcal{D}_M^{\sigma}$, we set $\mathcal{B}_a = \{s_a \mid \forall\, s \in \mathcal{B}, (s, s_a) \in \mathcal{R}\}$ as a set of states in $(\mathcal{D}_M^a)^{\eta}$. Obviously, $\mathcal{B}_a$ is a BSCC of $(\mathcal{D}_M^a)^{\eta}$. Next, we show that $\mathcal{B}_a$ is non-Zeno by contradiction. If $c \in loopCLK(\mathcal{B}_a)$, i.e., all states in $\mathcal{B}_a$ have clock c in their zone, we can conclude c never expires in $\mathcal{B}_a$. According to the definition of *probabilistic time-abstract simulation*, c should also never expires in $\mathcal{B}$, which conflicts with the fact that $\mathcal{B}$ is non-Zeno. Thus no c exists in $loopCLK(\mathcal{B}_a)$. Therefore $loopCLK(\mathcal{B}_a) = \emptyset$. Meanwhile, the existence of non-instantaneous transitions in $\mathcal{B}$ indicates not all transitions in $\mathcal{B}_a$ are instantaneous, which can be shown by the proof of Theorem 6.5.3. Thus $\mathcal{B}_a$ is non-Zeno. Therefore, all BSCCs in $(\mathcal{D}^a_M)^\eta$ are non-Zeno. So η is a non-Zeno scheduler of $(\mathcal{D}^a_M)^\eta$ according to Theorem 6.5.5. □

Next, we show how LTL-X properties are verified under the non-Zenoness assumption. After abstraction, a finite-state PA $\mathcal{D}^a_M$ is generated. Given an LTL-X formula ϕ, the automata-theoretic approach for probabilistic model checking [18] follows these steps: firstly, a deterministic Rabin automaton (DRA) equivalent to a given LTL-X formula is built. The product of the automaton and the abstract PA is then computed. Thirdly, MECs in the product which satisfy the Rabin acceptance condition are identified. Lastly, the probability of reaching any state of the MECs is calculated via a backward value iteration method, which equals the probability of satisfying the property. In the following, we extend this approach to handle non-Zenoness assumptions.

Given an abstract PA D generated from a PRTS model, we first define an MEC $\mathcal{M}\ (T, A)$ of $\mathcal{D}$ as *non-Zeno* if it satisfies the following conditions.

1. $loopCLK(T) = \emptyset$.
2. Not all transitions in $\mathcal{M}$ are instantaneous.

An MEC is *Zeno* if it is not *non-Zeno.* If a path reaches a Zeno MEC, then either there exists a clock which never expires on this path, or all transitions on this path after reaching this MEC are instantaneous. In both cases, infinite steps must happen within a finite time interval, therefore this path is Zeno. In the following, we refer to a state in a bottom Zeno MEC as a Zeno state.

Proposition 6.5.7 *Given a Zeno state s in PA $\mathcal{D}$, a scheduler σ satisfying $\sigma(s) \neq \emptyset$ is a Zeno scheduler.*

Proof Let Zeno MEC $\mathcal{M}\ (T, A)$ contain s. Because $\sigma(s) \neq \emptyset$, a BSCC $\mathcal{B}$ in $\mathcal{D}^\sigma$ can be generated from $\mathcal{M}$. Since $\mathcal{M}$ is Zeno, it satisfies $loopCLK(T) \neq \emptyset$ or all transitions in $\mathcal{M}$

Algorithm 4 Deciding Target MECs in PA

```
1: Let $\mathcal{V}, \mathcal{Z} = \emptyset$;
2: Let $C$ be the set of all MECs in $\mathcal{D}$;
3: for each $\mathcal{E} \in C$ do
4:     if $\mathcal{E}$ is non-Zeno then
5:         if $\mathcal{E}$ satisfies the Rabin acceptance condition then $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{E}$;
6:         end if
7:     else
8:         if $\mathcal{E}$ is bottom MEC then $\mathcal{Z} \leftarrow \mathcal{Z} \cup \mathcal{E}$;
9:         end if
10:    end if
11: end for
12: return $\mathcal{T}, \mathcal{Z}$;
```

are instantaneous. In both cases, $\mathcal{B}$ is guaranteed to be Zeno. Therefore $\mathcal{B}$ is also Zeno. According to Theorem 6.5.5, σ is a Zeno scheduler. □

Algo. 4 describes the modified algorithm used to decide the target states while labeling *Zeno states*. The algorithm takes a PA $\mathcal{D}$, which is the production of a PRTS model and a DRA representing an LTL-X property, as input and returns target states and $Zeno\ states$ of $\mathcal{D}$. Line 2 can be done via standard MEC searching algorithm in probabilistic verification [18]. Then for each MEC, we check whether it is Zeno or not. If it is non-Zeno, we check whether it satisfies the Rabin acceptance condition. If the answer is *yes*, then the whole MEC should be target states. Otherwise, we continue with next MEC. If it is Zeno as denoted by line 7, we check whether it is a bottom MEC. If there is no outgoing transitions from this MEC, we will add all its states into the set of *Zeno states*. MECs which are Zeno but not bottom are ignored in this algorithm.

Algo. 4 terminates since the number of MECs in a finite PA is also finite. States in $\mathcal{Z}$ are *Zeno states* according to the definition. Afterwards, it is critical to remove all Zeno schedulers in the given PA, which is done via Algo. 5.

In Algo. 5, we have a PA $\mathcal{D}$ $(\mathcal{S}, s_{init}, Act, Pr, AP, L)$ and its Zeno states $\mathcal{Z}$ as input. First, $\mathcal{V}$ representing visited states in $\mathcal{D}$ is set to be $\mathcal{Z}$. Starting from $\mathcal{Z}$, a backward search is executed. Line 3 indicates that $\mathcal{K}$ represents the pre-states of $\mathcal{Z}$. Then states in $\mathcal{K}$ are added to $\mathcal{V}$ (line 4). Afterwards, states in $\mathcal{K}$ are checked one by one to confirm which (α, μ) pairs lead them to $\mathcal{Z}$. All these pairs are removed since schedulers choosing them are Zeno schedulers, and Pr is updated. Lines 11 and 12 indicate that if the actions and corresponding distributions in one state are all removed, this state will be treated as Zeno

Algorithm 5 Removing Zeno Schedulers in PA

```
1: Let $\mathcal{V} = \mathcal{Z}$;
2: while $\mathcal{V} \neq \mathcal{S}$ do
3:     Let $\mathcal{K} = Pre(Z) - Z$;
4:     $\mathcal{V} \leftarrow \mathcal{V} \cup \mathcal{K}$;
5:     for each $s_k \in \mathcal{K}$ do
6:         for each $(\alpha, \mu) \in Act(s_k)$ do
7:             if $\exists\, s_z \in \mathcal{Z}, \mu(s_z) > 0$ then
8:                 $Pr \leftarrow Pr \backslash (s_k, \alpha, \mu)$;
9:             end if
10:        end for
11:        if $Act(s_k) = \emptyset$ then
12:            $\mathcal{Z} \leftarrow \mathcal{Z} \cup \{s_k\}$;
13:        end if
14:    end for
15: end while
16: $\mathcal{S} \leftarrow \mathcal{S} - \mathcal{Z}$;
17: return $(\mathcal{S}, s_{init}, Act, Pr, AP, L)$;
```

state and added in $\mathcal{Z}$. When the *while* loop terminates, i.e., all states in $\mathcal{S}$ are visited, states in $\mathcal{Z}$ are removed from $\mathcal{S}$, denoted as line 16. Afterwards, the reduced PA will be returned.

Because the state space of $\mathcal{D}$ is finite and the backward search eventually visits all states, the termination of Algo. 5 is obvious.

The worst-case complexity of Algo. 5 is $O(|\mathcal{S}| \times (|\mathcal{S}| + |\mathcal{T}|))$, where $\mathcal{T}$ represents all the transitions in $\mathcal{D}$. This can be seen as follows. The maximal number of iterations of the outermost loop is $|\mathcal{S}|$, as in each iteration at least one state is added to $\mathcal{V}$. In each iteration, the set of states that can reach $\mathcal{Z}$ needs to be computed. This takes $O(|\mathcal{S}| + |\mathcal{T}|)$ time, since each state s and transition t can be checked at most once.

The following theorem indicates Algo. 5 satisfies our requirement, i.e., all Zeno schedulers in $\mathcal{D}$ are removed.

Theorem 6.5.8 *Given a PA $\mathcal{D}$, a scheduler σ of $\mathcal{D}$ is removed by Algo. 5 iff it is Zeno.*

Proof (If) Let σ be a Zeno scheduler of $\mathcal{D}$, then $\mathcal{D}^\sigma$ must have Zeno paths reaching Zeno states. The backward search used in Algo. 5 (lines 4-11) rules out all paths reaching $\mathcal{Z}$ by removing undesired actions and distributions, thus corresponding schedulers are also removed. Therefore σ must be removed from $\mathcal{D}$.

(Only-If) According to Algo. 5, (α, μ) is removed from one state s if and only if s can reach $\mathcal{Z}$ through (α, μ). As a result, a scheduler σ of $\mathcal{D}$ satisfying $\sigma(s) = (\alpha, \mu)$ makes DTMC $\mathcal{D}^\sigma$ reaching $\mathcal{Z}$ with positive probability. Meanwhile, the paths reaching $\mathcal{Z}$ are Zeno paths according to the definition of Zeno states. Therefore, σ is a Zeno scheduler. We conclude that removed actions and corresponding distributions in Algo. 5 must belong to some Zeno schedulers. Therefore a removed scheduler must be Zeno.

In summary, the theorem holds. □

After removing all Zeno schedulers, traditional value iteration approach can be used in the reduced PA to calculate the maximal probability of reaching target states $\mathcal{T}$ from the initial state.

6.6 Implementation and Evaluation

We have implemented the proposed language and algorithms in a PRTS model checker to support system modeling and automatic verification of systems based on PRTS. This tool has been integrated into PAT model checking framework as an independent module. PAT is a self-contained environment for system modeling, simulation and verification. Properties such as reachability checking and LTL checking are supported in PAT. A system with several millions of states can be verified efficiently, and the average verification speed is around 10K states per second (or equivalently millions in one hour).

In the following, we use three different experiments to demonstrate the efficiency of our method and implementation. Note that for experimental results of probabilistic real-time systems, our approach generates over-approximation of accurate results. The experiment testbed is a PC running Windows Server 2008 64 Bit with Intel Xeon 4-Core CPU×2 and 32 GB memory. "-" in following tables indicates the corresponding experiments cannot be finished within 1 hour. Our tool can be downloaded at `http://www.patroot.com`.

6.6.1 Verification Under Non-Zenoness Assumption

In this part, we use two examples to demonstrate the effects of non-Zenoness assumptions in PRTS verification; next, the multi-lift model is used to show the efficiency of PRTS in real-life systems. Note there is no comparison with other model checkers in this part, because

System	States	Results	
		PAT without NZ	PAT with NZ
FDDI(3)	5084	1.0	0.0
TTA(8)	2533	1.0	0.0

Table 6.1: The Affect of Zeno Schedulers

as far as we know, our work is the only one which supports verification of probabilistic real-time models with non-Zenoness assumptions.

First, two benchmark systems affected by Zeno schedulers are evaluated: Fiber Distributed Data Interface (FDDI) [42] and Time Triggered Architecture (TTA) [75]. FDDI is composed by several identical stations and a ring, where the stations can communicate by synchronous messages through channels. Meanwhile, a TTA system is composed of host computers (the nodes) connected over a shared bus with time-division multiple-access (TDMA) strategy. Desired LTL-X properties are checked in these two systems. The experimental results are listed in table 6.1. The parameters in FDDI and TTA (3 and 8) represent the number of components or communication channels in these models respectively.

These two systems are verified with/without non-Zenoness (NZ) assumption. From the table, we have the following conclusions. First, Zeno schedulers could affect the verification results. Take FDDI for example. There exists an MEC in the system where the communication occurs without consuming time, i.e., all transitions are instantaneous in that MEC. However, this MEC satisfies the Rabin condition of the desired LTL-X property, so it existence affects the verification result. Second, non-Zeno assumption has no affect on the number of states evolved in the verification. Here the number of states listed are the states in the product PA, which is generated from the abstract PA representing the model and the automata representing the property.

Next, we verify the lift model and compare two strategies of assigning external requests. One is to assign the request to a *random* lift. The other is that an external request is always assigned to the 'nearest' lift. For simplicity, we assume external requests are never re-assigned. A lift works as follows. It firstly checks whether it should serve the current floor. If positive, it opens the door and then repeats from the beginning later. If negative, it checks whether it should continue traveling in the same direction (if there are internal requests or assigned external requests on the way) or change direction (if there are internal or assigned external requests on the other direction) or simply idle (otherwise). Note that

System	Random Assign			Nearest Assign		
	Result	Time (s)	Time-NZ (s)	Result	Time (s)	Time-NZ (s)
lift=2; floor=2; user=2	0.21875	1.262	1.517	0.13889	0.685	0.755
lift=2; floor=2; user=3	0.47656	15.918	31.002	0.34722	7.317	10.305
lift=2; floor=2; user=4	0.6792	98.836	217.145	0.53781	36.077	84.114
lift=2; floor=2; user=5	0.81372	407.023	1306.378	0.68403	102.301	250.125
lift=2; floor=3; user=2	0.2551	12.172	15.205	0.18	6.757	10.112
lift=2; floor=3; user=3	0.54009	109.203	264.588	0.427	43.865	89.810
lift=2; floor=4; user=2	0.27	11.406	15.845	0.19898	6.693	10.524
lift=3; floor=2; user=2	0.22917	47.499	112.361	0.10938	22.425	36.451
lift=3; floor=2; user=3	-	-	-	0.27344	1493.969	-

Table 6.2: Multi-lift Systems

it is constrained (using *within*) to react regularly. The property that a lift should not pass by without serving a user's external request is verified through a probabilistic reachability analysis problem (or equivalently a simple LTL-X formula). That is, what is the maximal probability of reaching a state in which a lift is passing by a floor where a user has requested to travel in the same direction. Verifications with and without non-Zenoness assumption are both conducted. Table 6.2 summarizes the experiment results.

In table 6.2, the parameters of the system denote the number of *lifts*, the number of *floors* and number of *user requests* respectively. We limit the number of user requests so as to check how the probability varies as well as to avoid state space explosion. Columns *Random Assign* and *Nearest Assign* demonstrate the two different strategies of assigning requests. Column *Result* shows the *maximum* probability of violating the desired property in each assigning strategy. Note that it can be shown that the minimum probability is always 0 (i.e., there exists a scheduler which guarantees satisfaction of the property). *Time* and *Time-NZ* represent the verification time cost without/with non-Zenoness assumption. We remark that in these experiments, non-Zenoness assumption has no affect of the verification results.

The following conclusions can be made. First, it takes at least two external requests, two lifts and two floors to constitute a bad behavior, e.g., one lift is at top floor (and later going down to serve a request), while a request for going down at the top floor is assigned to the other lift. Second, the more user requests, the higher the probability is. Intuitively, this means that with more requests, it is more likely that a request is ignored. Similarly, the probability is higher with more floors. Third, assigning requests to the 'nearest' lift performs better than random assignment, i.e., the maximum probability of exhibiting a bad behavior with the former is always lower than with the latter in all cases. Lastly,

System	Result	PAT		PRISM	
		States	Time(s)	States	Time(s)
ZC(100)	0.49934	404	**0.15**	135	0.45
ZC(300)	0.01291	4813	**0.65**	1499	0.73
ZC(500)	0.00027	12840	2.39	4067	**1.19**
ZC(700)	1E-5	24058	5.78	7655	**1.70**
FA(10K)	1	1352	**0.15**	525	0.47
FA(20K)	1	5030	**0.13**	1875	1.09
FA(30K)	1	9365	**0.5**	4048	1.67
FA(300K)	1	726407	**30.74**	-	-

Table 6.3: Benchmark Probabilistic Real-time Systems

non-Zenoness assumption consumes more time during verification. This is because of the overhead of finding MECs in reachability checking, especially when the state space of the model increases.

6.6.2 Probabilistic Real-time Benchmark Systems

Next, we compare our model checker with PRISM on verifying benchmark systems of probabilistic real-time system. Here we use two PTA models described in [78]. Besides the *zeroconf* (*ZC*) protocol, we pick the *firewire abstraction* (*FA*), which is used for IEEE 1394 FireWire root contention protocol. We build PTA models using PRISM and PRTS models using PAT, and verify the desired reachability properties to check the efficiency of these two tools. Note that PRISM supports refinement of an abstract PTA model to generate accurate verification results [78], while in our experiments we disabled its refinement to better compare the efficiency of these tools. Meanwhile, non-Zenoness assumption is not enabled in these experiments for fair comparison. The experimental results are listed in Table 6.3, and more efficient results between these two tools are highlighted in **bold**.

For *ZC*, we check the maximal probability of the device getting an IP (might be used or fresh) **after** a specific deadline. *ZC(d)* in Table 6.3 means the deadline is d time units. For *FA*, we check the maximal probability of successfully choosing a leader before a specified deadline, and the parameter d in *FA(d)* also means the deadline is d time units. Through the experimental results, we can see that PRTS and PRISM outperform each other in different cases. The number of states are different in both tools because of the different language design features. For *ZC*, when the timed constraint is small (100 and 300), PRTS has a better performance since the number of states are relatively small. As d increases, PRISM is better since the number of abstract states is increased with d and the symbolic engine in PRISM is efficient. For *FA*, PRTS always has better performance. This is mainly because the state space difference between models of these two tools is not as large as *ZC*. Further, when d increases to some extend (300K), the efficiency of PRISM to build the abstract model is quite low, which cannot be finished within 1 hour. These examples are sufficient to show the differences between these two tools in verifying probabilistic real-time systems. We remark that refinement from the above approximation results to get the accurate probability is one of our future work, and the digitization of real-time requirements such as the work [81] is also a potential method.

6.7 Related Work

There are several modeling methods and model checking algorithms for real-time probabilistic systems. Alur, Courcoubetis and Dill presented a model-checking algorithm for probabilistic real-time systems to verify TCTL formulae of probabilistic real-time systems [9]. Their specification is limited to deterministic Timed Automata, and its use of continuous probability distributions (a highly expressive modeling mechanism) does not permit the model to be automatically verified against logics which include bounds on probability. Remotely related is the line of work on Continuous-Time Markov Chains (CTMC) [17]. Different from CTMC, our work is based on discrete probability distributions. A method for analyzing the stochastic and timing properties of systems was proposed in [16]. It is based on MTBDD. Properties are expressed in a subset of PCTL. The method was not based on real-time but in the realm of discrete time. Similar work using discrete time includes [51, 81].

Research on combining quantitative timing and probability has been mostly based on Probabilistic Timed Automata (PTA) [57, 82]. PTA extends Timed Automata [10] with discrete probability distributions which are defined over a finite set of edges. It is a modeling formalism for describing formally both nondeterministic and probabilistic aspects of real-time systems. Based on PTA, symbolic verification techniques [131, 83] are developed using MTBDDs. In [22], Beauquier proposes another model of probabilistic Timed Automata. The model in [22] differs from PTA in that it allows different enabling conditions for edges related to a certain action and uses Büchi conditions as accepting conditions. In [79], probabilistic timed program (PTP) is proposed to model real-time probabilistic software (e.g., SystemC). PTP is an extension of PTA with discrete variables. PTA and PTP are closely related to PRTS. Different from PRTS, models based on PTA or PTP often have a simple structure, e.g., a network of automata with no hierarchy.

Research on verification with non-Zenoness assumption is mainly based on Timed Automata (TA). Syntactic conditions for TA to be free from Zeno runs have been identified [124, 56]. The conditions are often sufficient only [28]. In the setting of TA, it has been shown that it is not possible to determine if a run can be instantiated to a non-Zeno run given only zone graphs. The solution involving adding one extra clock has been discussed in [124, 126, 125]. Recently, it has been shown that adding one clock may result in an exponentially larger zone graph [66, 65]. The remedy is to transform the zone graph into a guess zone graph and require that all clocks that are bounded from above must be reset infinitely often during a run and the run must visit a state such that the clocks can be strictly positive [66]. In our work, probabilistic systems are taken into consideration, and all clocks are bounded and cannot be reset arbitrarily. As a result, detecting Zeno runs based on zone graphs is feasible and efficient.

Verification support for real-time probabilistic systems often uses a combined approach, i.e., combination of real-time verifiers with probabilistic verifiers [41]. Our approach is a combination of real-time zone abstraction with PA, which has no extra cost of linking different model checkers. This work is related to our previous works [116, 119], and the significant improvements here are: 1) we seamlessly combine all the features of these two papers to get a new language to model hierarchical probabilistic real-time systems and 2) the affect of dynamic zone abstraction is investigated to make the verification results reliable. To our best knowledge, our implementation is the first tool to support non-Zeno assumption in probabilistic real-time systems.

6.8 Conclusion

In this chapter we novelly propose a modeling language PRTS which is capable of specifying hierarchical complex systems with quantitative real-time features as well as probabilistic components. In order to apply model checking techniques, we use dynamic zone abstraction technique to reduce the state space of the model to be finite. An upper bound of the accurate maximal probability is generated during this abstraction. Meanwhile, model checking with non-Zenoness assumption in PRTS is discussed to rule out unrealistic results. In addition, we have extended our PAT model checker to support this kind of systems so that the techniques are easily accessible. Finally, several experiments are displayed to indicate the efficiency of PAT.

Chapter 7

Conclusion and Future Work

In this chapter, we briefly summarize the contributions of this thesis and discuss possible future directions of our work.

7.1 Summary

In this thesis, we systematically investigate the probabilistic model checking problem with PAT.

First, we have proposed a modeling language called PCSP# for hierarchical complex stochastic systems. PCSP# extends CSP with shared variables, user-defined data structures and probabilistic choices. It combines low-level programs with high-level specifications. The semantic model of PCSP# is Probabilistic Automata, which supports full non-determinism in modeling concurrent and probabilistic models. Several popular properties are supported in PCSP# verification, including deadlock checking, reachability checking and LTL checking. Besides, we have defined the trace refinement relation between a probabilistic model and a non-probabilistic specification. In order to increase the efficiency of LTL checking, we have developed an optimized approach based on safety recognition of LTL formulae. Therefore the verification of safety LTL properties can be transferred to refinement checking in our setting. In addition, we have proposed an anti-chain based approach to speed up the probabilistic refinement checking in PCSP#.

Next, we have applied model checking techniques in multi-agent systems to analyze their behaviors. In some complicated MAS models, mathematical deduction and simulation are not suitable, which makes formal verification as an alternative approach. We have investigated two representing scenarios: *robustness* of negotiation strategies, and dynamics of *dispersion games* [120]. A symmetry reduction technique: counter abstraction is applied to reduce the state space of the models. In order to analyze specific properties in MAS, we have proposed dedicated verification algorithms based on existing approaches.

In dispersion game analysis, DTMC is the actual semantic model of the corresponding PCSP# models. In DTMC verification, reachability analysis plays the key role, because verification of other properties, such as LTL and PCTL, can be transferred to reachability problem. Therefore, we have proposed a divide and conquer approach to eliminate loops in DTMC, which aims at increasing the efficiency of the traditional value iteration method. Several related strategies have been designed to make this approach feasible.

Last but not least, in order to capture the timed constraints in various real life systems, we have designed another modeling language named PRTS. Based on PCSP#, PRTS supports hierarchy, concurrency, real-time and probability. Different from PCSP# models, PRTS models have dense-time semantics, which generates PA with infinite number of states. To tackle this issue, dynamic zone abstraction has been used to make the state space of PRTS models finite. We have also proven that the abstract model has over-approximation verification results compared with the concrete model. Furthermore, we have taken non-Zenoness assumption into our consideration to rule out unrealistic verification results.

We have conducted various experiments to demonstrate the effectiveness and efficiency of our approaches, including real life cases, benchmark systems and manually-built models. All proposed languages and algorithms have been implemented in PAT model checking framework, which can be freely downloaded on our website `www.patroot.com`.

7.2 Future Work

Although we have achieved the above contributions, there are still several future extensions of our work, listed as follows.

- In Chapter 3, we have supported the probabilistic refinement checking, which is between a probabilistic model and an LTS specification. However, as shown in previous work such as [94], the refinement relation between different probabilistic processes is also important. Therefore, we will explore methods for checking refinement relationship between probabilistic models.

- In Chapter 4, we have shown that model checking techniques are applicable in analyzing some specific multi-agent systems. Two interesting directions of related future work exist. 1) It is worthy to investigate other systems and scenarios, which are more general and meaningful to MAS community. For example, instead of the single-agent best deviation considered in our work, multiple-agent deviation is more critical and complex. Suitable modeling and verification for this scenario are in demand. 2) More state-space reduction techniques are needed in order to increase the efficiency of verification, such as symmetry reduction and partial-order reduction, or even some domain-specific abstraction techniques according to the characteristics of MAS.

- In Chapter 5, the divide-and-conquer approach is used to speed up the reachability analysis of DTMC. However, currently parameters used in the algorithm such as B, B_L and B_U are mainly decided via experience, and are manually set before the experiments. Different systems may need different parameters to have the best performance. Therefore, one potential topic is to find the more efficient division strategies, which are automatic and suitable for general cases. In addition, besides DTMC, PA is also a popular probabilistic formalism. Therefore, another direction is extending our approach to PA. Concurrency also exists in many probabilistic systems, therefore the non-determinism is unavoidable in some cases. MECs in PA can also be eliminated via calculating the probability distributions from inputs to outputs. Due to the nondeterminism in PA, one of the challenges is that the number of resulting distributions may be exponential, so a suitable divide and conquer approach for PA is important.

- In Chapter 6, PRTS is proposed for modeling and verification of probabilistic real-time systems. However, dynamic zone abstraction can just generate over-approximation of the accurate results. Therefore, suitable refinement methods to generate accurate results in PRTS verification is worthy to be explored. Furthermore, currently we just take un-timed properties into consideration. In real life systems, timed properties are also critical. For example, in communication protocols, people may be interested in the property *"what is the probability that one message can be delivered successfully within 1 minute?"* It remains one of our future work to investigate the verification of timed properties in PRTS models.

In a nutshell, our future work includes supporting more widely used properties, designing more efficient verification algorithm, and investigating more applications of our approach.

Bibliography

[1] PRTS Model Checker. `http://www.comp.nus.edu.sg/~pat/cav12prts`. 6.3.1

[2] *The Second International Automated Negotiating Agent Competition (ANAC 2011)*, 2011. `http://www.itolab.nitech.ac.jp/ANAC2011/`. 4.2.2

[3] *The Third International Automated Negotiating Agent Competition (ANAC 2012)*, 2012. `http://anac2012.ecs.soton.ac.uk/`. 4.2.1, 4.2.2, 4.5.1

[4] P. A. Abdulla, Y.-F. Chen, L. Holk, R. Mayr, and T. Vojnar. When simulation meets antichains. In *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 158–174. Springer, 2010. 3.2.3, 3.7

[5] E. Ábrahám, N. Jansen, R. Wimmer, J.-P. Katoen, and B. Becker. DTMC Model Checking by SCC Reduction. In *QEST*, pages 37–46, 2010. 5.1, 5.2.1, 5.2.3, 2, 5.3, 5.3.3, 5.5

[6] B. Alpern and F. B. Schneider. Recognizing Safety and Liveness. *Distributed Computing*, 2(3):117–126, 1987. 3.2.2, 3.4.2, 3.7

[7] S. Alpern. Spatial dispersion as a dynamic coordination problem. Technical report, The London School of Economics", 2001. 4.1, 4.2.3, 4.4.2

[8] S. C. Althoen and R. McLaughlin. Gauss - Jordan reduction: a brief history. In *The American Mathematical Monthly*, volume 94(2), pages 130–142, 1987. 5.1, 5.2.3, 5.2.3

[9] R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking for Probabilistic Real-time Systems. In *ICALP*, pages 115–126, 1991. 6.7

[10] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994. 6.3.1, 6.7

[11] R. Alur and T. A. Henzinger. Reactive Modules. *Formal Methods in System Design*, 15(1):7–48, 1999. 1, 3.1, 3.7

[12] M. E. Andrés, P. R. D'Argenio, and P. V. Rossum. Significant Diagnostic Counterexamples in Probabilistic Model Checking. In *Haifa Verification Conference*, pages 129–148, 2008. 5.1, 5.2.1, 5.3, 5.5

[13] J. Aspnes and M. Herlihy. Fast Randomized Consensus Using Shared Memory. *Journal of Algorithms*, 15(1):441–460, 1990. 3.6, 5.4

[14] T. Baarslag, K. Fujita, E. H. Gerding, K. Hindriks, T. Ito, N. R. Jennings, C. Jonker, S. Kraus, R. Lin, V. Robu, and C. R. Williams. Evaluating practical negotiating agents: Results and analysis of the 2011 international competition. *Artificial Intelligence Journal*, To appear. 4.1, 4.1, 4.2.1, 4.2.2, 4.2.2

[15] T. Baarslag, K. Hindriks, C. Jonker, S. Kraus, and R. Lin. The first automated negotiating agents competition (anac 2010). *New Trends in Agent-Based Complex Automated Negotiations*, pages 113–135, 2010. 4.1, 4.2.1

[16] C. Baier, E. M. Clarke, V. H. Garmhausen, M. Z. Kwiatkowska, and M. Ryan. Symbolic Model Checking for Probabilistic Processes. In *ICALP*, pages 430–440, 1997. 6.7

[17] C. Baier, B. R. Haverkort, H. Hermanns, and J. Katoen. Model-Checking Algorithms for Continuous-Time Markov Chains. *IEEE Trans. Software Eng.*, 29(6):524–541, 2003. 6.7

[18] C. Baier and J. Katoen. *Principles of Model Checking*. The MIT Press, 2008. 1, 2.1.1, 2.1.2, 2.1.2, 2.3.1, 2.3.1, 2.3.2, 1, 3.1, 3.4.1, 3.4.1, 3.4.2, 4.4.2, 5.1, 5.5, 6.2, 6.5.3, 6.5.3

[19] P. Ballarini, M. Fisher, and M. Wooldridge. Uncertain agent verification through probabilistic model-checking. In *SASEMAS'09*, Lecture Notes in Computer Science, pages 162–174, 2009. 4.6

[20] S. S. Barold, R. X. Stroopbandt, and A. F. Sinnaeve. *Cardiac Pacemakers Step by Step: an Illustrated Guide*. Blachwell Publishing, 2004. 3.3.1, 6.1

[21] B.Arthur. Inductive reasoning and bounded rationality. *American Economic Association Papers*, 84:406–411, 1994. 4.1

[22] D. Beauquier. On Probabilistic Timed Automata. *Theor. Comput. Sci.*, 292(1):65–84, 2003. 6.7

[23] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks. UPPAAL 4.0. In *QEST*, pages 125–126. IEEE, 2006. 1, 6.1

[24] G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and W. Yi. Efficient Timed Reachability Analysis Using Clock Difference Diagrams. In *CAV*, pages 341–353, 1999. 6.4

[25] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Verifying multi-agent programs by model checking. *AAMAS*, 12:239–256, 2006. 4.6

[26] A. Bouajjani, P. Habermehl, L. Holk, T. Touili, and T. Vojnar. Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In *CIAA*, volume 5148 of *Lecture Notes in Computer Science*, pages 57–67. Springer, 2008. 3.7

[27] B. Bouzy and M. Métivier. Multi-agent learning experiments on repeated matrix games. In *ICML*, pages 119–126, 2010. 4.5.1.1

[28] H. Bowman and R. Gómez. How to Stop Time Stopping. *Formal Aspects of Computing*, 18(4):459–493, 2006. 6.7

[29] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Inf. Comput.*, 98(2):142–170, 1992. 3.6.2

[30] A. Butterfield, A. Sherif, and J. Woodcock. Slotted-Circus. In *IFM*, pages 75–97, 2007. 6.1

[31] S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/Event-Based Software Model Checking. In *IFM*, volume 2999 of *LNCS*, pages 128–147. Springer, 2004. 2.2

[32] K. Chatterjee, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Algorithms for omega-regular games with imperfect information. In *CSL*, volume 4207 of *Lecture Notes in Computer Science*, pages 287–302. Springer, 2006. 3.7

[33] Y. Chen and J. W. Sanders. Unifying Probability with Nondeterminism. In *FM*, volume 5850 of *LNCS*, pages 467–482. Springer, 2009. 3.7

[34] S. Cheshire, B. Adoba, and E. Gutterman. Dynamic configuration of IPv4 link local addresses. Available from http://www.ietf.org/rfc/rfc3927.txt. 6.1

[35] F. Ciesinski and C. Baier. LiQuor: A Tool for Qualitative and Quantitative Linear Time Analysis of Reactive Systems. In *QEST*, pages 131–132. IEEE Computer Society, 2006. 1, 3.1, 3.7

[36] F. Ciesinski, C. Baier, M. Größer, and J. Klein. Reduction Techniques for Model Checking Markov Decision Processes. In *QEST*, pages 45–54, 2008. 5.1, 5.3.3, 5.5

[37] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999. 1, 6.1, 6.2

[38] C. Claus and C. Boutilier. The dynamics of reinforcement learning in cooperative multiagent systems. In *AAAI'98*, pages 746–752, 1998. 4.1, 4.1

[39] A. David, K. G. Larsen, A. Legay, M. Mikucionis, and Z. Wang. Time for statistical model checking of real-time systems. In *CAV*, pages 349–355, 2011. 6.1

[40] J. Davies. *Specification and Proof in Real-Time CSP*. Cambridge University Press, 1993. 6.3.1

[41] C. Daws, M. Kwiatkowska, and G. Norman. Automatic Verification of the IEEE 1394 Root Contention Protocol with KRONOS and PRISM. *International Journal on Software Tools for Technology Transfer*, 5(2-3):221–236, 2004. 6.7

[42] C. Daws and S. Tripakis. Model checking of real-time reachability properties using abstractions. In *TACAS*, pages 313–329, 1998. 6.6.1

[43] D.Challet and Y.Zhang. Emergence of cooperation and organization in an evolutionary game. *Physica A*, 246:407, 1994. 4.1

[44] D. L. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In *Automatic Verification Methods for Finite State Systems*, pages 197–212, 1989. 6.1, 6.4, 6.5.1

[45] L. Doyen and J. F. Raskin. Antichains for the automata-based approach to model checking. *Logical Methods in Computer Science*, 5(1:5):1–20, 2009. 3.7

[46] O. Etzioni. Moving up the information food chain: Deploying softbots on the world wide web. In *AI Magazine*, pages 1322–1326, 1996. 4.1

[47] P. Faratin, C. Sierra, and N. R. Jennings. Using similarity criteria to make negotiation trade-offs. *Artifical Intelligence*, 142(2):205–237, 2003. 4.1

[48] A. Fehnker and P. Gao. Formal verification and simulation for performance analysis for probabilistic broadcast protocols. In *Proc. 5th International Conference on Ad-Hoc, Mobile, and Wireless Networks (ADHOC-NOW'06)*, volume 4104 of *LNCS*, pages 128–141. Springer, 2006. 1

[49] E. Filiot, N. Jin, and J.-F. Raskin. An antichain algorithm for ltl realizability. In *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 2009. 3.7

[50] M. Fruth. *Formal Methods for the Analysis of Wireless Network Protocols*. PhD thesis, Oxford University, 2011. 1

[51] V. H. Garmhausen, S. V. A. Campos, and E. M. Clarke. ProbVerus: Probabilistic Symbolic Model Checking. In *ARTS*, pages 96–110, 1999. 6.7

[52] P. Gastin and D. Oddoux. Fast LTL to Büchi Automata Translation. In *CAV*, volume 2102 of *LNCS*, pages 53–65. Springer, 2001. 3.2.2, 3.4.2

[53] M. Geilen. On the Construction of Monitors for Temporal Logic Properties. *Electr. Notes Theor. Comput. Sci.*, 55(2), 2001. 3.7

[54] G.Holzmann. The spin model checker. *TSE*, 23(5):279–295, 1997. 4.6

[55] E. R. Gomes and R.Kowalczyk. Dynamic analysis of multiagent -learning with e-greedy exploration. In *ICML'09*, 2009. 4.1, 4.1

[56] R. Gómez and H. Bowman. Efficient Detection of Zeno Runs in Timed Automata. In *5th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, volume 4763 of *Lecture Notes in Computer Science*, pages 195–210. Springer, 2007. 6.7

[57] H. Gregersen and H. E. Jensen. *Formal Design of Reliable Real Time Systems*. PhD thesis, 1995. 6.7

[58] T. Grenager, R. Powers, and Y. Shoham. Dispersion Games: General Definitions and Some Specific Learning Results. In *AAAI*, pages 398–403, 2002. 5.4

[59] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6:102–111, 1994. 4.6

[60] J. Hao, S. Song, Y. Liu, J. Sun, L. Gui, J. S. Dong, and H. fung Leung. Probabilistic model checking multi-agent behaviors in dispersion games using counter abstraction. In *PRIMA*, pages 16–30, 2012. 1.3

[61] J. Y. Hao and H. F. Leung. Abines: An adaptive bilateral negotiating strategy over multiple items. In *Proceedings of IAT'12*, 2012. 4.5.1

[62] K. Havelund and G. Rosu. Synthesizing Monitors for Safety Properties. In *TACAS*, volume 2280 of *LNCS*, pages 342–356. Springer, 2002. 3.7

[63] J. Heath, M. Kwiatkowska, G. Norman, D. Parker, and O. Tymchyshyn. Probabilistic model checking of complex biological pathways. In *Proc. Computational Methods in Systems Biology (CMSB'06)*, pages 32–47, 2006. 1

[64] J. Heath, M. Kwiatkowska, G. Norman, D. Parker, and O. Tymchyshyn. Probabilistic model checking of complex biological pathways. *Theoretical Computer Science*, 319(3):239–257, 2008. 1

[65] F. Herbreteau and B. Srivathsan. Efficient On-The-Fly Emptiness Check for Timed Büchi Automata. In *8th International Symposium on Automated Technology for Verification and Analysis (AVTA)*, Lecture Notes in Computer Science. Springer, 2010. 6.7

[66] F. Herbreteau, B. Srivathsan, and I. Walukiewicz. Efficient Emptiness Check for Timed Büchi Automata. In *22nd International Conference on Computer Aided Verification (CAV)*, volume 6174 of *Lecture Notes in Computer Science*, pages 148–161. Springer, 2010. 6.7

[67] K. Hindriks and D. Tykhonov. Opponent modeling in auomated multi-issue negotiation using bayesian learning. In *AAMAS'08*, pages 331–338, 2008. 4.1

[68] A. Hinton, M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM: A Tool for Automatic Verification of Probabilistic Systems. In *TACAS*, pages 441–444, 2006. 4.6

[69] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. 1.1, 3.1, 3.4, 6.1

[70] G. J. Holzmann. The Model Checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997. 6.1

[71] A. Itai and M. Rodeh. Symmetry Breaking in Distributed Networks. *Information and Computation*, 88:150–158, 1981. 1

[72] J. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen. The Ins and Outs of the Probabilistic Model Checker MRMC. In *QEST*, pages 167–176. IEEE Computer Society, 2009. 1, 3.1, 3.7

[73] J.-P. Katoen, M. Khattri, and I. S. Zapreev. A Markov Reward Model Checker. In *QEST*, pages 243–244, 2005. 5.1

[74] J.-P. Katoen, I. S. Zapreev, E. M. Hahn, H. H., and D. N. Jansen. The Ins and Outs of The Probabilistic Model Checker MRMC. In *QEST*, pages 167–176, 2009. 5.1

[75] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003. 6.6.1

[76] O. Kupferman and M. Y. Vardi. Model Checking of Safety Properties. *Formal Methods in System Design*, 19(3):291–314, 2001. 3.2.2, 3.7

[77] M. Kwiatkowska, G. Norman, and D. Parker. Using probabilistic model checking in systems biology. *ACM SIGMETRICS Performance Evaluation Review*, 35(4):14–21, 2008. 1

[78] M. Kwiatkowska, G. Norman, and D. Parker. Stochastic games for verification of probabilistic timed automata. In *FORMATS*, volume 5813 of *LNCS*, pages 212–227, 2009. 6.6.2

[79] M. Kwiatkowska, G. Norman, and D. Parker. A Framework for Verification of Software with Time and Probabilities. In *FORMATS*, LNCS. Springer, 2010. To appear. 6.7

[80] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *CAV*, volume 6806, pages 585–591, 2011. 1, 3.1, 3.6, 3.7, 5.1, 6.1

[81] M. Kwiatkowska, G. Norman, D. Parker, and J. Sproston. Performance Analysis of Probabilistic Timed Automata using Digital Clocks. *FMSD*, 29:33–78, 2006. 6.6.2, 6.7

[82] M. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Automatic Verification of Real-time Systems with Discrete Probability Distributions. *Theoretical Computer Science*, 282(1):101–150, 2002. 6.1, 6.2.3, 6.3.1, 6.5.2, 6.7

[83] M. Kwiatkowska, G. Norman, J. Sproston, and F. Wang. Symbolic Model Checking for Probabilistic Timed Automata. *Information and Computation*, 205(7):1027–1077, 2007. 1, 6.7

[84] M. Z. Kwiatkowska, G. Norman, and D. Parker. Symmetry reduction for probabilistic model checking. In *CAV*, pages 234–248, 2006. 4.1

[85] M. Z. Kwiatkowska, D. Parker, and H. Qu. Incremental Quantitative Verification for Markov Decision Processes. In *DSN*, pages 359–370, 2011. 5.1, 5.3.3, 5.5

[86] T. Latvala. Efficient Model Checking of Safety Properties. In *SPIN*, volume 2648 of *LNCS*, pages 74–88. Springer, 2003. 3.2.2, 3.7

[87] D. Lehmann and M. Rabin. On the Advantage of Free Choice: A Symmetric and Fully Distributed Solution to the Dining Philosophers Problem (Extended Abstract). In *POPL*, pages 133–138. ACM, 1981. 3.6

[88] Y. Liu, W. Chen, Y. A. Liu, and J. Sun. Model checking linearizability via refinement. In *FM*, pages 321–337, 2009. 2.4

[89] Y. Liu, W. Chen, Y. A. Liu, J. Sun, S. J. Zhang, and J. S. Dong. Verifying linearizability via optimized refinement checking. *IEEE Trans. Software Eng.*, 39(7):1018–1039, 2013. 2.4

[90] J. E. M. P. Wellman, S. Singh, Y. Vorbeychik, and V. Soni. Strategic interactions in a supply chain game. *Computational Intelligence*, 21(1):1–26, 2005. 4.2.2

[91] B. P. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: An Introduction to TCOZ. In *ICSE*, pages 95–104, 1998. 6.1

[92] W. H. Maisel, M. Moynahan, B. D. Zuckerman, T. P. Gross, O. H. Tovar, D. Tillman, and D. B. Schultz. Pacemaker and ICD Generator Malfunctions. *The Journal of American Medical Association*, 295(16):1901–1906, 2006. 3.3.1

[93] C. Morgan, T. S. Hoang, and J. Abrial. The Challenge of Probabilistic *Event B* - Extended Abstract. In *ZB*, volume 3455 of *LNCS*, pages 162–171. Springer, 2005. 3.7

[94] C. Morgan, A. McIver, K. Seidel, and J. W. Sanders. Refinement-Oriented Probability for CSP. *Formal Asp. Comput.*, 8(6):617–647, 1996. 1.1, 3.1, 3.3.1, 3.7, 7.2

[95] X. Nicollin, J. Sifakis, and S. Yovine. Compiling Real-time Specifications into Extended Automata. *IEEE Transactions on Software Engineering*, 18(9):794–804, 1992. 3.6

[96] J. Ouaknine and J. Worrell. Timed CSP = Closed Timed Safety Automata. *Electrical Notes Theoretical Computer Science*, 68(2), 2002. 6.4

[97] M. V. P. Stone. Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, 8:345–383, 2000. 4.1

[98] A. Pnueli. The Temporal Logic of Programs. In *FOCS*, pages 46–57. IEEE, 1977. 2.2

[99] A. Pnueli, J. Xu, and L. Zuck. Liveness with (0,1,∞)-counter abstraction. In *CAV'02*, pages 107–122, 2002. 4.1

[100] A. Pnueli and L. Zuck. Verification of Multiprocess Probabilistic Protocols. *Distributed Computing*, 1(1):53–72, 1986. 3.6

[101] J. W. Pratt. Risk aversion in the small and in the large. *Econometrica*, 32:122–136, 1964. 4.2.2

[102] A. S. Rao. Agentspeak(l): Bdi agents speak out in a logical computable language. In *MAAMAW'96*, pages 42–55, 1996. 4.6

[103] T. G. Rokichi. *Representing and Modeling Digital Circuits*. PhD thesis, 1993. 6.4

[104] A. W. Roscoe. Model-checking CSP. pages 353–378, 1994. 3.1, 3.2.1, 3.2.1, 3.4.1

[105] A. W. Roscoe, P. H. B. Gardiner, M. Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical Compression for Model-Checking CSP or How to Check 10^{20} Dining Philosophers for Deadlock. In *TACAS*, pages 133–152, 1995. 3.1, 3.2.1

[106] S. Saha, A. Biswas, and S. Sen. Modeling opponent decision in repeated one-shot negotiations. In *AAMAS'05*, pages 397–403, 2005. 4.1

[107] S. Schneider. *Concurrent and Real-time Systems*. John Wiley and Sons, 2000. 6.3.1, 6.4

[108] A. P. Sistla. Safety, Liveness and Fairness in Temporal Logic. *Formal Asp. Comput.*, 6(5):495–512, 1994. 3.2.2, 3.4.2, 3.7

[109] F. Somenzi and R. Bloem. Efficient Büchi Automata from LTL Formulae. In *CAV*, volume 1855 of *LNCS*, pages 248–263. Springer, 2000. 3.2.2

[110] M. Stoelinga. An introduction to probabilistic automata. *Bulletin of the EATCS*, 78:176–198, 2002. 2

[111] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Berlin, New York: Springer-Verlag, 2002. 5.2.3

[112] J. Sun, Y. Liu, and J. S. Dong. Model checking csp revisited: Introducing a process analysis toolkit. In *ISoLA*, pages 307–322, 2008. 2.4

[113] J. Sun, Y. Liu, J. S. Dong, and C. Chen. Integrating specification and programs for system modeling and verification. In W.-N. Chin and S. Qin, editors, *Proceedings of*

the third IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE'09), pages 127–135. IEEE Computer Society, 2009. 1.1, 3.1, 3.3.1

[114] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards flexible verification under fairness. In *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 709–714. Springer, 2009. 2.4

[115] J. Sun, Y. Liu, J. S. Dong, and H. H. Wang. Specifying and verifying event-based fairness enhanced systems. In *ICFEM*, pages 5–24, 2008. 2.4

[116] J. Sun, Y. Liu, J. S. Dong, and X. Zhang. Verifying Stateful Timed CSP Using Implicit Clocks and Zone Abstraction. In *ICFEM*, pages 581–600, 2009. 6.3.1, 6.4, 6.7

[117] J. Sun, Y. Liu, A. Roychoudhury, S. Liu, and J. S. Dong. Fair model checking with process counter abstraction. In *FM*, pages 123–139. Springer, 2009. 4.1

[118] J. Sun, Y. Liu, S. Song, J. S. Dong, and X. Li. Prts: An approach for model checking probabilistic real-time hierarchical systems. In *ICFEM*, pages 147–162, 2011. 1.3

[119] J. Sun, S. Song, and Y. Liu. Model checking hierarchical probabilistic systems. In *ICFEM*, volume 6447 of *Lecture Notes in Computer Science*, pages 388–403. Springer, 2010. 1.3, 6.7

[120] Y. S. T. Grenager, R. Powers. Dispersion games: general definitions and some specific learning results. In *AAAI'02*, pages 398–403, 2002. 1.1, 4.1, 4.1, 4.2.3, 4.4.2, 7.1

[121] E. M. Tadjouddine, F. Guerin, and W. Vasconcelos. Abstraction for model checking game-theoretical properties of auction(short paper). In *AAMAS'08*, pages 1613–1616, 2008. 4.6

[122] R. E. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972. 4.4.1, 5.3.1

[123] R. K. Treiber. Systems programming: Coping with parallelism. Technical report, IBM Almaden Research Center, 1986. 3.6.3

[124] S. Tripakis. Verifying Progress in Timed Systems. In *5th International AMAST Workshop ARTS on Formal Methods for Real-Time and Probabilistic Systems*, volume 1601 of *Lecture Notes in Computer Science*, pages 299–314. Springer, 1999. 6.7

[125] S. Tripakis. Checking Timed Büchi Automata Emptiness on Simulation Graphs. *ACM Transactions on Computational Logic*, 10(3):1–19, 2009. 6.7

[126] S. Tripakis, S. Yovine, and A. Bouajjani. Checking Timed Büchi Automata Emptiness Efficiently. *Formal Methods in System Design*, 26(3):267–292, 2005. 6.7

[127] K. Tuyls, K. Verbeeck, and T. Lenaerts. A selection-mutation model for q-learning in multi-agent systems. In *AAMAS'03*, pages 693–700, 2003. 4.1, 4.1

[128] M. Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *LICS*, pages 332–344. IEEE Computer Society, 1986. 3.2.2

[129] J. M. Vidal and E. H. Durfee. Predicting the expected behavior of agents that learn about agents: The clri framework. *AAMAS*, 6:77–107, 2003. 4.1, 4.1

[130] P. Vytelingum, D. Cliff, and N. Jennings. Strategic bidding in continuous double auctions. *Artificial Intelligence*, 172(14):1700–1729, 2008. 4.2.2

[131] F. Wang and M. Kwiatkowska. An MTBDD-based Implementation of Forward Reachability for Probabilistic Timed Automata. In *ATVA*, pages 385–399, 2005. 6.7

[132] T. Wang, S. Song, J. Sun, Y. Liu, J. S. Dong, X. Wang, and S. Li. More anti-chain based refinement checking. In *ICFEM*, pages 364–380, 2012. 1.3

[133] C. R. Williams, V. Robu, E. H. Gerding, and N. R. Jennings. Using gaussian processes to optimise concession in complex negotiations against unknown opponents. In *Proceedings of IJCAI'12*, pages 432–438, 2012. 4.1, 4.1, 4.5.1.2

[134] M. Wooldridge. Agent-based software engineering. *IEE Proceedings on Software Engineering*, 144(1):26–37, 1997. 1

[135] M. Wooldridge, M. Fisher, M. P. Huget, and S. Parsons. Model checking multi-agent systems with mable. In *AAMAS'02*, pages 952–959, 2002. 4.6

[136] M. D. Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 17–30. Springer, 2006. 3.2.3, 3.2.3, 3.7

[137] M. D. Wulf, L. Doyen, N. Maquet, and J.-F. Raskin. Antichains: Alternative algorithms for ltl satisfiability and model-checking. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 63–77. Springer, 2008. 3.7

[138] W.Visser, K. Havelund, G.Brat, and S.Park. Model checking programs. In *ASE'00*, pages 3–12, 2000. 4.6

[139] Y.Azar, A.Z.Broder, A.R.Karlin, and E.Upfa. Balanced allocations. *SIAM Journal on Computing*, 29(1):190–200, 2000. 4.1

[140] H. Yong. The evolution of conventions. *Econometrica*, 61(1):57–84, 1993. 4.2.2

[141] H. L. S. Younes, E. M. Clarke, and P. Zuliani. Statistical Verification of Probabilistic Properties with Unbounded Until. In *SBMF*, pages 144–160, 2010. 5.1

[142] M. Zheng, J. Sun, Y. Liu, J. S. Dong, and Y. Gu. Towards a model checker for nesc and wireless sensor networks. In *Formal Methods and Software Engineering*, pages 372–387, 2011. 2.4

[143] M. Zheng, J. Sun, D. Sanán, Y. Liu, J. S. Dong, and Y. Gu. Towards bug-free implementation for wireless sensor networks. In *SenSys*, pages 407–408, 2011. 2.4

[144] M. C. Zheng. An automatic approach to verify sensor network systems. *Secure Software Integration and Reliability Improvement Companion, IEEE International Conference on*, 0:7–12, 2010. 2.4

[145] H. Zhu, S. Qin, J. He, and J. Bowen. PTSC: Probability, Time and Shared-Variable Concurrency. *International Journal on Innovations in Systems and Software Engineering*, 5(4):271–294, 2009. 3.7

Appendix A

Concrete Operational Semantics

The following are concrete firing rules associated with process constructs other than those discussed in Chapter 6.

$$\frac{}{(V, \mathit{Stop}) \xrightarrow{\epsilon} (V, \mathit{Stop})} \; [\ st\ ]$$

$$\frac{}{(V, \mathit{Skip}) \xrightarrow{\epsilon} (V, \mathit{Skip})} \; [\ sk1\ ] \qquad \frac{}{(V, \mathit{Skip}) \xrightarrow{\checkmark} (V, \mathit{Stop})} \; [\ sk2\ ]$$

$$\frac{}{(V, e\{prog\} \rightarrow P) \xrightarrow{\epsilon} (V, e\{prog\} \rightarrow P)} \; [\ as1\ ]$$

$$\frac{}{(V, e\{prog\} \rightarrow P) \xrightarrow{e} (upd(V, prog), P)} \; [\ as2\ ]$$

$$\frac{V \vDash b}{(V, \textbf{if}\ (b)\ \{P\}\ \textbf{else}\ \{Q\}) \xrightarrow{\tau} (V, P)} \; [\ if1\ ]$$

$$\frac{V \nvDash b}{(V, \textbf{if}\ (b)\ \{P\}\ \textbf{else}\ \{Q\}) \xrightarrow{\tau} (V, Q)} \; [\ if2\ ]$$

$$\frac{}{(V, \textbf{if}\ (b)\ \{P\}\ \textbf{else}\ \{Q\}) \xrightarrow{\epsilon} (V, \textbf{if}\ (b)\ \{P\}\ \textbf{else}\ \{Q\})}\ [\ if3\ ]$$

$$\frac{(V,P) \xrightarrow{e} (V',P')}{(V,P\Box Q) \xrightarrow{e} (V',P')}\ [\ ex1\ ] \qquad \frac{(V,Q) \xrightarrow{e} (V',Q')}{(V,P\Box Q) \xrightarrow{e} (V',Q')}\ [\ ex2\ ]$$

$$\frac{(V,P) \xrightarrow{\epsilon} (V,P'),\ (V,Q) \xrightarrow{\epsilon} (V,Q')}{(V,P\Box Q) \xrightarrow{\epsilon} (V,P'\Box Q')}\ [\ ex3\ ]$$

$$\frac{(V,P) \xrightarrow{\checkmark} (V,P')}{(V,P;\ Q) \xrightarrow{\tau} (V,Q)}\ [\ se1\ ] \qquad \frac{(V,P) \xrightarrow{\epsilon} (V,P'), \checkmark \notin En(V,P)}{(V,P;\ Q) \xrightarrow{\epsilon} (V,P';\ Q)}\ [\ se2\ ]$$

$$\frac{(V,P) \xrightarrow{e} (V',P'), \checkmark \notin En(V,P)}{(V,P;\ Q) \xrightarrow{e} (V',P';\ Q)}\ [\ se3\ ]$$

$$\frac{(V,P) \xrightarrow{e} (V',P'), e \notin \alpha(Q)}{(V,P \parallel Q) \xrightarrow{e} (V',P' \parallel Q)}\ [\ pl1\ ]$$

$$\frac{(V,Q) \xrightarrow{e} (V',Q'), e \notin \alpha(P)}{(V,P \parallel Q) \xrightarrow{e} (V',P \parallel Q')}\ [\ pl2\ ]$$

$$\frac{(V,P) \xrightarrow{x} (V,P'), (V,Q) \xrightarrow{x} (V,Q'), x \in (\alpha(Q) \cap \alpha(P)) \cup \mathbb{R}_+}{(V,P \parallel Q) \xrightarrow{x} (V,P' \parallel Q')}\ [\ pl3\ ]$$

$$\frac{(V,Q) \xrightarrow{x} (V',Q'), P \widehat{=} Q}{(V,P) \xrightarrow{x} (V',Q')}\ [\ def\ ]$$

Appendix B

Abstract Operational Semantics

The following are abstract firing rules associated with process constructs other than those discussed in Chapter 6.

$$\frac{}{(V, \mathit{Skip}, D) \overset{\checkmark}{\rightsquigarrow} (V, \mathit{Stop}, D^{\uparrow})} \quad [\ \mathit{aki}\ ]$$

$$\frac{V \vDash b}{(V, \textbf{if}\ (b)\ \{P\}\ \textbf{else}\ \{Q\}, D) \overset{\tau}{\rightsquigarrow} (V, P, D^{\uparrow})} \quad [\ \mathit{aif}1\ ]$$

$$\frac{V \nvDash b}{(V, \textbf{if}\ (b)\ \{P\}\ \textbf{else}\ \{Q\}, D) \overset{\tau}{\rightsquigarrow} (V, Q, D^{\uparrow})} \quad [\ \mathit{aif}2\ ]$$

$$\frac{}{(V, e\{\mathit{prog}\} \rightarrow P, D) \overset{e}{\rightsquigarrow} (\mathit{upd}(V, \mathit{prog}), P, D^{\uparrow})} \quad [\ \mathit{aev}\ ]$$

$$\frac{(V, P, D) \overset{x}{\rightsquigarrow} (V', P', D')}{(V, P \mid Q, D) \overset{x}{\rightsquigarrow} (V', P', D' \wedge \mathit{idle}(Q))} \quad [\ \mathit{aex}1\ ]$$

$$\frac{(V, Q, D) \overset{x}{\rightsquigarrow} (V', Q', D')}{(V, P \mid Q, D) \overset{x}{\rightsquigarrow} (V', Q', D' \wedge \mathit{idle}(P))} \quad [\ \mathit{aex}2\ ]$$

$$\frac{(V, P, D) \overset{e}{\rightsquigarrow} (V', P', D'), e \notin \alpha(Q)}{(V, P \parallel Q, D) \overset{e}{\rightsquigarrow} (V', P' \parallel Q, D' \wedge idle(Q))} \quad [\ apl1\ ]$$

$$\frac{(V, Q, D) \overset{e}{\rightsquigarrow} (V', Q', D'), e \notin \alpha(P)}{(V, P \parallel Q, D) \overset{x}{\rightsquigarrow} (V', P \parallel Q', D' \wedge idle(P))} \quad [\ apl2\ ]$$

$$\frac{\begin{array}{l}(V, P, D) \overset{e}{\rightsquigarrow} (V, P', D'),\\ (V, Q, D) \overset{e}{\rightsquigarrow} (V, P', D''), e \in \alpha P \cap \alpha Q\end{array}}{(V, P \parallel Q, D) \overset{e}{\rightsquigarrow} (V, P' \parallel Q', D' \wedge D'')} \quad [\ apl3\ ]$$

$$\frac{(V, P, D) \overset{x}{\rightsquigarrow} (V', P', D'), x \neq \checkmark}{(V, P;\ Q, D) \overset{x}{\rightsquigarrow} (V', P';\ Q, D')} \quad [\ ase1\ ]$$

$$\frac{(V, P, D) \overset{\checkmark}{\rightsquigarrow} (V', P', D')}{(V, P;\ Q, D) \overset{\tau}{\rightsquigarrow} (V', Q, D')} \quad [\ ase2\ ]$$

$$\frac{(V, Q, D) \overset{x}{\rightsquigarrow} (V', Q', D'), P \mathrel{\widehat{=}} Q}{(V, P, D) \overset{x}{\rightsquigarrow} (V', Q', D')} \quad [\ adef\ ]$$