SYMBOLIC IMPLEMENTATION OF MODEL-CHECKING PROBABILISTIC TIMED AUTOMATA

by

FUZHI WANG

A thesis submitted to
The University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
The University of Birmingham
September 2006

Abstract

In this thesis, we present symbolic implementation techniques for model checking probabilistic timed automata as models for systems, for example, communication networks and randomised distributed algorithms. Given a system model as probabilistic timed automata and a specification, such as, "a leader will be elected within 5 time units with probability 0.999" and "the message can be successfully delivered within 3 time units with probability 0.985", a probabilistic real-time model checker can automatically verify if the model satisfies the specification. Motivated by the success of symbolic approaches to both non-probabilistic real-time model checking and untimed probabilistic model checking, we present a symbolic implementation of model checking for probabilistic timed automata, which is based on the data structure called Multi-terminal Binary Decision Diagrams (MTBDDs). Our MTBDD-based symbolic implementation is generic with respect to the support of the real-time information. Two data structures representing the real-time information are supported, Difference Bound Matrices (DBMs) and Difference Decision Diagrams (DDDs). We develop explicit and symbolic implementations of model checking for probabilistic timed automata, focusing on probabilistic reachability properties. The explicit implementation concerns both forward and backward generation of the zone graph, where the latter results in non-convex zones in the case of calculating the minimum probability. The symbolic implementation is of the forward exploration and is built on the PRISM software. We evaluate the performance of the implementation on several real-world protocol case studies. The thesis demonstrates that model checking for probabilistic timed automata is feasible in practice and can be efficient on some real-world protocols. We conclude that to ensure efficiency of the implementation of the backward approach, support for efficient manipulation of non-convex zones and canonicity are important.

Acknowledgments

I am grateful to my supervisor Prof. Kwiatkowska for helpful comments and suggestions that have made me progress. Special thanks go to Gethin Norman, David Parker and Jeremy Sproston for their help during my studies. I would also like to thank Gethin Norman for taking the time to read and comment on an early draft of this thesis. I would like to thank Mark Ryan and Hayo Thielecke who gave me helpful feedback and comments on my research. I would also like to acknowledge the authors of the DDD library, Jesper Møller, Jakob Lichtenberg, Henrik Hulgaard and Henrik Reif Andersen, for letting us use their DDD implementation. Thanks must also go to the School of Computer Science at the university of Birmingham and Overseas Research Students Award Scheme (ORS) for funding my studies. Finally, I would like to thank my wife and my parents for their patience and support throughout.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Embedded computer-based systems have been used increasingly in nearly every aspect of modern life. In safety-critical applications, it could result in disasters, for example, financial or even loss of human life if such systems fail. Traditional methods of checking such a system satisfying its requirements, for example, techniques based on testing and simulation, are often inadequate to detect errors, especially in highly concurrent designs. As embedded systems become more pervasive and complex, verification and analysis tools are desirable during the design stage. Much effort has been taken to advance techniques to automatically verify the correctness of these systems.

Correctness is relative to what the system is supposed to do and such requirements are written in natural language in traditional methods. However, descriptions in natural language are often ambiguous. A formal language with mathematical precision is desirable in order to avoid the ambiguities. In verification, formal methods have been put into practice. A successful application of formal verification in computer science is model checking, which is due to the fact that the method is fully automatic. There are two core elements in model checking. Firstly, a formal description with well-defined semantics for the systems and desired properties must be given. Secondly, an algorithmic method can be applied to reason about formal statements. In model checking, the systems are often modeled as kinds of transition systems and the properties are specified as kind of logic formulae.

Embedded systems, especially safety-critical systems often exhibit both stochastic and

timing behaviour because the environment with which the systems interact is stochastic and time-changing in nature. Communication networks and randomised distributed algorithms are two real-life examples. Besides the correctness properties, we are also interested in verification of the performance and reliability properties. These kinds of properties contain information about both probability and timing, such as,"a leader will be elected within 5 time units with probability 0.999".

Since the introduction of model checking [CE82, QS82], it has been extended to real-timed systems [ACD90, AH91, UPP, KRO] and probabilistic systems [HJ94, PRI]. However, model checking for probabilistic real-time systems is still largely in the theoretical phase. In [KNSS99], a method is proposed for dealing with probabilistic timed automata which have both probabilistic and timed behaviours. The properties are specified in logic Probabilistic Timed Computation Tree Logic (PTCTL) [KNSS99]. However, the complexity of the verification algorithms for such systems is too high because they are based on constructing the region graph [AD94] which is exponential in both the number of clocks and the length of the clock constraints used in the model and the specification. Optimised algorithms such as the forward reachability analysis [KNSS02] and backward reachability analysis [KNS00, KNS03b] have been proposed. These algorithms have the potential to reduce the high complexity of model checking for probabilistic timed automata as demonstrated by an implementation of forward reachability in [DKN02, DKN04]. The forward reachability algorithm can reduce the complexity when constructing quotient of the region graph by performing forward search to construct the zone graph which is often smaller than the region graph, although the worst case complexity is same. One drawback of the forward reachability is that it can only obtain an upper bound of the maximum probability of reaching a set of target states [KNSS02]. However, by using the backward reachability algorithm [KNS00, KNS03b], both the exact maximum and minimum probability of reaching the target set can be obtained. Furthermore, the backward approach can be applied to verify full PTCTL.

Although algorithms have been proposed for analysing real-time probabilistic systems, they have not been implemented within a single software tool. One difficulty for implementing model checking algorithms for them is that both probability and timing

information have to treated simultaneously and existing symbolic data structure based on Binary Decision Diagrams (BDDs) [Bry86] could not handle both timed transitions and probability distributions at the same time. Case studies such as that in [DKN02], which combines KRONOS [KRO] and PRISM [PRI] to verify IEEE-1394 Root Contention Protocol, is not a fully symbolic implementation and moreover, can be inefficient as it requires model transformation via textual files. First, a set of states and probabilistic transitions among them reached before a deadline starting from the source state is calculated by using KRONOS [KRO]. Then, the resulting probabilistic model is written in a textual file and input into PRISM and probabilistic analysis is performed by PRISM. The analysis could be more efficient if the two steps could be implemented in a single symbolic tool. Since probabilistic real-time systems contain information about both real-time and probability values, we have to deal with them at the same time if we want to efficiently perform analysis of real-time probabilistic systems. In the case of model checking for non-probabilistic real-time systems, the time is modelled as clocks ranging over the domain of reals, which means the state space is infinite. Region graph techniques [AD94] reduce the infinite time space to finite state space and symbolic methods are applied to handle real-time information. Similarly, in the case of model checking for probabilistic systems, symbolic methods are adapted to handle the probability information. However, current symbolic approaches can not represent real-time information and probability values in a single framework. Thus, developing a symbolic approach that can symbolically represent both real-time information and probability values at the same time has been set as the goal of this thesis.

Main Contributions of the Thesis

Base on algorithms proposed in [KNS00, KNSS02, KNS03b], an explicit implementation of the forward and the backward algorithms for model checking of probabilistic timed automata is presented and experimentally evaluated. Bottlenecks are identified for each of them. A symbolic implementation for the efficient model checking of probabilistic reachability for probabilistic timed automata via the forward algorithm is presented. A symbolic representation is developed using Multi-terminal Binary Decision Diagrams (MTBDDs)

[CFM+93] as the underlying representation of the finite-state graph, which is generic since the underlying representation of the timing information could be any of existing symbolic data structures for timing. Two data structures are experimentally evaluated, namely Difference Bound Matrices (DBMs) [Dil89] and Difference Decision Diagrams (DDDs) [MLAH99c]. Performance of the symbolic implementation is compared to the symbolic approach based on digital clocks [KNS03a]. Using three case studies, FireWire root contention protocol (the Tree Identify Protocol of the IEEE 1394 High Performance Serial Bus modelled in [SV99], IEEE 802.3 CSMA/CD (Carrier Sense, Multiple Access with Collision Detection) protocol [NSY92] and Milner's scheduler [Mil89], we conclude that symbolic model checking for real-world case studies is feasible, but the efficiency depends on regularity, the model size and the length of constants. We identify the main contribution to the performance slow-down for both of the algorithm and the data structure in the case of the calculating the minimum probability using the backward approach. Finding a data structure with support for both non-convex zones and canonicity is a challenging future research direction. All source code developed as part of this thesis is available from [PTA].

Other Publications Except for the DDD [ML98] library and some function calls from PRISM, all the implementations described in this thesis and in [KNSW04] and [WK05] are entirely the work of the author. Some of the work in this thesis has previously been published in jointly authored papers. In [KNSW04], an early version of the explicit implementation of the backward algorithm was presented, which is included in Chapter 7, and this version is developed using Java and C programming language by the author.

The design of the data structure and the model description language used in Chapters 6, 7 and 8 are entirely the work of the author. An early symbolic version of the algorithm in Chapter 8 was published in [WK05], which presented results for the symbolic implementation based on Multi-Terminal Binary Decision Diagrams (MTBDDs) [CFM+93] .

Layout of the Thesis

Chapter 2 gives information on formal verification and technical review of related work. Chapters 4 and 5 present background information on model checking non-probabilistic

timed systems, untimed probabilistic systems and probabilistic timed automata and information on symbolic representation of timing information and probability matrices. Chapters 6 to 8 discuss our implementation, in explicit and symbolic form. Experimental results and evaluation on selected case studies are given in each chapter. In Chapters 6 and 7, we consider the implementation of model checking for probabilistic timed automata via the forward and backward algorithms respectively, in an explicit form. A novel symbolic implementation of the forward approach is given in Chapter 8, where comparison between symbolic implementation and the approaches presented in [DKN02, DKN04] is given. Chapter 9 concludes with a critical evaluation of the work of this thesis and suggestion for future direction.

Chapter 2

Review of Related Work

In this chapter we present an overview of the established approaches to formal system verification and summarise the work from the literature which is closely related to the topic of this thesis. We start with the two dominant formal verification approaches in Section 2.1. Section 2.2 gives an overview of model checking. An overview of symbolic model checking is given in Section 2.3. Sections 2.4, 2.5 and 2.6 contain an overview of model checking for probabilistic systems, timed systems and probabilistic timed systems, respectively.

2.1 Formal Verification

There are two dominant approaches to formal verification [CWA+96] of software and systems, theorem proving [Cha73, Duf91] and model checking [WVF95, CES86, CGP99, BBF+01]. In theorem proving, the system model and system specification to be proved are described in terms of mathematical statements. The process of verification is about proving theorems of the system. The proof must show that the statement of the theorem can be formally derived from axioms using inference rules. In general, this process is semi-automatic and the user must be expert in logic, in that he must guide the process of verification. However, theorem proving is more powerful because it can deal with infinite state space. The other formal verification is model checking. Unlike theorem proving, the process of model checking is completely automatic.

2.2 Model Checking

Model checking is basically an exhaustive graph search technique that checks whether a state machine satisfies certain properties. Model checking comes in two varieties according to the the way the properties are expressed. In automata-theoretic approach, both the system and the specification are described in automata. And questions about systems and their specifications, such as satisfiability of specification and correctness of systems, can be reduced to questions about emptiness and containment of automata.

In temporal-logic model checking, the system is modelled as an automaton, while the specification is described in a temporal logic. The question of satisfiability is to determine whether or not the system satisfies a property specified in temporal logic. In other words, the basic technique of model checking is to model the system by a finite state automaton and use a model checking algorithm to verify whether the automaton has the properties expressed in temporal logic.

2.2.1 Temporal Logic

Temporal logic is a form of logic especially appropriate for statements and reasoning about behaviours of order in time. Although first-order logic is rather expressive and can express events of order in time, it is not intuitive since it explicitly uses variables to refer to objects. Temporal logic was invented to suppress explicit first-order variables so as to formalise natural language sentences about events in time by using modal adverbs like "possible" and "always". Temporal logics come in two varieties: linear and branching. In linear temporal logic (LTL) [Pnu77, Var95], formulae are interpreted over infinite words, while in branching temporal logics, for example, Computation Tree Logic (CTL) [CE82], formulae are interpret over infinite trees. According to linear or branching logic, the algorithms of model checking come into LTL and CTL model checking, respectively. There are different types of properties, and each type of property matches or corresponds to a particular type of temporal formula. Among these properties, the safety property, the liveness property and reachability property[Lam77] are most useful when analysing systems. Informally, a safety property specifies that "bad things" do not happen on all executions of a system, a liveness property specifies that "good thing" eventually happen

on all executions of a system and a reachability property specifies that a set of states can be reachable from initial states. In terms of verifying reachability properties, there are two kinds of approaches, the forward and backward approaches [LV93a, LV93b]. Informally, the forward algorithm starts in the initial state and computes the set of successors in the reachability graph, it terminates when the intersection of reachable states with the intended state is non-empty. The backward algorithm starts in a final state and computes the set of predecessors in the reachability graph. It terminates when the intersection of reachable states with the initial state is non-empty.

2.2.2 State Explosion Problem

Model checking techniques suffer from a well-known problem - the state-explosion problem. This problem is caused when the size of the state space generated becomes so large that it is impossible to represent it in the computer memory given current memory configuration of computers.

To cope with the problem of state explosion encountered in model checking, there have been a number of verification techniques developed for dealing with the state explosion problem, for example, symbolic methods which are based on Binary Decision Diagram or its variants [CFM+93], abstraction [CGL94, DT98], partitioning [BCL91], partial order reduction [WW96, GKPP95], on-the-fly [BTY97a] and symmetry [CGP99].

Model checking algorithms can be classified according to two criteria.

The first criterion concerns the basic information unit that the algorithm deals with. Enumerative algorithms reason in terms of single states which are represented explicitly during the search of the state-space. Symbolic algorithms reason in terms of sets of states, which are represented implicitly by means of predicates.

The second criterion concerns the time during which the state-space is actually generated. In so called on-the-fly algorithms, states are computed and stored on demand, while in other algorithms, the whole state-space has to be generated a priori.

Both symbolic and on-the-fly algorithms have often been proved useful in practice in tackling the state-explosion problem.

2.3 Symbolic Model Checking

The early model checkers used explicit analysis techniques to enumerate the states of the system. This explicit methods was subject to the well-known problem of the state-explosion as the state space that has to be explicitly represented may increase exponentially, especially in the case of parallel composition of a system which consists of a number of modules acting concurrently. Several techniques of model abstraction and state-space reduction have been developed to address the state-explosion problem. These techniques can be adopted to reduce the state space, but they may cause some loss of information. This may be unacceptable, for example when safety properties are verified. One of the most successful approaches that can allow a larger state space to be explored is symbolic model checking. Symbolic model checking is first introduced with the work of McMillan [BCM+90, McM93], in which a data structure called Binary Decision Diagrams (BDDs) [Bry86] is used to implicitly represent the set of states and the transition relation between states. BDDs were popularised by the work of Bryant [Bry86], who developed a set of efficient algorithms for manipulating the data structure introduced in the work of [Lee59, Ake78] by placing ordering on them. Since the work of [BCM+90, McM93], symbolic model checking is used to refer to a technique used in model checking to implicitly represent and manipulate the states and the transition relation of a system.

Although the techniques of model checking were oriented toward the verification of hardware circuits [BC95, CGH+93, BCL+94], they have been extended and applied to probabilistic systems and timed systems, for which corresponding symbolic data structures have also been developed.

2.4 Model Checking Untimed Probabilistic Systems

Formal verification based on temporal logic has been successfully extended to the verification problems of probabilistic systems. Early works in this field were focusing on the verification of qualitative properties. These included work of [CY88] which considered models of two types, Discrete-Time Markov Chains (DTMCs) and Markov decision processes (MDPs). The verification of quantitative properties is more involved than that of

qualitative properties since the exact probability has to be computed for a given property in addition to the satisfaction of that property. In the work of [HJ94], the temporal logic PCTL was introduced for the verification of DTMCs. The verification of quantitative properties for MDPs was considered in [CY90, BdA95, BK98]. In recent years much progress has been made concerning model checking probabilistic systems which are modelled as variants of Markov Chains. Tools such as PRISM (Probabilistic Symbolic Model Checker) [PRI, KNP00, KNP01] have been developed and applied to several real-world case studies. Other tools include ETMCC [HKMKS00], CASPA [KSW04] and MRMC (Markov Reward Model Checker) [KKZ05].

2.4.1 Probabilistic Model Checker - ETMCC

ETMCC [HKMKS00] is developed jointly by the Stochastic Modeling and Verification group at the University of Erlangen-Nürnberg, Germany, and the Formal Methods group at the University of Twente, the Netherlands. ETMCC is the first implementation of a model checker for Discrete-Time Markov Chains (DTMCs) and Continuous-Time Markov Chains (CTMCs). It uses numerical methods to model check PCTL [HJ94] and CSL [ASSB96, BKH99a] formulas respectively for DTMCs and CTMCs. The current version of ETMCC comes along with an experimental model checking engine supporting verification techniques to check action based CSL (aCSL) [RDN90] requirements against action-labelled continuous time Markov chains.

2.4.2 Probabilistic Model Checker - PRISM

PRISM [PRI, KNP00, KNP01], which is a tool designed for the analysis of probabilistic systems, is a probabilistic model checker being developed at the University of Birmingham. PRISM has been used to analyse several real-world case studies [PRI]. There are three different types of probabilistic models that PRISM can support directly: Discrete-Time Markov Chains (DTMCs), Markov decision processes (MDPs) and Continuous-Time Markov Chains (CTMCs). Here we are only interested in model checking MDPs because the semantics of probabilistic timed automata takes the form of MDPs. The PRISM model checker accepts a simple, module-based system description language. The tool

translates the system description into the appropriate model and then computes the set of reachable states. The model can then be analysed against a given specification. In the case of MDPs, this the tool supports PCTL [BK98, BdA95] model checking. PRISM supports three different model checking engines: one symbolic using MTBDDs (Multi-Terminal Binary Decision Diagrams); one based on sparse matrix techniques; and one hybrid engine which combines both symbolic and sparse approaches.

2.5 Model Checking Non-probabilistic Timed Systems

The techniques presented above are suited for the verification of systems where only causal relations of time are important. There are some applications where it is desirable to consider quantitative aspects of timing behaviour. However, this problem becomes more difficult when we consider real-time model checking. As it was noted in [DT98], there are three factors which are affecting the size of the state space. The state space under consideration grows exponentially not only with the number of concurrent components, but with the number of clocks and the length of the clock constraints used in the model and the specification as well.

In the last few years, model checking has been successfully implemented for non-probabilistic real-time systems which are modelled as timed automata [AD94] and a number of tools for automatic verification of systems have emerged [UPP, KRO, HYT]; these tools have by now reached a state, where they are mature enough for application on realistic case studies [JWK+96].

There are different formalisms for modelling timed systems, among which, timed automata which were proposed by Alur and Dill [AD94], are one of the most successful formalisms for the description of timed systems. A timed automaton is an ordinary automaton extended with real-valued clocks, which increase at the same rate as time. Clock values are usually assumed to be nonnegative real. Unlike traditional model checking which is performed on finite state automata, timed automata have infinite state space because of the real value of the clocks. Since clocks are real-valued, the state space of

timed automata is infinite. Much of the work on model checking timed automata is focused on using a finite representation for the infinite state space. All of them are based on the decidability of region techniques [AD94], in which the decidable result is obtained by the definition of regions, an finite partition on the infinite state space. However, model checking algorithms usually work on zones instead of regions because the region technique is not feasible due to its exponential characteristics. Zones, which are unions of regions, are an alternative way to obtain a finite representation for the infinite state space.

2.5.1 Real-time Model Checker - UPPAAL

UPPAAL is a tool suite for validation and verification of real-time systems modelled as networks of timed automata extended with data variables. UPPAAL consists of three main parts: a graphical user interface, a simulator and a model-checker engine. Modelling can be done in the graphical user interface. The simulator is helpful when debugging design errors because it can run interactively to check whether the system works as intended and generate traces. The verifier checks for simple invariants and reachability properties for efficiency reasons. Other properties may be checked by using test automata or systems decorated with debugging information [LPY97b]. UPPAAL implements the forward search algorithm in which the state space is explored in a breadth-first manner. It also uses on-the-fly verification combined with a symbolic technique, reducing the verification problem to that of solving simple constraints systems. The computation of clock constraints is aided with the data structure known as Difference Bound Matrices (DBMs) [BY04]. The non-convex zones are stored and manipulated in the data structure called Clock Difference Diagrams (CDDs) [BLP+99].

2.5.2 Real-time Model Checker - KRONOS

KRONOS is a tool which implements a model checking algorithm for the timed temporal logic TCTL [ACD93], a timed extension of CTL [CGP99, HR00]. KRONOS implements both the forward and backward algorithms [DY95]. It allows one to express and verify not only reachability properties but liveness properties as well. The system is modelled as a set of concurrently operating timed auotmata. KRONOS supports verification based on

both the region and simulation graphs [BTY97b]. To improve the exploration of the state space, KRONOS also implements an on-the-fly technique. In this approach, a symbolic graph called a simulation graph is constructed. The computation of clock constraints is also aided with the DBM data structure.

2.6 Model Checking Probabilistic Timed Systems

Existing tools such as UPPAAL [UPP], KRONOS [KRO] and HyTech [HYT] can verify timing properties against models based on timed automata [AD94]. However, they do not provide probabilistic analysis, which is useful, for example, when calculating the likelihood of a certain behaviour or predicting systems' performance is needed.

Although the tool PRISM [PRI] provides the functionality to analyse probabilistic temporal properties of the system models, it cannot deal with dense-timed systems directly.

In [ACD91b], Alur, Courcoubetis and Dill presented a model-checking algorithm for probabilistic real-time systems; their specification was based on deterministic timed automata. This work was extended in the work of Moura and Pinto [MP02], where they allowed nondeterministic timed automata to be used to specify properties of probabilistic real-time systems.

In [ACD91a], Alur, Courcoubetis and Dill presented a model-checking algorithm for verification of TCTL formulae of probabilistic real-time systems. All these works, [ACD91b], [ACD91a] and [MP02], were restricted to the verification of qualitative properties.

In [BCHG$^+$97], a method for analysing the stochastic and timing properties of systems was proposed, which is achieved by the combination of the tool Verus [CCMM95] and the algorithm proposed in the paper. Although the work of [BCHG$^+$97] was based on MTBDDs and could verify a subset of PCTL, it was not real-time but in the realm of discrete time. This was similar to the work of [HGCC99], which only supported DTMC models.

In [Bea03], Beauquier proposed a model of probabilistic timed automata which was similar to the model used in this thesis. The model in [Bea03] differs in that it allows different enabling conditions for edges related to a certain action and it uses Bûchi conditions

as accepting conditions.

In [HM05], Dang and Zhang also proposed a model of probabilistic timed automata. However, they use Duration Calculus as the specification notation.

There are two, non-symbolic, methods for dense-time probabilistic timed automata: the forward exploration algorithm implemented in [DKN02, DKN04], and the experimental implementation of the backward exploration algorithm implemented in [KNSW04]. The forward method is not guaranteed to produce exact reachability probabilities [KNSS02], but only requires simple operations; on the other hand, backward analysis can produce exact probabilities and be applied to full Probabilistic Timed Computation Tree Logic (PTCTL) [KNSS02] but at a cost of higher computational complexity. The method of [DKN02, DKN04], which combines KRONOS [BDM+98] and PRISM [PRI] to verify the IEEE-1394 Root Contention Protocol, requires three steps: firstly, a set of states and probabilistic transitions among them reachable from the source state before the deadline is calculated using KRONOS [BDM+98]; secondly, the result is translated into a PRISM model description which is input into PRISM and, finally, probabilistic analysis is performed. This method could be inefficient because it can generate large files that have to be parsed by PRISM. Our experimental implementation of [KNSW04] suffers from the same problem, as it performs a translation into the PRISM modelling language.

Although PRISM has been able to support modelling systems using discrete time, it is infeasible for some cases. For example, because events can only occur at integer time values, some problems like bounded delay [CGP99] cannot be solved correctly.

The most commonly used data structure for representing timing information in real-time verification tools [BDM+98, LPY97b] is Difference Bound Matrices (DBMs) [CGP99, BY04]. A number of BDD-like data structures, e.g. Clock Difference Diagrams (CDDs) [LPWY99, BLP+99], Difference Decision Diagrams (DDDs) [MLAH99d, MLAH99c, MLAH99a, ML98] and Clock-Restriction Diagram (CRDs) [Wan03], have been proposed for use in verifying real-time systems, but not yet extended to the probabilistic case. MTBDDs have been successfully applied in model checking of probabilistic systems, and also probabilistic timed automata and timed systems with digital clocks [KNPS06]. Recently, MTBDDs have also been applied to real-time systems [SB03]. Although the approach in [SB03]

uses a single data structure MTBDDs to represent both timing and discrete information in order to leverage well-known techniques for BDDs or MTBDDs, it involves SAT-based analysis.

Chapter 3

Preliminaries

3.1 General Notations

Throughout this thesis, the Boolean, reals, non-negative reals, integers and naturals are written as $\mathbb{B}$, $\mathbb{R}$, $\mathbb{R}^+$, $\mathbb{Z}$ and $\mathbb{N}$, respectively. We use variables such as x, y, z, δ, t ranging over $\mathbb{R}$, c and d ranging over $\mathbb{Z}$ and i, j, k ranging over $\mathbb{N}$. We use the symbol ∞ for infinity.

Let U and V denote sets. The set operations of intersection, union, complementation and set difference are denoted $U \cap V$, $U \cup V$, $\bar{U}$ and $U \setminus V$. We use the symbol $\emptyset$ for the empty set. Set inclusion and strict inclusion are denoted by $U \subseteq V$ and $U \subset V$. We write $x \in U$ if x is a member of set U, and $x \notin U$ if x is not a member of set U. We use the symbol $\mathbb{Z}_\infty$ for $\mathbb{Z} \cup \{\infty\}$. The symbol $\bowtie$ is such that $\bowtie \in \{<, \leq, =, >, \geq\}$. The symbol $\gtrsim$ is such that $\gtrsim \in \{\geq, >\}$ and the symbol $\lesssim$ is such that $\lesssim \in \{\leq, <\}$.

Logical conjunction, disjunction, negation, implication and bi-implication are written as $\wedge$, $\vee$, $\neg$, $\Rightarrow$ and $\Leftrightarrow$.

Let AP be a fixed finite set of atomic propositions.

3.1.1 Clocks and Zones

Let $\mathcal{X} = \{x_1, \ldots, x_n\}$ be a set of variables in $\mathbb{R}^+$. We call these variables clocks.

Definition 3.1.1 *(Clock valuation) Given a set of clocks $\mathcal{X}$, clock valuation is a function*

$v : \mathcal{X} \mapsto \mathbb{R}^+$ *that assigns a non-negative real value* $v(x)$ *to each clock* $x \in \mathcal{X}$.

The set of all clock valuations is denoted by $\mathbb{R}^+_{\mathcal{X}}$. For $X \subseteq \mathcal{X}$, $v[X := 0]$ is the clock valuation v_1 such that $\forall x \in X.v_1(x) = 0$ and $\forall x \notin X.v_1(x) = v(x)$. For $\delta \in \mathbb{R}^+$, $v[X + \delta]$ is the clock valuation v_2 such that $\forall x \in \mathcal{X}.v_2(x) = v(x) + \delta$ and $\delta \cdot v$ is the valuation v_3 such that $\forall x \in \mathcal{X}.v_3(x) = \delta \cdot v(x)$. The clock valuation v *satisfies* $x \bowtie c$ if $v(x) \bowtie c$, v *satisfies* $x{-}y \bowtie c$ if $v(x) - v(y) \bowtie c$, where $\bowtie = < \text{ or } \leq$.

The set of *zones* of $\mathcal{X}$, written $Zones(\mathcal{X})$, is defined inductively by the syntax:

$$\zeta ::= x \bowtie c \mid x{-}y \bowtie c \mid \neg\zeta \mid \zeta \vee \zeta$$

where $x, y \in \mathcal{X}$, $c \in \mathbb{Z}$ and $\bowtie = < \text{ or } \leq$. As usual, $\zeta_1 \wedge \zeta_2 = \neg(\neg\zeta_1 \vee \neg\zeta_2)$. An *atomic zone* on $\mathcal{X}$ involves only one or two clocks, which take the form $x \bowtie c$ or $x - y \bowtie c$ where $x, y \in \mathcal{X}$, $c \in \mathbb{Z}$ and $\bowtie = < \text{ or } \leq$.

The *complementation* of an *atomic zone*

- $x \bowtie c$ is $-x\bar{\bowtie}(-c)$ or
- $x{-}y \bowtie c$ is $y{-}x\bar{\bowtie}(-c)$

where $\bar{\bowtie} = \leq$ *or* $<$ if $\bowtie = <$ *or* $\leq$ respectively.

The clock valuation v *satisfies* the zone ζ, written $v \triangleleft \zeta$, if and only if ζ resolves to true after substituting each clock $x \in \mathcal{X}$ with the corresponding clock value $v(x)$ from v.

Intuitively, the semantics of a zone is the set of clock valuations (subset of $\mathbb{R}^+_{\mathcal{X}}$) which satisfy the zone. This enables us to use the above syntax for zones interchangeably with semantic, set-theoretic operations.

Let ζ_1 and ζ_2 denote zones. The zone operations of intersection, union, complementation and set difference are denoted $\zeta_1 \cap \zeta_2$, $\zeta_1 \cup \zeta_2$, $\bar{\zeta_1}$ and $\zeta_1 \setminus \zeta_2$. Zones difference is defined via complementation as: $\zeta_1 \setminus \zeta_2 = \zeta_1 \cap \bar{\zeta_2}$. The test for inclusion $\zeta_1 \subseteq \zeta_2$ is equivalent to $\zeta_1 \setminus \zeta_2 = \emptyset$.

Note that more than one zone may represent the same set of clock valuations. We henceforth consider only canonical zones, which are zones for which the constraints are as "tight" as possible.

A zone ζ is called convex if for all $v_1, v_2 \triangleleft \zeta$, for any $0 < \delta < 1, \delta \cdot v_1 + (1-\delta) \cdot v_2 \triangleleft \zeta$. For example, atomic zones are convex. For any valid convex zone $\zeta \in \mathit{Zones}(\mathcal{X})$, there exists a $O(|\mathcal{X}|^3)$ algorithm to compute the canonical zone of ζ [Dil89].

If ζ is non-convex then it can be written as $\zeta_1 \cup \cdots \cup \zeta_k$, where $\zeta_1, \ldots, \zeta_k$ are all convex [Tri98].

Operations on zones

Definition 3.1.2 *(c-equivalence) Given $c \in \mathbb{N}$, two clock valuations v and v' are called c-equivalent if:*

- for any clock x, either $v(x) = v'(x)$, or $v(x) > c$ and $v'(x) > c$
- for any pair of clocks x, y, either $v(x) - v(y) = v'(x) - v'(y)$, or $|v(x) - v(y)| > c$ and $|v'(x) - v'(y)| > c$

Given a zone ζ, $\mathit{close}(\zeta, c)$ is defined as the greatest zone $\zeta' \supseteq \zeta$, such that for all $v' \triangleleft \zeta'$ there exists $v \triangleleft \zeta$ and v, v' are *c-equivalent*.

We require the following classical operations on zones [HNSY92, Tri98]. For zones $\zeta, \zeta' \in \mathit{Zones}(\mathcal{X})$ and subset a of clocks $X \subseteq \mathcal{X}$, let:

$$
\begin{aligned}
\zeta_{/X} &\stackrel{\text{def}}{=} \{v \mid \exists v' \triangleleft \zeta . \forall x \in X . v(x) = v'(x)\} \\
\swarrow \zeta &\stackrel{\text{def}}{=} \{v \mid \exists t \geq 0 . v{+}t \triangleleft \zeta\} \\
\nearrow \zeta &\stackrel{\text{def}}{=} \{v \mid \exists t \geq 0 . v{-}t \triangleleft \zeta\} \\
\swarrow_{\zeta'} \zeta &\stackrel{\text{def}}{=} \left\{v \mid \exists t \geq 0 . \left(v{+}t \triangleleft \zeta \ \wedge \ \forall t' \leq t . (v{+}t' \triangleleft \zeta \vee \zeta')\right)\right\} \\
[X := 0]\zeta &\stackrel{\text{def}}{=} \{v \mid v[X := 0] \triangleleft \zeta\} \\
\zeta[X := 0] &\stackrel{\text{def}}{=} \{v[X := 0] \mid v \triangleleft \zeta\} .
\end{aligned}
$$

The zone $\zeta_{/X}$ can be obtained by performing existential quantification on clocks in X from zone ζ. The zone $\swarrow \zeta$ contains the clock valuations that can, by increasing all clocks with a same value, reach a clock valuation in ζ. The zone $\nearrow \zeta$ contains the clock valuations that can, be reachable from a clock valuation in ζ by increasing all clocks with a same value. The zone $\swarrow_{\zeta'} \zeta$ contains the clock valuations that can, by increasing all clocks

with a same value, reach a clock valuation in ζ and remain in ζ' until ζ is reached. The zone $[X := 0]\zeta$ contains the clock valuations which result in a clock valuation in ζ when the clocks in X are reset to 0. The zone $\zeta[X := 0]$ contains the clock valuations which are obtained from clock valuations in ζ by resetting the clocks in X to 0.

The following results [Tri98] can be used to define $\zeta[X := 0]$ and $[X := 0]\zeta$ through $\zeta_{/X}$:

$$\zeta[X := 0] \stackrel{\text{def}}{=} \zeta_{/X} \cap \big(\bigwedge_{x \in X} (x = 0) \big)$$

$$[X := 0]\zeta \stackrel{\text{def}}{=} \big(\zeta \cap (\bigwedge_{x \in X} (x = 0) \big)_{/X}$$

3.2 Probability

3.2.1 Discrete Probability Distributions

A (discrete probability) *distribution* over a finite set Q is a function $\mu : Q \to [0, 1]$ such that $\sum_{q \in Q} \mu(q) = 1$. Let $\mathit{support}(\mu)$ be the subset of Q such that $q \in \mathit{support}(\mu)$ if and only if $\mu(q) > 0$. Given $Q' \subseteq Q$, we let $\mu(Q') = \sum_{q \in Q'} \mu(q)$. For any $q \in Q$, the *point distribution* μ_q denotes the distribution which assigns probability 1 to q. For a possibly uncountable set Q_∞, let $\mathsf{Dist}(Q_\infty)$ be the set of distributions over finite subsets of Q_∞.

Chapter 4

Model Checking for Timed Automata and Probabilistic Systems

In this chapter, we present an overview of the model checking techniques that have been developed for timed automata and probabilistic systems. Section 4.1 covers model checking for non-probabilistic timed systems. Section 4.2 introduces model checking for untimed probabilistic systems. Finally, the data structures used in this thesis are introduced in Section 4.3.

4.1 Model Checking Non-probabilistic Timed Systems

When analysing real-time aspects of systems, the idea is first to model a system using timed automata [AD94]. Then a model checker is used to check both timed and untimed reachability properties, i.e. if a certain set of states is reachable or not. In this section, we define timed automata and briefly describe some techniques used in their model checking.

4.1.1 Labeled Timed Transition System

Definition 4.1.1 *A* Labelled timed system *is a tuple* $LTTS = (S, \bar{s}, L, Act, T)$ *where:*

- S *is the (possible infinite) set of* states*;*

- $\bar{s} \in S$ *is the* initial state*;*
- $L : S \to 2^{AP}$ *is a* labelling function*;*
- Act *is the set of* actions*;*
- $T \subseteq S \times (Act \cup \mathbb{R}^+) \times S$ *is the* transition relation.

A labeled timed transition system LTTS is any graph with two types of labeled edges, discrete and time edges. Discrete edges are labeled with labels of the Act (i.e. $T \subseteq S \times Act \times S$). Time edges are labeled in $\mathbb{R}^+$ (i.e. $T \subseteq S \times \mathbb{R}^+ \times S$).

We use $s \xrightarrow{\delta} s'$ to denote a transition where $\delta \in \{\mathbb{R}^+ \cup Act\}$ is a lable which denotes the duration or an action of a transition between s and s'. A path of a LTTS is an infinite sequence of transitions $s_0 \xrightarrow{\delta_0} s_1 \xrightarrow{\delta_1} s_2 \cdots$ where either $\delta_i \geq 0$ or $\delta_i \in Act$. We denote by $Path(s)$ the set of paths starting in the state $s \in S$ in the LTTS.

For any path

$$\omega = s_0 \xrightarrow{\delta_0} s_1 \xrightarrow{\delta_1} s_2 \xrightarrow{\delta_2} \cdots$$

of a labeled timed transition system, the duration up to the n+1th state of ω, denoted $\mathcal{D}_\omega(n{+}1)$, equals $\sum_{i=0}^{n} t_i$ where $t_i = \delta_i$ if $\delta_i \in \mathbb{R}^+$ or $t_i = 0$ if $\delta_i \in Act$.

Definition 4.1.2 *A path of a labelled timed transition system LTTS is* divergent *if and only if for any $t \in \mathbb{R}^+$, there exists $j \in \mathbb{N}$ such that $\mathcal{D}_\omega(j){>}t$.*

We denote by $Path_{div}(s)$ the set of divergent paths starting in the state $s \in S$ in the LTTS.

Definition 4.1.3 *A labelled timed transition systems LTTS is* non-zeno *if and only if there are non-divergent paths for each state.*

4.1.2 Timed Automata

A timed automaton is an ordinary automaton extended with real-valued clocks, which increase at the same rate as time.

Definition 4.1.4 *(Timed automaton) A timed automaton is a tuple* $TA = (L, \mathcal{L}, \overline{l}, Act, \mathcal{X}, I, T)$ *which contains:*

- *a finite set* L *of nodes,*
- *a function* $\mathcal{L} : L \rightarrow 2^{AP}$ *assigning to each node of the automaton the set of atomic propositions that are true in that node,*
- *a start node* $\overline{l} \in L$*,*
- *a finite set* Act *of action,*
- *a finite set* $\mathcal{X}$ *of clocks,*
- *a function* $I : L \rightarrow Zones(\mathcal{X})$ *assigning to each node an invariant condition,*
- *a transition relation* $T \subseteq L \times Zones(\mathcal{X}) \times Act \times X \times L$ *that describes the edges of the automaton, which consists of a source node, an enabling condition called guard, an action label, a set of clocks to be reset and a target node*

Definition 4.1.5 *An* edge *of* TA *is a tuple of the form* (l, g, X, l') *where* $g \in Zones(\mathcal{X})$ *is the guard.*

There are two types of clock constraints: invariants labelling nodes, and guards labelling transitions. Each node of the timed automaton is labelled with an invariant, a Boolean condition on the clocks stating how long the system can stay in that node; transitions are labelled with actions and Boolean conditions called guards stating whether the action can be taken, and the set of clocks whose value is to be reset.

Figure 4.1 shows a simple timed automaton. Action labels are omitted. The example can be used to control the door opening or closing. If the door is in the state of closing and a control button is pressed, the door will be opened. Once the door is opened, it will stay open for 2 to 3 time units, then the door will close. There is no constraint on node 'close'. The constraint '$x < 3$' on node 'open' is an invariant. The 'true' guard denotes the fact that the button can be pressed at any time. The constraint '$x > 2$' is a guard.

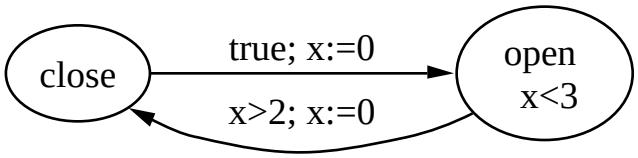

Figure 4.1: A simple timed automaton example

Semantics of Timed Automata

The semantics of a timed automata can be given in terms of labeled timed transition system.

Definition 4.1.6 *Let* $TA = (L, \mathcal{L}, \bar{l}, Act, \mathcal{X}, I, T)$ *be a timed automaton. The* semantics of TA *is defined as the labeled timed transition system*
$LTTS_{TA} = (S, \bar{s}, \mathcal{L}', Act, T')$ *where:*

- $S \subseteq L \times \mathbb{R}^+$ *and* $(l, v) \in S$ *if and only if* $v \triangleleft I(l)$*;*
- $\bar{s} = (\bar{l}, 0)$*;*
- $((l, v), \delta, (l', v')) \in T'$ *if and only if one of the following conditions holds:*
  time transitions: *there exists* $\delta' \in \mathbb{R}^+$*, such that* $v + \delta' \triangleleft I(l)$ *for all* $0 \leq \delta' \leq \delta$*,*
  discrete transitions: *there exists an edge* $e = (l, g, X, l')$ *such that* $v \triangleleft g$*,* $v' \triangleleft I(l')$ *and* $v' = v[X := 0]$*.*
- $\mathcal{L}'(l, v) = \mathcal{L}(l)$ *for any* $(l, v) \in S$ *and* $\delta \in \mathbb{R}^+ \cup Act$*.*

We say that TA is non-zeno if and only if $LTTS_{TA}$ is non-zeno.

A *position* of a path $\omega \in Path(s)$ is a pair (i, δ) consisting of a nonnegative integer i and a nonnegative real δ. The *duration* from the beginning of a path ω up to position (i, δ) is denoted as $\mathcal{D}_\omega(i) + \delta$. The state at position (i, δ) of ω is $\omega(i, \delta)$. For convenience, the state s at position (i, δ) of ω is also denoted as a pair of the form (l, v), where the first element is a node and the second element is a valuation of all of the clocks with the value at position (i, δ). The initial state is $s_0 = (\bar{l}, 0)$.

A state of the $LTTS_{TA}$ is a pair (l, v) where $l \in L$ is a node of TA and v is a clock valuation. The underlying model of timed automata takes the form of classical labelled transition systems enhanced with information about time. Since the clock values range over $\mathbb{R}^+$, there are infinitely many states in the transition system because clock valuations are part of states. In other words, the model of a timed automaton is an infinite state transition graph. The system starts its execution with all clocks set to 0. The automaton may only stay in a node, letting time pass, if the clocks satisfy the invariant. When a guard is satisfied, the corresponding transition can be taken. Additionally, a set of clocks can be reset to 0 when automaton executes a discrete transition. Note that the fact that clocks are real-valued means that the transition system is generally infinite state, and that the real-valued durations of time that may elapse from a state means that the transition system is generally infinitely branching.

We consider model checking over divergent paths only.

4.1.3 Timed Computation Tree Logic (TCTL)

TCTL employs a set of *formula clocks*, $\mathcal{Z}$, disjoint from the clocks $\mathcal{X}$ of the timed automaton. Formula clocks are assigned values by a *formula clock valuation* $\mathcal{E} : \mathcal{Z} \to \mathbb{R}$, which uses the notation for clock valuations in the standard way.

Definition 4.1.7 *The syntax of* TCTL *[HNSY92] is defined as follows:*

$$\phi ::= p \mid x + c \leq y + d \mid \neg\phi \mid \phi \vee \phi \mid z.\phi \mid \forall[\phi_1 \ \mathcal{U} \ \phi_2] \mid \exists[\phi_1 \ \mathcal{U} \ \phi_2]$$

where $p \in AP$, $x, y, z \in \mathcal{X} \cup \mathcal{Z}$ *are clocks,* $\bowtie \in \{\leq, <\}$ *and* $c, d \in \mathbb{N}$.

The logic TCTL can express timing constraints and includes the reset quantifier $z.\phi$, used to reset the clock z so that ϕ is evaluated from a state at which $z = 0$.

Definition 4.1.8 *Let* $LTTS_{TA} = (S, \bar{s}, \mathcal{L}', Act, T')$ *be a labelled timed transition system. For any state* $s \in S$ *and formula clock valuation* $\mathcal{E}$*, the satisfaction relation* $s, \mathcal{E} \models \phi$ *is*

defined inductively as follows:

$$
\begin{aligned}
&s, \mathcal{E} \models \mathit{true} \\
&s, \mathcal{E} \models a && \Leftrightarrow \quad a \in L(s) \\
&s, \mathcal{E} \models \neg\phi && \Leftrightarrow \quad s, \mathcal{E} \not\models \phi \\
&s, \mathcal{E} \models \phi_1 \vee \phi_2 && \Leftrightarrow \quad s, \mathcal{E} \models \phi_1 \text{ or } s, \mathcal{E} \models \phi_2 \\
&s, \mathcal{E} \models z.\phi && \Leftrightarrow \quad s, \mathcal{E}[z := 0] \models \phi \\
&s, \mathcal{E} \models \exists[\phi_1 \mathcal{U}\ \phi_2] && \Leftrightarrow \quad \text{there exists a path } \omega\ \in \mathit{Path}_{\mathit{div}}(s) \text{ there exists a position } (i,t) \\
&&& \qquad \text{of } \omega \text{ such that } \omega(i,t), \mathcal{E} + \mathcal{D}_\omega(i) + t\ \models\ \phi_2, \text{ and} \\
&&& \qquad \text{for all } j < i, \text{ we have:} \\
&&& \qquad \omega(j,t), \mathcal{E} + \mathcal{D}_\omega(j) + t\ \models\ \phi_1 \vee \phi_2 \\
&&& \qquad \text{and for } j = i \text{ and } t' < t \text{ we have:} \\
&&& \qquad \omega(j,t'), \mathcal{E} + \mathcal{D}_\omega(j) + t'\ \models\ \phi_1 \vee \phi_2. \\
&s, \mathcal{E} \models \forall[\phi_1 \mathcal{U}\ \phi_2] && \Leftrightarrow \quad \text{for all paths } \omega\ \in \mathit{Path}_{\mathit{div}}(s) \text{ there exists a position } (i,t) \\
&&& \qquad \text{of } \omega \text{ such that } \omega(i,t), \mathcal{E} + \mathcal{D}_\omega(i) + t\ \models\ \phi_2, \text{ and} \\
&&& \qquad \text{for all } j < i, \text{ we have:} \\
&&& \qquad \omega(j,t), \mathcal{E} + \mathcal{D}_\omega(j) + t\ \models\ \phi_1 \vee \phi_2 \\
&&& \qquad \text{and for } j = i \text{ and } t' < t \text{ we have:} \\
&&& \qquad \omega(j,t'), \mathcal{E} + \mathcal{D}_\omega(j) + t'\ \models\ \phi_1 \vee \phi_2.
\end{aligned}
$$

In TCTL, we can express properties such as: "the leader will be elected within 8 time units", which is represented as the TCTL formula $z.\forall[\mathit{true}\ \mathcal{U}\ (\mathit{leader} \wedge (z < 8))]$.

4.1.4 Symbolic States and Operations

A set of states of a $LTTS_{TA}$ is called a *symbolic state* which is denoted as (l, ζ) where l is the node of timed automata TA and ζ is the zone. We assume all states of a symbolic state U are associated with the same node.

Given a symbolic state $U = (l, \zeta)$ where ζ is a convex zone, $c \in \mathbb{N}$, we generalise the *close* operation on zones to symbolic states:

$$close(U, c) \stackrel{\text{def}}{=} (l, close(\zeta, c))$$

In this thesis, we define:

$$\begin{aligned} until(\zeta', \zeta) &\stackrel{\text{def}}{=} \swarrow_{\zeta'} \zeta \\ z.(U) &\stackrel{\text{def}}{=} (l, [X := 0]\zeta) \; where \; U = (l, \zeta) \end{aligned}$$

Given symbolic states $U = (l, \zeta')$ and $V = (l, \zeta)$, the operation $\swarrow_{\zeta'} \zeta$ on zones can also be generalised to symbolic states:

$$tpre_U(V) \stackrel{\text{def}}{=} \{(l, until(\zeta', \zeta))\} \; where \; U = (l, \zeta') \; and \; V = (l, \zeta).$$

Let U be a symbolic state, e an edge of TA and c a natural number constant. We define the following operations on U:

$$\begin{aligned} time\text{–}succ(U) &\stackrel{\text{def}}{=} \{s' | \exists s \in U, \delta \in \mathbb{R}. s \xrightarrow{\delta} s'\} \\ time\text{–}pred(U) &\stackrel{\text{def}}{=} \{s | \exists s' \in U, \delta \in \mathbb{R}. s \xrightarrow{\delta} s'\} \\ disc\text{–}succ(e, U) &\stackrel{\text{def}}{=} \{s' | \exists s \in U. s \xrightarrow{e} s'\} \\ disc\text{–}pred(e, U) &\stackrel{\text{def}}{=} \{s | \exists s' \in U. s \xrightarrow{e} s'\} \\ post(e, U, c) &\stackrel{\text{def}}{=} close(time\text{–}succ(disc\text{–}succ(e, U)), c) \\ pre(e, U) &\stackrel{\text{def}}{=} disc\text{–}pred(e, time\text{–}pred(U)) \end{aligned}$$

The operation $time\text{–}succ(U)$ computes symbolic state that can be reachable by staying at the same node as U and by increasing the clocks with same value. The symbolic state

computed by operation $time\text{–}pred(U)$ is that can reach U by staying at the same node as U and increasing the clocks with same value. The operation $disc\text{–}succ(U)$ is used for computing the symbolic state reachable from U by taking a discrete transition edge e. The result of the operation $disc\text{–}pred(U)$ is the symbolic state that can reach U by taking a discrete transition edge e. The symbolic state computed by operation $post(e, U, c)$ which applies both operations of $disc\text{–}succ$ and $time\text{–}succ$ is called successor state. The symbolic state computed by operation $pre(e, U, c)$ which applies both operations of $disc\text{–}pred$ and $time\text{–}pred$ is called predecessor state.

Given a timed automaton $TA = (L, \mathcal{L}, \overline{l}, Act, \mathcal{X}, I, T)$, symbolic states (l, ζ) and (l', ζ') and an edge $e = (l, g, X, l')$, the following results [Tri98] can be used to implement the $post$ and pre operations using basic operations on zones.

$$
\begin{aligned}
time\text{–}succ((l,\zeta)) &\stackrel{\text{def}}{=} (l, \nearrow \zeta \cap I(l)) \\
time\text{–}pred((l,\zeta)) &\stackrel{\text{def}}{=} (l, \swarrow \zeta \cap I(l)) \\
disc\text{–}succ(e,(l,\zeta)) &\stackrel{\text{def}}{=} (l', (\zeta \cap g[X := 0]) \cap I(l')) \\
disc\text{–}pred(e,(l',\zeta')) &\stackrel{\text{def}}{=} (l, [X := 0]\zeta' \cap g \cap I(l))
\end{aligned}
$$

4.1.5 Model Checking for Timed Systems

We will give here a short introduction to the ideas of the region and the zone techniques for timed automata. These will be used to generate finite-state quotient representations of timed automata.

The Region Graph Technique

The region graph technique [AD94] is due to the work of Alur and Dill. The method specifies how the infinite space of clock valuations can be partitioned into a finite set of equivalence classes which are called clock regions.

Region Equivalence The key idea behind clock regions is to define an equivalence relation, which is recalled briefly below.

For any $\delta \in \mathbb{R}$, $fr(\delta)$ denotes the fractional part of δ, and $intg(\delta)$ denotes the integral part of δ; that is, $\delta = intg(\delta) + fr(\delta)$. For each clock $x \in \mathcal{X}$, let $C \in \mathbb{N}$ be the smallest constant which is greater than or equal to the absolute value $|c|$ of every constant $c \in \mathbb{Z}$ appearing in an invariant or a guard. The equivalence relation between two clock valuations $v(x)$ and $v'(x)$ is defined as:

- For all $x \in \mathcal{X}$, either $intg(v(x))$ and $intg(v'(x))$ are the same, or both $v(x)$ and $v'(x)$ are greater than C
- For all $x, y \in \mathcal{X}$, $v(x) \leq C$ and $v(y) \leq C$ and $fr(v(x)) \leq fr(v(y))$ iff $fr(v'(x)) \leq fr(v'(y))$
- For all $x \in \mathcal{X}$, $v(x) \leq C$ and $fr(v(x)) = 0$ iff $fr(v'(x)) = 0$

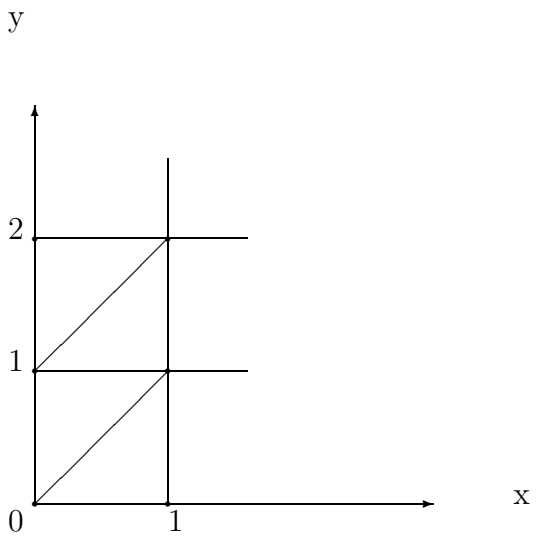

Figure 4.2: An example of regions

Figure 4.2 shows the partitioning of clock valuations for two clocks x and y, with $c_x = 1$ and $c_y = 2$. There are a total of 28 regions: 6 corner points (i.e. $x = 0, y = 2$), 14 open segments (i.e. $0 < x < 1, y = 2$) and 8 open regions (i.e. $0 < x < 1, 0 < y < 1, x < y$).

The Zone Technique

Model checking algorithms can work directly on the region technique to decide properties. However, the region technique is often not feasible because, although a finite graph is obtained by this technique, it still suffers from the state explosion problem. The number of the equivalence classes grows exponentially with the number of clocks and the length of the clock constraints used in the model and the specification. Zones are an alternative way to obtain a finite representation for the infinite state space. Zones are a coarser partitioning of the clock valuations; formally, zones are unions of regions.

On-the-fly Reachability Analysis

Much of the research concerned with model checking timed automata has been focused on forward approach which is capable for the reachability analysis [BBF+01]. The forward reachability approach is defined as follows: given a set of target states, determine whether a target state of a system is reachable from the initial state.
The reachability analysis is very important because, firstly, many properties such as safety properties are expressed as reachability or non-reachability of a certain set of states, and secondly, other classes of properties are also based on the same concept. The key idea behind reachability analysis is to construct a reachability graph which is much coarser than the region graph because only discrete transitions are explicit; the timed transitions are implicit and they are encoded inside the nodes of a reachability graph.

Algorithms [ACD93] are based on the notion of region graphs, in which the full reachability graph is constructed for a given automaton before the properties are checked directly on the reachability graph. However, it is inefficient to construct the whole reachability graph if the goal of verification is only to verify simple properties such as safety properties.

It has been pointed out in [YPD94] that the simple logical properties such as safety or time bounded properties can be verified without constructing the whole reachability graph of a timed system. In the case of model checking timed automata, most of the existing algorithms are now using on-the-fly techniques. Here, 'on-the-fly' means that, instead of searching the whole reachability graph, the model checking algorithm will terminate as soon as the answer is obtained. These on-the-fly techniques are implemented in tools such

as KRONOS [BDM+98, DOTY95] and UPPAAL [BLL+98, LPY97b].

The Forward Reachability Algorithm

Figure 4.3 shows the on-the-fly forward reachability algorithm [BY04], where (l_0, ζ_0) is the initial state with clocks set to 0 and (l_t, ζ_t) is the set of target states. The algorithm will return answer "YES" if the target state can be reached from initial state. Lines 1-2 initialise the set $Passed$ and $Wait$, which are used to store the set of symbolic states for those already explored and those waiting to be explored, respectively. Lines 3-12 generate the states. The "on-the-fly" means that the algorithm terminates as soon as possible when it finds the answer (Line 6). Line 4 takes a state from the set $Wait$, which is moved to the set $Passed$ (Line 9). Line 10 calculates the next state by the *post* operation. If the newly found state has not been explored before, it is put into the set $Wait$ in Line 11. Finally, line 15 returns answer "NO" if no positive answer is obtained.

Data Structures Based on Zone Techniques

Due to the successful symbolic encoding methods, for example by using Binary Decision Diagrams (BDDs) [Bry86] to represent sets of states and relations between states as predicates over Boolean variables, it is possible to verify systems with a very large number of states [BCM+90]. However, these symbolic methods do not easily generalise to real-time systems [ACD93] because such systems use dense real-valued variables to model time.

It has been pointed out in [MHA02] that, to solve the reachability problem for a timed system, there are two key problems that have to be addressed. One problem is how to represent the infinite state space R of a timed system, and the other problem is how to perform the basic operations (resetting clocks, advancing time, etc.) on this representation to compute the reachable state space.

A Difference Bound Matrix (DBM) [Dil89, LLPY97, CGP99] is a data structure which represents convex sets of regions (convex union of regions which is also called a zone). However, a set containing non-convex zones that are apart from each other cannot be

OnTheFlyForward$((l_0, \zeta_0), (l_t, \zeta_t))$

```
1.  Passed := ∅
2.  Wait := {(l_0, ζ_0)}
3.  while Wait ≠ ∅
4.      take (l, ζ) from Wait
5.      if (l = l_t) ∧ (ζ ∩ ζ_t ≠ ∅)
6.          return "YES"
7.      end if
8.      if not ζ ⊆ ζ' for all (l, ζ') ∈ Passed
9.          add (l, ζ) to Passed
10.         for all edges e = (l, g, X, l') such that (l', ζ') = post(e, (l, ζ))
11.             add (l', ζ') to Wait
12.         end for
13.     end if
14. end while
15. return "NO"
```

Figure 4.3: The on-the-fly forward reachability algorithm

efficiently represented. Using DBMs, non-convex unions of zones may be represented by a list of DBMs, each representing one zone of the union. This representation is inefficient because the number of DBMs will become very large as the number of non-convex zones increases. In particular, this representation increases the computational complexity of the check for zone inclusion, which is expensive $O(n^4)$.

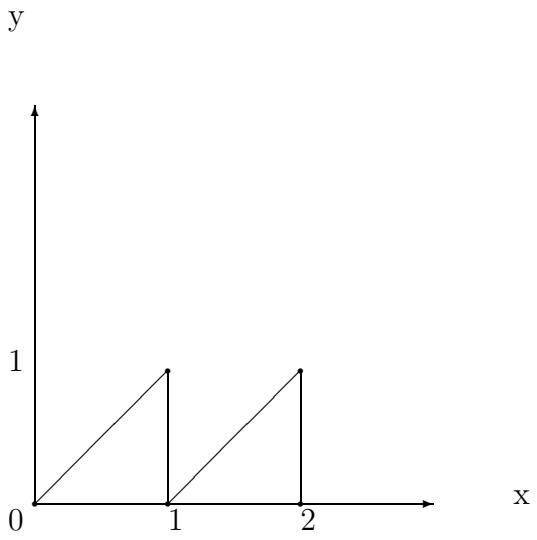

Figure 4.4: Non-convex union of regions example

Figure 4.4 shows a non-convex union of regions which includes the areas of the two triangles.

To efficiently represent both convex and non-convex regions, other data structures called Clock Difference Diagrams (CDDs) [LPWY99, BLP+99], Clock-Restriction Diagram (CRDs) [Wan03] and and Difference Decision Diagrams (DDDs) [MLAH99b, MLAH99d, MLAH99c, MLAH99a, ML98] have been proposed. Of these, only DDDs are open source.

In this thesis, we only consider two data structures for the representation of timing information: DBMs and DDDs. More information about these data structures is given in Section 4.3.

4.2 Model Checking for Untimed Probabilistic Systems

The need for probabilistic modelling has been advocated in several papers. Probability enables the modelling of phenomena related to reliability and performance. Verification of quality of service through model checking has already been applied to real-world case studies. For reliability and performance analysis, there are two kinds of models, discrete and continuous. In the case of the discrete models, several models of distributed probabilistic systems are based on Markov decision processes [BdA95, BK98] or on closely related formalisms.

Traditional model checking involves verifying properties of labelled state transition systems. In the case of probabilistic model checking, however, the models also incorporate information about the likelihood of transitions occurring between states.

In this section, we first briefly overview some basic concepts relating to Markov decision processes which will serve as semantics of probabilistic timed automata.

4.2.1 Markov Decision Processes

Nondeterminism is not only useful for modelling concurrency but also useful when the exact probability of a transition is not known, or when it is known but not considered relevant. A Markov decision process (MDP) allows one to describe both nondeterministic and probabilistic behaviour. It also allows us to describe the behaviour of a number of probabilistic systems operating in parallel.

Definition 4.2.1 *An MDP is a tuple* $(S, \overline{s}, Steps, L)$ *where:*

- S *is a set of states,*
- $\overline{s} \in S$ *is the initial state,*
- *a transition function* $Steps : S \to 2^{Dist(S)}$ *assigning each state* $s \in S$ *to a finite, non-empty subset of* $Dist(S)$*,*
- *a labelling function* $L : S \to 2^{AP}$ *assigning to each state the set of atomic propositions that are true in that state.*

For a given state $s \in S$, $Steps(s)$ is a set of nondeterministic choices available in that state. Each nondeterministic choice is a probability distribution.

A path in the MDP is a non-empty finite or infinite sequence of transitions $\omega = s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} s_2 \ldots$ where $s_i \in S$, $\mu_{i+1} \in Steps(s_i)$ and $\mu_{i+1}(s_{i+1}) > 0$ for all $i \geq 0$. For a path ω and $i \in \mathbb{N}$, we denote by $\omega(i)$ the $(i+1)$th state of ω, and by $last(\omega)$ the last state of ω if ω is finite. The sets of all finite and infinite paths starting in state s are denoted $Path_{fin}(s)$ and $Path_{ful}(s)$, respectively. Because of the introduction of nondeterminism, both the nondeterministic and probabilistic choices have to be resolved when a path of an MDP is traced. The nondeterministic choices are assumed to be made by an adversary (also known as a 'scheduler' or 'policy'), which selects a choice based on the history of choices made so far. Formally, an adversary A is a function mapping every finite path ω_{fin} of the MDP onto a distribution $A(\omega_{fin}) \in Steps(last(\omega_{fin}))$. We denote by $Path^A_s$ the subset of paths from s which corresponds to adversary A. The set of all adversaries for a given MDP $\mathcal{M}$ is denoted $Adv_{\mathcal{M}}$. The sets of all finite and infinite paths starting in state s under an adversary A is denoted $Path^A_{fin}(s)$ and $Path^A_{ful}(s)$, respectively. To reason about the probabilistic behaviour of the MDP under an adversary A, we need to determine the probability with which certain paths are taken. This is achieved by defining, for each state $s \in S$, a probability measure $Prob^A_s$ over $Path^A_{ful}(s)$. More precisely, for an MDP $(S, \overline{s}, Steps, L)$ and state $s \in S$, under a given adversary A, the behaviour of the MDP is purely probabilistic and can be given by a transition probabilistic matrix $\mathbf{P}^A_s$ over finite paths which is defined as below. For two finite paths $\omega_{fin}, \omega'_{fin} \in Path^A_{fin}(s)$:

$$\mathbf{P}^A_s(\omega_{fin}, \omega'_{fin}) = \begin{cases} \mu(s') & \text{if } \omega'_{fin} \text{ is of the form } \omega_{fin} \xrightarrow{\mu} s' \text{ and } A(\omega_{fin}) = \mu \\ 0 & \text{otherwise,} \end{cases}$$

and $L^A_s(\omega_{fin}) = L(last(\omega_{fin}))$ for each $\omega_{fin} \in Path^A_{fin}(s)$. We can define a probability measure $Prob^A_s$ over $Path^A_{ful}(s)$ using the construction given in [BK98].

Probabilistic Reachability

Definition 4.2.2 *Let $PS = (S, \overline{s}, Steps, L)$ be an Markov decision processes. Then the* reachability probability *with which a set $F \subseteq S$ of target states, can be reached from a*

state $s \in S$, for an adversary $A \in Adv_{PS}$, is:

$$ProbReach^A(s, F) \stackrel{\text{def}}{=} Prob_s^A\{\omega \in Path_{ful}^A \mid \omega(0) = s \,\&\, \exists i \in \mathbb{N} \,.\, \omega(i) \in F\} \,.$$

Furthermore, the maximal *and* minimal *reachability probabilities are defined respectively as*

$$\begin{aligned} MaxProbReach_{PS}(s, F) &\stackrel{\text{def}}{=} \sup_{A \in Adv_{PS}} ProbReach^A(s, F) \\ MinProbReach_{PS}(s, F) &\stackrel{\text{def}}{=} \inf_{A \in Adv_{PS}} ProbReach^A(s, F) \end{aligned}$$

4.2.2 Probabilistic Computation Tree Logic (PCTL)

To write specifications of MDPs, we use the logic PCTL. The syntax of PCTL is as follows:

$$\phi \;::=\; true \;\big|\; a \;\big|\; \neg\phi \;\big|\; \phi \wedge \phi \;\big|\; \mathcal{P}_{\bowtie p}[\phi \,\mathcal{U}\, \phi] \big|\; \mathcal{P}_{\bowtie p}[\phi \,\mathcal{V}\, \phi]$$

where a is an atomic proposition, $\bowtie \in \{\leq, <, \geq, >\}$, $p \in [0, 1]$.

The meaning of a state s satisfying the formula $\mathcal{P}_{\bowtie p}[\psi]$ is that the probability of a path from s satisfying ψ is in the range $\bowtie p$ *for all adversaries*.

We use the abbreviation $\diamond\phi$ for $true \,\mathcal{U}\, \phi$ inside PCTL formula. For example, the formula for probabilistic reacahbility is defined as:

$$\mathcal{P}_{\bowtie p}[\Diamond\phi] \;\stackrel{\text{def}}{=}\; \mathcal{P}_{\bowtie p}[true \,\mathcal{U}\, \phi]$$

.

We use formula $\mathcal{P}_{\gtrsim p}[\Diamond\phi]$ for specifying minimum probabilistic reachability and formula $\mathcal{P}_{\lesssim p}[\Diamond\phi]$ for specifying maximum probabilistic reachability.

PCTL can be used to express properties such as: 'with probability less than 0.94, a leader is elected', which is represented as the formula $\mathcal{P}_{<0.94}[\Diamond\ elected]$, where *elected* is an atomic proposition labelling the state of elected leader.

Definition 4.2.3 *Let $\mathcal{M} = (S, \overline{s}, Steps, L)$ be a labelled MDP. For any state $s \in S$, the*

satisfaction relation $s \models \phi$ is defined inductively as follows:

$$\begin{aligned}
s &\models \mathit{true} && \text{for all } s \in S \\
s &\models a && \Leftrightarrow \quad a \in L(s) \\
s &\models \neg\phi && \Leftrightarrow \quad s \not\models \phi \\
s &\models \phi_1 \wedge \phi_2 && \Leftrightarrow \quad s \models \phi_1 \text{ and } s \models \phi_2 \\
s &\models \mathcal{P}_{\bowtie p}[\psi] && \Leftrightarrow \quad p_s^A(\psi) \bowtie p \text{ for all } A \in Adv_{\mathcal{M}}
\end{aligned}$$

where for any adversary $A \in Adv_{\mathcal{M}}$:

$$p_s^A(\psi) \stackrel{\text{def}}{=} Prob_s^A(\{\omega \in Path_s^A \mid \omega \models \psi\})$$

and for any path $\omega \in Path_{ful}$:

$$\omega \models \phi_1 \ \mathcal{U} \ \phi_2 \quad \Leftrightarrow \quad \exists i \in \mathbb{N}.(\ \omega(i) \models \phi_2 \text{ and } \forall j < i \ .\ \omega(j) \models \phi_1)$$

$$\omega \models \phi_1 \ \mathcal{V} \ \phi_2 \quad \Leftrightarrow \quad \forall i \in \mathbb{N}.(\forall j < i.(\omega(j) \not\models \phi_1 \Rightarrow \omega(i) \models \phi_2))$$

PCTL Model Checking The model checking algorithm for PCTL over MDPs takes as input a labelled MDP $(S, \bar{s}, \mathit{Steps}, L)$ and a PCTL formula ψ and returns the set of states $\mathit{Sat}(\psi) \subseteq S$ of states satisfying the formula.

For the formula $\mathcal{P}_{\bowtie p}[\psi]$, it need to determine whether $p_s^A(\psi)$ satisfies the bound $\bowtie p$ for all adversaries A. This is done by computing either the *minimum* or *maximum* probability over all adversaries, depending on whether the relational operator $\bowtie$ defines an upper or lower bound. If $\bowtie$ is $\leq$ or $<$, then:

$$\mathit{Sat}(\mathcal{P}_{\bowtie p}[\psi]) \quad = \quad \{s \in S \mid p_s^{max}(\psi) \bowtie p\}$$

and if $\bowtie$ is $\geq$ or $>$, then:

$$\mathit{Sat}(\mathcal{P}_{\bowtie p}[\psi]) \quad = \quad \{s \in S \mid p_s^{min}(\psi) \bowtie p\}$$

where:

$$p_s^{max}(\psi) \stackrel{\text{def}}{=} \sup_{A \in Adv_{\mathcal{M}}} [p_s^A(\psi)]$$
$$p_s^{min}(\psi) \stackrel{\text{def}}{=} \inf_{A \in Adv_{\mathcal{M}}} [p_s^A(\psi)] .$$

In this thesis, we use PRISM as the model checker for PCTL against MDPs. Table 4.1 lists the main procedures in PRISM used by our software tool.

Table 4.1: PRISM procedure

PRISM Procedure	Description
PRISM.parseModelFile	parse a PTA model
PRISM.parsePropertiesFile	parse a PCTL formulae
PRISM.buildModel	return an MTBDD representation of parsed model
PRISM.modelCheck	return maximum probability for a PCTL formula of $\mathcal{P}_{\lesssim p}[\Diamond\phi]$ against an given model

4.3 Data Structures

In this section, we introduce data structures used in this thesis for representing both timing information and probability distributions.

4.3.1 Data Structures for Encoding Timing Information

To solve the reachability problem for a timed system, one issue that has to be addressed is how to represent the infinite state space $\mathbb{R}$ of a timed system. Two data structures are discussed here, and especially their common principles, the main differences in their implementation choices, and the trade offs made. The first data structure considered is the Difference Bound Matrices (DBMs), whereas the second is the Difference Decision Diagrams (DDDs).

The two data structures have some common features. For example, both assume a special clock x_0 whose value is always zero. Recall that there are two kinds of constraints

encountered in regions. One of them is the constraint that involves only one clock, and the other is that representing bounds on the differences between two different clocks. For constraints involving only one clock, these constraints can also be represented as bounds on the difference between two clock values through using the special clock x_0 whose value is always zero. For example, X is a set of clocks and for $x_i \in X$, $x_i < c$ (c is a integer) is expressed as $x_i - x_0 < c$, while $c < x_i$ as $x_0 - x_i < c$. Thus, the constraints can be uniformly represented as the difference between two of clocks.

To solve the reachability problem for a timed system, the other issue that has to be addressed is how to perform the basic verification operations (resetting clocks, advancing time, etc.) on this representation to compute the reachable state space. The perspectives to be considered about the data structures are: operations on them, canonicity, computational complexity, space consumption and ordering sensitivity.

Although the two data structures considered here for representing timed information provide different solutions for representing timing constraints, both provide representation for continuous timing data, as well as operations for manipulating clock advance and resetting. They differ in their space consumption, computational complexity, canonicity and ordering sensitivity.

Below we describe Difference Bound Matrices and Difference Decision Diagrams and their implementation of corresponding basic operations. Table 4.2 gives the uniform basic operations used in this thesis.

Other uniform operations, such as *Emptiness*, *Inclusion*, *Exists*, *Equivalence* and *Normalise* are also described below. In this thesis, these uniform operations are also extended to the symbolic states if it would not cause any confusion. We will also abuse "( )" for enclosing a state or a symbolic state. An enumeration of set elements is enclosed in "{ }".

Difference Bound Matrices

Difference Bound Matrices are a data structure that can efficiently represent sets of adjacent regions (convex union of adjacent regions). A DBM is actually a square matrix whose elements represent bounds on the difference between two clock values. For a set of n

Table 4.2: Uniform basic operations

Uniform Operation	Basic Zone Operation
$TimeSuccessors$	$\nearrow \zeta$
$TimePredecessors$	$\swarrow \zeta$
$ResetClocks$	$\big(\bigwedge_{x \in Y} (x = 0)\big)$
$FreeClocks$	$\zeta_{/Y}$
$until$	$\swarrow_{\zeta'} \zeta$
$Conjunction$	$\cap$
$Disjunction$	$\cup$
$Complement$	$\bar{\zeta}$

clocks $\{x_1, \cdots, x_n\}$, using the special clock x_0 whose value is always zero, the constraints over these clocks can be encoded as a $(n+1) \times (n+1)$ square matrix D whose indices range over the interval $[0..n]$ and whose elements belong to the pair $(\mathbb{Z}_\infty, \bowtie)$, where $\bowtie$ is either "$<$" or "$\leq$". In DBMs, "$<$" and "$\leq$" are ordered such that "$<$" $<$ "$\leq$". Z are ordered as normal, and $\forall d \in Z$, $d < \infty$. For two pairs $D_1 = (d_1, \bowtie_1)$ and $D_2 = (d_2, \bowtie_2)$, we denote $min(D_1, D_2) = (min(d_1, d_2), min(\bowtie_1, \bowtie_2))$ where $d_1, d_2 \in \mathbb{Z}_\infty$ and $\bowtie_1, \bowtie_2 \in \bowtie$.

The matrix D has the following properties.

• All the elements on the main diagonal have the form $(0, \leq)$.

• The elements on the first column of D encode the upper bounds of the clocks. That is, if $x_i - x_0 \bowtie c$ appears in the constraint, then D_{i0} is the pair $(c, \bowtie)$; otherwise it is $(\infty, <)$.

• The elements on the first row of D encode the lower bounds of the clocks. If $x_0 - x_i \bowtie c$ appears in the constraint, D_{0i} is $(c, \bowtie)$; otherwise it is $(0, \leq)$.

• Other elements D_{ij} (i-th row and j-th column) are the pairs $(c, \bowtie)$ which encode the constraint $x_i - x_j \bowtie c$. If there is no explicit constraint on the difference between x_i and x_j in the conjunction, the element D_{ij} is set to $(\infty, <)$.

In DBMs, an element corresponds to an atomic zone. An element of pair $(\infty, <)$ is called *trivial element*, an element of pair rather than $(\infty, <)$ is called *non-trivial element*.

$$D_1 = \begin{bmatrix} (0,\leq) & (0,\leq) & (0,\leq) \\ (1,\leq) & (0,\leq) & (\infty,<) \\ (1,\leq) & (0,\leq) & (0,\leq) \end{bmatrix} \quad D_2 = \begin{bmatrix} (0,\leq) & (-1,\leq) & (0,\leq) \\ (2,\leq) & (0,\leq) & (\infty,<) \\ (1,\leq) & (-1,\leq) & (0,\leq) \end{bmatrix}$$

Figure 4.5: Two DBMs representing the non-convex union of regions in Figure 4.4

A DBM cannot directly represent non-convex unions of regions or zones. In practice, they can be represented through using lists of DBMs.

Canonicity

A canonical data structure makes testing of functional properties, for example, equivalence straightforward. In this subsection, we discuss the canonicity properties and alternative way to check equivalence if the data structure does not have a canonical form.

Every region or zone (convex union of regions) can be represented by a DBM. However, many different DBMs can represent the same clock zone because some difference constraints between two different clocks do not appear explicitly and some of the bounds may not be tight enough. Although there are more than one DBM representing the same zone, there exists a unique matrix, called the canonical DBM, that encodes the same zone and whose elements are tightest. The canonical DBM can be obtained by applying the Floyd-Warshall's algorithm [CLR90]. A canonical representative has two important properties, namely it guarantees the existence of the following:

- a simple method for determining whether a conjunction of zones has a solution, and
- a simple method for checking whether two matrices represent the same zone.

However, because DBMs cannot directly represent non-convex unions of regions, which are actually represented through using lists of DBMs in practice, it is inefficient to check whether two sets of regions or lists of regions are same when the number of DBMs for representing the timing information becomes very large. The tool KRONOS uses some heuristics to check whether the union of two DBM's is indeed a DBM, but such heuristics are computationally expensive.

Complexity

In general, the cost of operations on a $n \times n$ matrix for a DBM would have time-complexity $O(n^2)$ where n is the number of clocks. However, certain operations such as time successor require a DBM in canonical form. This canonical form is computed by applying Floyd-Warshall's algorithm [CLR90]. Computation of this canonical form is the most costly operation on DBMs with time-complexity $O(n^3)$ where n is the number of clocks.

Ordering Sensitiveness

A DBM is just a matrix. There is no assumption about ordering on this matrix, and thus ordering does not have any effect on DBM.

Space Consumption

For a single DBM, the size of the DBM is bounded. Each DBM encoding convex constraints requires $O(n^2)$ memory space, where n is the number of clocks including the extra clock. As a DBM is just a data structure for representing adjacent regions, there is no sharing between DBMs. Thus, the total space required is $O(k \times n^2)$ where k is the number of the DBMs. However, the method [LLPY97] can be used to reduce the memory requirement. Figure 4.5 shows the use of 2 DBMs to represent the non-convex unions of regions in Figure 4.4 which requires 18 matrix elements in DBMs.

Operations

In the following, we assume that D is a list of DBMs, $D1$, $D2$ and $D3$ are DBMs, $\bar{x} = (x_0, x_1, \ldots, x_n)$ is a set of $n+1$ clocks and D_{ij},$D1_{ij}$ and $D2_{ij}$ are elements at row i and column j in the matrices.

$Conjunction(D1, D2)$**.** Given two DBMs $D1$ and $D2$, $D3$=$Conjunction(D1, D2)$ is such that for all $0 \leq i, j \leq n$, $D3_{ij} = min(D1_{ij}, D2_{ij})$.

$TimeSuccessors(D1)$**.** This operation requires a DBM in a canonical representative form. As time elapses, clock differences remain the same, since all clocks increase at

the same rate. Lower bounds do not change either since there are no decreasing clocks. Upper bounds have to be pushed to infinity, since an arbitrary period of time may pass. Recall that the elements on the first column encode the upper bounds on each clock. Thus, the only elements that need to be changed are those on the first column of the matrix: $D1_{ij} = (\infty, <)$ if $j = 0$.

ResetClocks$(D1, x_i, d)$**.** Resetting a clock to d can be implemented by changing elements below in the matrix D_1:
$D1_{ij} = (\infty, <)$ if $i \neq j$
$D1_{ji} = (\infty, <)$ if $i \neq j$
$D1_{i0} = (-d, \leq)$
$D_{0i} = (d, \leq)$

FreeClocks$(D1, x_i)$**.** A free clock can be implemented by changing elements below in the matrix D_1:
$D1_{ij} = (\infty, <)$ if $i \neq j$
$D1_{ji} = (\infty, <)$ if $i \neq j$
$D1_{i0} = (0, \leq)$
$D_{0i} = (\infty, <)$

TimePredecessors$(D1)$**.** This operation requires a DBM in a canonical representative form. Recall that the elements on the first row encode the lower bounds on each clock. Thus, the only elements that need to be changed are those on the first row of the matrix: $D1_{ij} = (0, \leq)$ if $i = 0$. The meaning of this operation is to push the lower bounds to 0.

Disjunction$(D1, D2)$**.** The disjunction of two DBMs is not necessarily a DBM, since the disjunction of two constraints can be non-convex. In practice, the disjunction of two DBMs is represented as a list of DBMs.

Complement$(D1)$**.** The complement of a DBM is a list of DBMs $D_1, \ldots, D_k$; each D_x is the complement of a non-trivial element of $Dx_{i,j}$, for $1 \leq x \leq k$.

Emptiness$(D1)$**.** For a single DBM, the empty set check can be done through checking its diagonal elements in its canonical form; it returns **true** if $D1_{i,j}$ is $(\leq, 0)$, and otherwise **false**. For a list of DBMs, the empty set check can be done through checking each DBM on the list.

Inclusion$(D1, D2)$**.** For two single DBMs in canonical form, the operation returns **true** if $D1_{i,j}$ is less than or equal to $D2_{i,j}$ for any i, j, otherwise **false**. For two lists of DBMs, the set inclusion testing is reduced to checking $Emptiness(Conjunction(Complement(D1), D2))$.

Equivalence$(D1, D2)$**.** For two single-DBMs, the equivalence check can be done through first bringing them into canonical forms and then checking each corresponding element in both matrices. For two lists of DBMs, it is reduced to checking set inclusion in both directions.

$Normalise(D1, k)$**.** In forward search, the algorithm does not always terminate. The $Normalise$ operator ensures the termination of forward search. It can be computed from the canonical form of $D1$ by setting $D1_{i,j} = (\infty, <)$ if $value(D1_{i,j}) > k$ or setting $D1_{i,j} = (-k, <)$ if $value(D1_{i,j}) < -k$.

Exists$(x_i, D1)$**.** The *Exists* operator removes all clock constraints related to clock x_i. It can be computed from $D1$ simply by removing i-th and i-th row and column from $D1$.

until(ζ_1, ζ_2) The DBM-based *until*(ζ_1, ζ_2) operation needed in the backward algorithm can be computed using the result from [Tri98], in which the *until* operation gives the set that can reach ζ_2 via ζ_1 by letting time pass:

$$until(\zeta_1, \zeta_2) = \zeta_1 \cap (\swarrow \zeta_2) \setminus (\swarrow ((\swarrow \zeta_2 \setminus (\zeta_1 \cup \zeta_2)))$$

assuming ζ_2 is convex.

If the second argument of *until* is non-convex, a more general result showing that *until* is union-distributive on its second argument can be exploited as follows: $until(\zeta_1, \zeta_2 \cup \zeta_2') = until(\zeta_1, \zeta_2) \cup until(\zeta_1, \zeta_2')$ assuming both ζ_2 and ζ_2' is convex.

However, in order to compute *until* (also *tpre*), two complementations are required because the set difference operation is computed through complementation. Unfortunately, complementation is an expensive operation and causes the generation of non-convex zones.

Difference Decision Diagrams

The data structure called Difference Decision Diagrams (DDDs) is a symbolic data structure designed to efficiently represent both convex and non-convex unions of zones, called Difference Constraint Expressions (DCEs). DDDs are based on Binary Decision Diagrams (BDDs). Like BDDs, a DDD is also a Directed Acyclic Graph (DAG) due to sharing of isomorphic sub-graphs. The vertex set contains two terminals 0 and 1 of out-degree zero, and a set of non-terminal vertices of out-degree two. A non-terminal vertex v corresponds to an integer- or real-valued difference constraint expression between two clocks. A non-terminal vertex v in DDDs is a tuple $(v(x), v(y), v(op), v(constant))$ where $v(x)$ and $v(y)$ are variables ranging over $\mathbb{R}$, $v(op) \in \{<, \leq\}$ and $v(constant) \in \mathbb{Z}$. Unlike in a BDD, the same pair of clocks can appear more than once along the path in a DDD. A path in a DDD is a finite sequence of edges. A path corresponds to a conjunction of difference constraints that is called a Difference Constraint System (DCS) in DDDs. A DCS corresponds to a DBM. A DDD contains a representation of DBMs as a special case. A path that ends with true or false is called a 1-path or 0-path respectively. A path is feasible if and only if the corresponding Difference Constraint System (a DBM) has a solution. If the Difference Constraint System has no solution, the path is infeasible. Unlike BDD, both 0- and 1-paths can be feasible or infeasible because the difference constraints can interact with each other along the path.

A DBM has no requirement for ordering. However, DDDs require ordering. In DDDs, the total ordering of both two single clocks and two pairs of different clocks complies to the following:

- Assuming that the variables $x_1, \cdots, x_n$ can be ordered.
- Two pairs of variables, which are ordered such that $(x_i, x_j) < (y_i, y_j)$ if and only if $x_j < y_j$ or $(x_j = y_j$ and $x_i < y_i)$

We say the ordering on pairs of variables (x_i, x_j) of a vertex in a DDD is *normalised* if $x_i > x_j$.

We assume that n variables $x_1, \ldots, x_n$ have the following ordering: $x_1 < x_2 < \cdots < x_n$, where $x_i \in \mathbb{R}^+$ and $1 \leq i \leq n$. The n variables can have at most $n(n-1)/2$ pairs of *normalised* ordering of the form (x_i, x_j), which are ordered as follows:

$$
\begin{aligned}
&(x_2, x_1) < (x_3, x_1) < (x_4, x_1) < \ldots \ (x_n, x_1) < \\
&(x_3, x_2) < (x_4, x_2) < \ldots \ (x_n, x_2) < \\
&\vdots \\
&(x_{n-1}, x_{n-2}) < \\
&(x_n, x_{n-1})
\end{aligned}
$$

In DDDs, the ordering of two non-terminal vertices u and v is defined according to the following:

- The "$<$" $<$ "$\leq$".
- $u(constant)$ and $v(constant)$ are ordered as in $\mathbb{Z}$.
- $u < v$ if $(u(x), u(y))$ and $(v(x), v(y))$ are normalised and $(u(x), u(y)) < (v(x), v(y))$ and $u(op) < v(op)$ and $u(constant) < v(constant)$.

In DDDs, all non-terminal vertices are less than the terminal vertices 0 and 1.

Canonicity

Like BDDs, DDDs can be ordered and reduced. In DDDs, there are two kinds of reductions that can be used: local reduction and path reduction. The path-reduced DDD is also called a semi-canonical form of DDDs, which is obtained by removing the infeasible paths. A semi-canonical form of DDDs is sufficient for deciding satisfiability and tautology. However, both local reduction and path reduction on DDDs do not provide a canonical representation of difference constraint expressions. In other words, there is no canonical form for DDDs. Although DDDs cannot be brought into canonical from, DDDs support BDD-like function (Apply) for testing of bi-implication that can be used to test equivalence. Given two DDDs, if the result of bi-implication is always true or tautology after applying the path reduction then this means the two DDDs are equivalent.

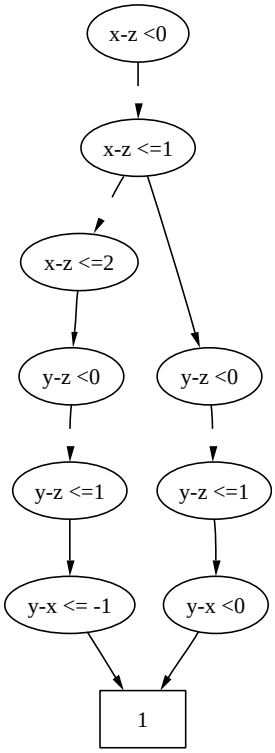

Figure 4.6: A ordered DDD representing the non-convex union of regions in Figure 4.4

Complexity

The worst-case running time for some most important algorithms in DDDs is PSPACE-complete.

Ordering Sensitiveness

Like BDDs, DDDs are also sensitive to ordering. Similarly to BDDs, the structure of DDDs will change if two levels of DDDs are swapped. Figure 4.6, which is the DDD representation of the non-convex unions of regions in Figure 4.4, shows a DDD using of variables ordering of $z < x < y$.

Space Consumption

For DDDs, the size is not bounded by the number of variables because there can be arbitrarily many tests on the same pair of variables. Similarly to BDDs sharing of sub-structures, a DDD representation of a union of zones is likely to be much smaller than the explicit DBM-list representation. Figure 4.6, the DDD representation of the non-convex unions of regions in Figure 4.4, requires 10 DDD nodes.

Operations

The algorithms for the DDD data structure are similar to those of Binary Decision Diagrams (BDDs). DDDs support a BDD-like function (Apply) that can be instantiated to yield all common Boolean operations. Similarly to BDDs, all operations assume that DDDs are ordered. However, it is expensive (the computational complexity for it is PSPACE-complete) to construct and maintain a data structure for operations such as time successor and clock reset.

Below we list the DDD operations. In the following, we assume that D, $D1$ and $D2$ are DDDs, x, y, z are real variables, the value of z is always zero and c is a constant.

- Function Creates$(x, c, \bowtie)$ returns the DDD representing $x \bowtie c$.
- Function Not$(D1)$ returns the complementation of a DDD.

- Function Applying($op, D1, D2$) returns the DDD representing $D1$ *op* $D2$ if *op* is a binary operation, for example, '|' and '&', which are for operations 'or' and 'and' on sets; or return the **true** or **false** if *op* is a logic operation, for example, $\Leftrightarrow$, $\Rightarrow$ etc.
- Function Exists(x, D) returns the DDD by performing the existential quantification on the real variable x.
- Function PathReduce(D) returns the DDD by removing the infeasible paths.

Below we list the operations needed for the model checking algorithms through operations on DDDs.

$Conjunction(D1, D2)$. Intersection of two DDDs is computed by Applying($\wedge, D1, D2$).

$TimeSuccessors(D1)$. The direct version of this operation in DDD involves two additional variables and requirement of calling Exists. For the consideration of performance, the $TimeSuccessors$ operator in DDDs can be implemented by transforming a DDD into a DBM, then applying the corresponding $TimeSuccessors$ operation for DBMs and finally transforming back to DDDs.

$ResetClocks(D1, x, c)$. This operation is computed by
Applying($\wedge$, Exists($x, D1$), Creates($x, c, =$)).

$FreeClocks(D1, x)$. This operation is computed by Exists($x, D1$).

$TimePredecessors(D1)$. This operation is computed by
Applying($\wedge$, Exists($z, D1$), Creates($z, 0, \geq$)).

$Disjunction(D1, D2)$. The union of two DDDs is computed by:
Applying($\vee, D1, D2$).

$Complement(D1)$. Complementing is a straitforward DDD operation Not($D1$).

$Emptiness(D)$. This operation is computed by Applying($\Leftrightarrow$, PathReduce(D), 0).

Inclusion$(D1, D2)$**.** The set inclusion testing is computed by:
Applying$(\Rightarrow, D1, D2)$.

Equivalence$(D1, D2)$**.** DDD has no canonical from. DDDs support BDD-like function (Apply) for testing of bi-implication that can be used to test equivalence. It is computed by Applying$(\&,$ Applying$(\Rightarrow, \mathsf{D1}, \mathsf{D2}),$ Applying$(\Rightarrow, \mathsf{D2}, \mathsf{D1}))$.

Normalise$(D1, k)$**.** Although the forward search algorithm does not always terminate, it has the nice property that is the zones generated are convex. A convex zone in DDDs has only one path which corresponds to a DBM. The *Normalise* operator in DDD can be implemented by transforming a DDD into a DBM, then applying the corresponding operation for DBMs and finally transforming back to DDDs.

Exists$(x, D1)$**.** This is obtained by DDD's operation Exists$(x, D1)$.

Implementation of *until*(ζ', ζ) **in DDDs.** By introducing an extra clock z, which is alway zero, the formula for *until*(ζ', ζ) can be transformed into: *until*(ζ', ζ)=$\exists z'.(\zeta \wedge (z' \leq z) \wedge \forall z''.((z' \leq z'' \leq z) \Rightarrow (\zeta \vee \zeta')[z''/z]))[z'/z]$. In the implementation of *tpre* in DDDs, z' and z'' can be treated as two extra clocks which can then be eliminated by existential quantification. As the BDD-like data structure is very sensitive to the number of variables and the variables' ordering, introducing two extra clocks z' and z'' means a further increase of the domain of DDDs. Using duality between universal and existential quantification, i.e.: $\forall x.\phi = \neg\exists x.\neg\phi$, the above formula can be further transformed into: *until*(ζ', ζ)=$\exists z'.(\zeta \wedge (z' \leq z) \wedge \neg(\exists z''.(\neg((z' \leq z'' \leq z) \Rightarrow (\zeta \vee \zeta'))[z''/z])))[z'/z]$. Hence, the *until*$(\zeta', \zeta)$ operation can be performed in terms of logical operations of complementation, set union and existential quantification.

Operations for Post and Pre

Having introduced the basic operations on DBMs and DDDs, we can now define the operations needed to implement the forward and backward search algorithms.

$dpost((s,\zeta),e,k)$	
1.	$\phi_1 = Conjunction(I_s,\zeta)$
2.	$\phi_2 = Conjunction(\phi_1,g)$
3.	$\phi_3 = FreeClock(\phi_2,X)$
4.	$\phi_4 = ResetClock(\phi_3,X)$
5.	$\phi_5 = Conjunction(\phi_4,I'_s)$
6.	$\zeta' = Normalise(\phi_5,k)$
7.	**if** $Emptiness(\zeta') =$ **false**
8.	**return** (s',ζ')
9.	**else**
10.	**return** $\emptyset$

Figure 4.7: The *dpost* operation

$post((s,\zeta),e,k)$	
1.	$\phi_1 = TimeSuccessor(\zeta)$
2.	**return** $dpost((s,\phi_1),e,k)$

Figure 4.8: The *post* operation

$post((s,\zeta),e,k)$**.** Let e denote a transition from the current node s to s' with enabling clock constraints g and reset clocks in set X, and let (s,ζ) denote the current state and its invariant I_s, s' next node and I'_s be its invariant, where k is the maximal constant appearing in the system or specification.

The operation *dpost* that computes the set of states reachable from (s,ζ) via the discrete transition e is shown in Figure 4.7, which is the implementation of operation *disc–succ*$(e,(l,\zeta))$ described in Section 4.1.4.

$dpre((s',\zeta'),e)$**.** Let e denote a transition from the current node s to s' with enabling clock constraints g and reset clocks in set X, and let (s,ζ) denote the current state and its invariant I_s, s' next node and I'_s be its invariant.

$dpre((s', \zeta'), e)$	
1.	$\phi_1 = ResetClock(\zeta', X)$
2.	$\phi_2 = FreeClock(\phi_1, X)$
3.	$\phi_3 = Conjunction(\phi_3, g)$
4.	$\zeta = Conjunction(I_s, \phi_3)$
5.	**if** $Emptiness(\zeta) =$ **false**
6.	**return** (s, ζ)
7.	**else**
8.	**return** $\emptyset$

Figure 4.9: The *dpre* operation

The operation *dpre* that computes the set of states that can reach (s', ζ') via the discrete transition e is shown in Figure 4.9, which is the implementation of operation *disc–pred*$(e, (l', \zeta'))$ described in Section 4.1.4.

4.3.2 Data Structure for Encoding Probability Information

Multi-terminal Binary Decision Diagrams (MTBDDs) [CFM+93] are an extension of BDDs. MTBDDs extend BDDs by allowing them to represent functions which can take any value, not just 0 or 1. In other words, BDDs can have only two terminals, while MTBDDs can have more than two terminals. In the case of analysis of probabilistic properties of systems, numerical computation is required. BDDs are in fact a special case of MTBDDs. An MTBDD is a DAG (Directed Acyclic Graph). Like BDDs, the vertices of the graph are called nodes. There are two kinds of nodes in an MTBDD, the non-terminal nodes and terminal nodes. As in BDDs, a non-terminal node is labelled with a single variable and each non-terminal node has exactly two children. However, unlike BDDs, the terminal node which has no children is labelled by a real number. MTBDDs can be reduced to canonical form by imposing ordering among nodes. However, similarly to BDDs, the size of MTBDDs is extremely sensitive to the ordering of its variables.

In Markov decision processes (MDPs), the probability distributions of transitions are

defined as vectors and matrices. One of the most interesting applications of an MTBDD is its ability to represent both vectors and matrices. As far as numerical computation is concerned, MTBDDs support methods for implementing standard matrix operations, such as scalar multiplication, matrix addition and matrix multiplication. First, consider how to use an MTBDD to encode a vector. Given a real-valued vector of length 2^n, an MTBDD encoding can be obtained by mapping to reals from vector's indices which are encoded into n Boolean variables. A vector $\underline{\mathsf{v}}$ of length 2^n is a mapping from indices to reals, where $\underline{\mathsf{v}}(i) \in R$ and $0 \leq i \leq 2^n-1$. We use $enc : \{0, \ldots, 2^n-1\} \rightarrow \mathbb{B}^n$ to denote the vector's indices which are encoded into n Boolean variables. We use $f_v[\underline{\mathsf{x}} = enc(i)] = \underline{\mathsf{v}}(i)$ for $0 \leq i \leq 2^n-1$ to represent a vector $\underline{\mathsf{v}}$. The idea to use a MTBDD to encode a vector can be extended to matrices. A 2^n by 2^n matrix can be treated as $2^n \times 2^n$ vectors. Thus, a 2^n by 2^n matrix can be encoded by using $2n$ Boolean variables, n of which encode row indices and n of which encode column indices. If the size of the vectors and matrices is not the power of two, we can still use the method above to encode a MTBDD simply by padding vectors or matrices with extra elements which are set to 0. A $2^n \times 2^n$ matrix $\mathbf{M}$ is denoted as $f_{\mathbf{M}}[\underline{\mathsf{x}} = enc(i), \underline{\mathsf{y}} = enc(j)] = \mathbf{M}(i,j)$ for $0 \leq i,j \leq 2^n-1$. Figure 4.10 and 4.11 show an example of a matrix and its corresponding reduced MTBDD. The example is a 4×4 matrix, so we can use two row Boolean variables (x_1, x_2) and two column ones (y_1, y_2) to encode the corresponding MTBDD. For example, if the Boolean value of variables (x_1, x_2) and (y_1, y_2) is (0,1) and (0,0) respectively, then this means the row index is 1 and column index is 0. This corresponds to the matrix entry (1,0)=2.

Below we list the BDD and MTBDD operations needed for the symbolic algorithm introduced in Chapter 8. In the following, we assume $\mathbf{M}$ is an MTBDD, and $\underline{\mathsf{x}}, \underline{\mathsf{y}}, \underline{\mathsf{z}}$ are the Boolean vectors which correspond to the MTBDD variables for row, column and non-deterministic choice in an MDP matrix.

- Function Apply$(op, \mathbf{M_1}, \mathbf{M_2})$ returns the MTBDD representing $\mathbf{M_1}$ *op* $\mathbf{M_2}$ where *op* is a binary operation over reals (e.g. $+, -, \times, \div$, etc). In this thesis, we use operations $\times$ *and* $+$ for operations *op* ($\times$ and $+$) when two MTBDDs operate over the reals. We use operations $\vee$, $\wedge$ and $\setminus$ for operations 'or', 'and' and 'difference' on sets when $\mathbf{M_1}$ and $\mathbf{M_2}$ are BDDs.

$$M = \begin{bmatrix} 0 & 4 & 0 & 3 \\ 2 & 0 & 0 & 3 \\ 0 & 0 & 0 & 3 \\ 0 & 0 & 2 & 0 \end{bmatrix}$$

x_1	y_1	x_2	y_2	Entry in M
0	0	0	1	(0,1)=4
0	1	0	1	(0,3)=3
0	0	1	0	(1,0)=2
0	1	1	1	(1,3)=3
1	1	0	1	(2,3)=3
1	1	1	0	(3,2)=2

Figure 4.10: A matrix and its MTBDD encoding

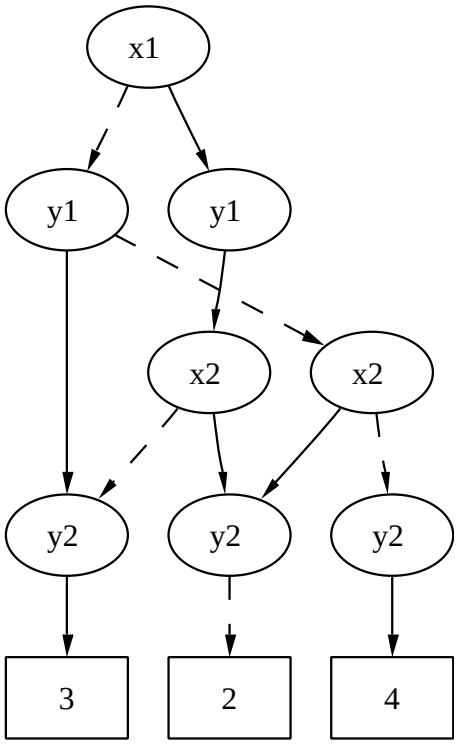

Figure 4.11: Reduced MTBDD representing the matrix in Figure 4.10

- Function Threshold($\mathbf{M}, >, 0$) returns the BDD by replacing each terminal node with 1 if and only if its value is greater than 0.
- Function ThereExists($\underline{\mathsf{x}}, \mathbf{M}$) returns the MTBDD by performing existential quantification on the the Boolean vector $\underline{\mathsf{x}}$.
- Function ReplaceVars($\mathbf{M}, \underline{\mathsf{y}}, \underline{\mathsf{x}}$) returns the MTBDD by replacing the Boolean vector $\underline{\mathsf{y}}$ with $\underline{\mathsf{x}}$.

Chapter 5

Model Checking for Probabilistic Timed Automata

In this chapter, we present an overview of the model checking algorithms that have been developed for probabilistic timed automata. These algorithms form the basis of the implementation study in this thesis. We introduce the model of probabilistic timed automata in Section 5.2. The logic Probabilistic Timed Computation Tree Logic (PTCTL) is described in Section 5.3. Section 5.4 presents algorithms for model checking for probabilistic timed automata.

5.1 Labelled Timed Probabilistic Systems

Below we recall *labelled timed probabilistic systems* which is an extension of Markov decision processes.

Definition 5.1.1 *A* Labelled timed probabilistic system *is a tuple* $LTPS = (S, \bar{s}, L, Act, Steps)$ *where:*

- S *is the (possible infinite) set of* states*;*
- $\bar{s} \in S$ *is the* initial state*;*
- $L : S \rightarrow 2^{AP}$ *is a* labelling function*;*

- *Act is the set of* actions*;*
- *Steps* $\subseteq S \times (Act \cup \mathbb{R}^+) \times \mathsf{Dist}(S)$ *is the* probabilistic transition relation*, such that, if* $(s, t, \mu) \in$ *Steps for any* $t \in \mathbb{R}^+$*, then* μ *is a point distribution.*

As for Markov decision processes, we can introduce paths and adversaries for labelled timed probabilistic systems. As in real-time systems, we restrict attention to *time-divergent adversaries* so that unrealisable behaviour (i.e. corresponding to time not advancing beyond a bound) is disregarded during analysis. For any path

$$\omega = s_0 \xrightarrow{t_0,\mu_0} s_1 \xrightarrow{t_1,\mu_1} s_2 \xrightarrow{t_2,\mu_2} \cdots$$

of a timed probabilistic system, the duration up to the n+1th state of ω, denoted $\mathcal{D}_\omega(n{+}1)$, equals $\sum_{i=0}^{n} t_i$. We say that a path ω is *divergent* if for any $t \in \mathbb{R}^+$, there exists $j \in \mathbb{N}$ such that $\mathcal{D}_\omega(j){>}t$.

Definition 5.1.2 *An adversary A of a labelled timed probabilistic system LTPS is* divergent *if and only if for each state s of LTPS the probability under* $Prob_s^A$ *of the divergent paths of* $Path_{ful}^A(s)$ *is* 1*. Let* Adv_{LTPS} *be the set of divergent adversaries of LTPS.*

A restriction imposed on timed probabilistic systems is that of *non-zenoness*, which stipulates that there does not exist a state from which time cannot diverge, as this situation is considered to be a modelling error.

Definition 5.1.3 *A labelled timed probabilistic systems LTPS is* non-zeno *if and only if there exists a divergent adversary of LTPS.*

5.2 Probabilistic Timed Automata

Probabilistic timed automata are an extension of both timed automata and Markov decision processes.

5.2.1 Syntax of Probabilistic Timed Automata

Definition 5.2.1 *(Probabilistic timed automaton) A probabilistic timed automaton [KNSS02] is a tuple* $PTA = (L, \mathcal{L}, \bar{l}, Act, \mathcal{X}, inv, prob)$ *which contains:*

- *a finite set L of nodes,*
- *a function $\mathcal{L} : L \to 2^{AP}$ assigning to each node of the automaton the set of atomic propositions that are true in that node,*
- *a start node $\overline{l} \in L$,*
- *a finite set Act of action labels,*
- *a finite set $\mathcal{X}$ of clocks,*
- *a function $inv : L \to Zones(\mathcal{X})$ assigning to each node an invariant condition,*
- *a probabilistic transition relation $prob \subseteq L \times Zones(\mathcal{X}) \times Act \times Dist(2^{\mathcal{X}} \times L)$ which consists of a source node, an enabling condition, an action label, and a set of probability distributions p which assigns probability to pairs of the form (X, l') for a set of clocks X to be reset and target node l'.*

Definition 5.2.2 *An* edge *of PTA generated by $(l, g, p) \in prob$ is a tuple of the form (l, g, p, X, l') such that $p(X, l') > 0$. Let $edges(l, g, p)$ be the set of edges generated by (l, g, p), and let $edges = \bigcup_{(l,g,p) \in prob} edges(l, g, p)$.*

The definition of probabilistic timed automata is similar to that of timed automata. There are two types of clock constraints: invariants labelling nodes, and guards labelling transitions. Each node of a timed automaton is labelled with an invariant, a Boolean condition on the clocks stating how long the system can stay in that node. Transitions are labelled with actions, Boolean conditions called guards stating whether the action can be taken, and the set of clocks whose value is to be reset. The only difference between probabilistic timed automata and timed automata is that edges in probabilistic timed automata are generalised to discrete probability distributions. And one requirement on the edges is that the guard must be same for all of the edges if they belong to a discrete probability distribution.

Figure 5.1 shows a simple probabilistic timed automaton where the actions are omitted. The example can be used to control the door opening or closing. If the door is in the closed state and a control button is pressed, the door will be opened with probability 0.99

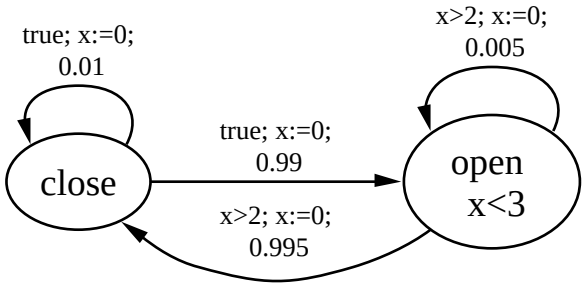

Figure 5.1: Probabilistic timed automaton example

or stay closed with probability 0.01 when it is faulty. Once the door is opened, it will stay opened for 2 to 3 time units, then it will close with probability 0.995 or remain open after which it will repeatedly try to close.

5.2.2 Semantics of Probabilistic Timed Automata

The semantics of a probabilistic timed automaton is defined in terms of labelled timed probabilistic system.

Definition 5.2.3 *Let* $PTA = (L, \mathcal{L}, \overline{l}, Act, \mathcal{X}, inv, prob)$ *be a probabilistic timed automaton. The* semantics of PTA *is defined as the labelled timed probabilistic system* $LTPS_{PTA} = (S, \bar{s}, \mathcal{L}', Act, Steps)$ *where:*

- $S \subseteq L \times \mathbb{R}^+$ *and* $(l, v) \in S$ *if and only if* $v \triangleleft inv(l)$*;*
- $((l, v), t, \mu) \in Steps$ *if and only if one of the following conditions holds:*
  time transitions: $t \geq 0$, $\mu{=}\mu_{(l,v+t)}$ *and* $v{+}t' \triangleleft inv(l)$ *for all* $0 \leq t' \leq t$
  discrete transitions: $t = 0$ *and there exists* $(l, g, p) \in prob$ *such that* $v \triangleleft g$*, for any* $(X, l') \in support(p)$ *we have* $(l', v[X := 0]) \in S$*, and for any* $(l', v') \in S$*:*

$$\mu(l', v') = \sum_{\substack{X \subseteq \mathcal{X} \,\&\\ v' = v[X:=0]}} p(X, l');$$

- $\mathcal{L}'(l, v) = \mathcal{L}(l)$ *for any* $(l, v) \in S$*.*

We say that PTA is non-zeno if and only if $LTPS_{PTA}$ is non-zeno.

The definition of $LTPS$ is close correspondence to Markov decision processes. The system starts its execution with all clocks set to 0. The values of all the clocks increase at the same rate as time. At any point in time, if the clocks satisfy the invariant the automaton can stay in a node and let time pass. When a guard is satisfied, a discrete transition can be taken. However, the choice of the next node of the automaton is now both probabilistic and nondeterministic. Discrete transitions are instantaneous and consist of the following two steps which resolve nondeterminism and probability in succession: as soon as the system makes a nondeterministic choice among the set of distributions, then supposing that the probability distribution $p \in Dist(2^{\mathcal{X}} \times L)$ is chosen, the system then makes a probabilistic transition according to p. When the automaton executes a discrete transition, a set of clocks can additionally be reset to 0. We use (l, g, p) for a probability distribution which could be chosen at node l associated with the guard condition g according to p. The set of probabilistic branches of (l, g, p) is denoted $edges(l, g, p)$.

5.2.3 Parallel Composition

The parallel composition of several interacting sub-components is often useful to define complex systems.

Let $PTA_1 = (L_1, \mathcal{L}_1, \overline{l}_1, Act_1, \mathcal{X}_1, inv_1, prob_1)$ and $PTA_2 = (L_2, \mathcal{L}_2, \overline{l}_2, Act_2, \mathcal{X}_2, inv_2, prob_2)$

Definition 5.2.4 *[KNS03a] The parallel composition of two probabilistic timed automata PTA_1 and PTA_2, where $\mathcal{X}_1 \cap \mathcal{X}_2 = \emptyset$, is the probabilistic timed automaton*

$$PTA_1 \| PTA_2 = (L_1 \times L_2, \mathcal{L}_1 \cup \mathcal{L}_2, (\overline{l_1}, \overline{l_2}), Act_1 \cup Act_2, \mathcal{X}_1 \cup \mathcal{X}_2, I, prob)$$

where $inv(l, l') = inv_1(l) \wedge inv_2(l')$ for all $(l, l') \in L_1 \times L_2$ and $((l_1, l_2), g, \sigma, p) \in prob$ if and only if one of the following conditions holds:

1. *$\sigma \in Act_1 \setminus Act_2$ and there exists $(l_1, g_1, \sigma_1, p_1) \in prob_1$ such that $p = p_1$*

2. *$\sigma \in Act_2 \setminus Act_1$ and there exists $(l_2, g_2, \sigma_2, p_2) \in prob_2$ such that $p = p_2$*

3. $\sigma \in Act_1 \cap Act_2$ *and there exists* $(l_1, g_1, \sigma_1, p_1) \in prob_1$ *and* $(l_2, g_2, \sigma_2, p_2) \in prob_2$ *such that* $g = g_1 \wedge g_2$ *and* $p = p_1 \otimes p_2$

where for any $l_1 \in L_1, X_1 \subseteq \mathcal{X}_1$, $l_2 \in L_2, X_2 \subseteq \mathcal{X}_2$, *we let* $p_1 \otimes p_2(X_1 \cup X_2, (l_1, l_2)) = p_1(X_1, l_1) \cdot p_2(X_2, l_2)$.

Condition 3 specifies that the requirement for the labels is that they must be the same name so that two or more automata can synchronise if and only if they have the same synchronisation labels and both their guard conditions are fulfilled.

5.3 Probabilistic Timed Computation Tree Logic (PTCTL)

We now describe the probabilistic real-time logic - Probabilistic Timed Computation Tree Logic (PTCTL) which can be used to specify properties of probabilistic timed automata.

PTCTL is a combination of two extensions of the branching-time temporal logic CTL, the real-time temporal logic TCTL [HNSY92] discussed in Section 4.1.3 and the probabilistic temporal logic PCTL discussed in Section 4.2.2. PTCTL is obtained by enhancing TCTL with the probabilistic operator $\mathcal{P}_{\bowtie p}[\cdot]$ from PCTL.

As in TCTL, PTCTL employs a set of *formula clocks*, $\mathcal{Z}$, disjoint from the clocks $\mathcal{X}$ of the PTA. Formula clocks are assigned values by a *formula clock valuation* $\mathcal{E} : \mathcal{Z} \to \mathbb{R}$, which uses the notation for clock valuations in the standard way.

Definition 5.3.1 *The syntax of* PTCTL *is defined as follows:*

$$\phi ::= a \mid \zeta \mid \neg\phi \mid \phi \wedge \phi \mid z.\phi \mid \mathcal{P}_{\bowtie p}[\phi \; \mathcal{U} \; \phi] \mid \mathcal{P}_{\bowtie p}[\phi \; \mathcal{V} \; \phi]$$

where $a \in AP$, $\zeta \in Zones(\mathcal{X} \cup \mathcal{Z})$, $z \in \mathcal{Z}$, $\bowtie \in \{\leq, <, >, \geq\}$ *and* $p \in [0, 1]$.

Note that the values of system clocks in $\mathcal{X}$ and formula clocks in $\mathcal{Z}$ can be obtained from a state and a formula clock valuation, respectively. Take a particular $\zeta \in Zones(\mathcal{X} \cup \mathcal{Z})$, and consider the syntactic interpretation of ζ as a conjunction of constraints. Then, given a state (l, ν) and a formula clock valuation $\mathcal{E}$, we denote by $\zeta[(l, \nu), \mathcal{E}]$ the Boolean value

obtained by replacing each occurrence of a system clock $x \in \mathcal{X}$ in ζ by $\nu(x)$, and of a formula clock $z \in \mathcal{Z}$ in ζ by $\mathcal{E}(z)$.

Similarly, we use the abbreviation $\diamond\phi$ for $true\ \mathcal{U}\ \phi$ inside PTCTL formula.

In PTCTL, we can express properties such as:
"with probability 0.95 or greater, the leader could be elected within 8 time units", which is represented as the PTCTL formula $z.\mathcal{P}_{\geq 0.95}[\Diamond(z < 8)]$.

Definition 5.3.2 *Let $\mathcal{M} = (S, \bar{s}, L, Act, \mathcal{D}, Steps)$ be a labelled timed probabilistic system. For any state s of $\mathcal{M}$, formula clock valuation $\mathcal{E}$, and* PTCTL *formula ϕ, the satisfaction relation $s, \mathcal{E} \models \phi$ is defined inductively as follows:*

$$
\begin{array}{lll}
s, \mathcal{E} \models true & & \text{for all } s \text{ and } \mathcal{E} \\
s, \mathcal{E} \models a & \Leftrightarrow & a \in L(s) \\
s, \mathcal{E} \models \zeta & \Leftrightarrow & \zeta[s, \mathcal{E}] = true \\
s, \mathcal{E} \models \phi_1 \wedge \phi_2 & \Leftrightarrow & s, \mathcal{E} \models \phi_1 \text{ and } s, \mathcal{E} \models \phi_2 \\
s, \mathcal{E} \models \neg\phi & \Leftrightarrow & s, \mathcal{E} \not\models \phi \\
s, \mathcal{E} \models z.\phi & \Leftrightarrow & s, \mathcal{E}[z := 0] \models \phi \\
s, \mathcal{E} \models \mathcal{P}_{\bowtie p}[\phi_1\ \mathcal{U}\ \phi_2] & \Leftrightarrow & p^A_{s,\mathcal{E}}(\phi_1\ \mathcal{U}\ \phi_2) \bowtie p \text{ for all } A \in Adv_{\mathcal{M}} \\
s, \mathcal{E} \models \mathcal{P}_{\bowtie p}[\phi_1\ \mathcal{V}\ \phi_2] & \Leftrightarrow & p^A_{s,\mathcal{E}}(\phi_1\ \mathcal{V}\ \phi_2) \bowtie p \text{ for all } A \in Adv_{\mathcal{M}}
\end{array}
$$

where for any adversary $A \in Adv_{\mathcal{M}}$ and for any path $\omega \in Path$:

$$
p^A_{s,\mathcal{E}}(\phi_1\ \mathcal{U}\ \phi_2) \;=\; Prob^A_s(\{\omega \in Path^A_{ful}(s) \mid \omega, \mathcal{E} \models \phi_1\ \mathcal{U}\ \phi_2\})
$$

$$\begin{aligned}\omega, \mathcal{E} \models \phi_1 \,\mathcal{U}\, \phi_2 \quad \Leftrightarrow \quad & \text{there exist } i \in \mathbb{N}, t \in \mathbb{R} \text{ such that} \\ & \omega(i,t), \mathcal{E} + \mathcal{D}_\omega(i) + t \models \phi_2, \text{ and} \\ & \text{for all } j < i, \text{ we have:} \\ & \omega(j,t), \mathcal{E} + \mathcal{D}_\omega(j) + t \models \phi_1 \vee \phi_2 \\ & \text{and for } j = i \text{ and } t' < t \text{ we have:} \\ & \omega(j,t'), \mathcal{E} + \mathcal{D}_\omega(j) + t' \models \phi_1 \vee \phi_2 \,.\end{aligned}$$

$$p^A_{s,\mathcal{E}}(\phi_1 \;\mathcal{V}\; \phi_2) \;=\; Prob^A_s(\{\omega \in Path^A_{ful}(s) \mid \omega, \mathcal{E} \models \phi_1 \;\mathcal{V}\; \phi_2\})$$

$$\begin{aligned}\omega, \mathcal{E} \models \phi_1 \;\mathcal{V}\; \phi_2 \quad \Leftrightarrow \quad & \text{for all } i \in \mathbb{N}, t \in \mathbb{R} \text{ such that if} \\ & \text{for all } j \leq i, t' \in \mathbb{R} \text{ such that } j = i \\ & \text{implies } t' < t \text{ we have:} \\ & \omega(j,t'), \mathcal{E} + \mathcal{D}_\omega(j) + t' \not\models \phi_1 \wedge \phi_2 \\ & \text{then } \omega(i,t), \mathcal{E} + \mathcal{D}_\omega(i) + t \models \phi_2.\end{aligned}$$

5.4 Algorithms for Model Checking Probabilistic Timed Automata

In [KNSS99], a method is proposed for dealing with probabilistic timed automata which have both probabilistic and timed behaviours. However, the complexity of this algorithm is too high when verifying such systems because it is based on constructing the region graph [AD94]. One advantage of the method is that it can establish the correctness of a probabilistic timed automaton model against any properties expressible in PTCTL. Optimised algorithms such as the forward reachability analysis [KNSS02] and backward

reachability analysis [KNS00, KNS03b] have been proposed. These algorithms have the potential to reduce the high complexity of model checking for probabilistic timed automata as demonstrated by an implementation of forward reachability in [DKN02, DKN04]. The forward reachability algorithm can reduce the complexity when constructing quotient of the region graph by performing forward search to construct the zone graph which is often smaller than the region graph, although the worst case complexity is same. One drawback of the forward reachability is that it can only obtain an upper bound of the maximum probability of reaching a set of target states [KNSS02]. However, by using the backward reachability algorithm [KNS00, KNS03b], both the exact maximum and minimum probability of reaching the target set can be obtained. Furthermore, the backward approach can be applied to verify full PTCTL.

One successful method applied to model checking for non-probabilistic timed systems is the on-the-fly technique. However, on-the-fly cannot be applied to model checking probabilistic timed systems. The reason is that we have to traverse the whole state space in order to calculate the probability information.

As in the non-probabilistic real-time reachability case, our finite state model derived from the probabilistic timed automaton is obtained, not by the region construction, but by forward or backward search through the infinite-state space of probabilistic timed automaton. Once this has been done, probabilistic reachability analysis is then performed on the finite-state model through computation of probabilities using classical model checking algorithms for Markov decision processes based on linear programming.

The following sections describe how to obtain a finite state model as a Markov decision process by performing the forward or backward reachability analysis.

5.4.1 The Forward Algorithm

Figure 5.2 shows a modified algorithm of forward reachability from [KNSS02]. The purpose of the modification is to make the algorithm more readable as the original one does not make the construction of the probabilistic edges explicit.

The algorithm accepts two arguments: the initial symbolic state and the target set of nodes. A symbolic state takes the form of (l, ζ) where l is a node in the given probabilistic

$ForwardReachability((\bar{l}, \zeta_0), R)$	
1.	$c := MaxConstant$
2.	$ResultSet := \emptyset$
3.	$TargetSet := \emptyset$
4.	$Edges := \emptyset$
5.	$FrontSet := \{Normalise(TimeSuccessors((\bar{l}, \zeta_0)), c)\}$
6.	**repeat**
7.	**choose**$(l, \zeta) \in FrontSet$
8.	**if** $l \in R$ **then** $TargetSet := TargetSet \cup \{(l, \zeta)\}$
9.	**else**
10.	**for each** $e \in edges(l, g, p)$ do
11.	**let** $(l', \zeta') := post(e, (l, \zeta), c)$
12.	**if** $\zeta' \neq \emptyset$ and $(l', \zeta') \notin Z$ then
13.	$FrontSet := FrontSet \cup \{(l', \zeta')\}$
14.	$Edges := Edges \cup \{((l, \zeta), e, (l', \zeta'))\}$
15.	**end if**
16.	**end for each**
17.	**end if**
18.	$ResultSet := ResultSet \cup \{(l, \zeta)\}$
19.	**until** $FrontSet = \emptyset$
20.	**return** $ResultSet$, $TargetSet$, $Edges$

Figure 5.2: The Forward algorithm

timed automaton and ζ is a zone. Line 1 assigns the maximum constant appearing in the model to a variable c. Lines 2-5 of the algorithm initialise the set $ResultSet$, $Edges$ and the set $TargetSet$ to the empty set and the set $FrontSet$ to include the initial symbolic state whose zone part is obtained by setting all clock values to zero. The set $ResultSet$ is used to store those symbolic states which have been explored, the set $TargetSet$ to store symbolic states whose node part belongs to target set R, the set $Edges$ to store the probabilistic transitions between symbolic states, and the set $FrontSet$ to store those symbolic states whose successors may need to be explored. Lines 6-19 of the algorithm repeat taking a symbolic state (l, ζ) from $FrontSet$ until $FrontSet$ is empty. While $FrontSet$ is non-empty, it means that there are reachable symbolic states for which the successor states may not have been explored. In such a case, the statements within the body of the repeat until loop are executed. This involves taking a symbolic state (l, ζ) from $FrontSet$ and adding it to $ResultSet$. If its node part is a member of the target set R, then it is added to the set $TargetSet$. Otherwise,it takes each successor independently of the symbolic state (l, ζ) and looks to see if it is in $ResultSet$: if the successor has already been in $ResultSet$, do nothing; otherwise, place it in $FrontSet$ and construct probabilistic edge using current symbolic state and this successor and add to Edges (Lines 13-14).

Given a probabilistic timed automaton $PTA = (L, \mathcal{L}, \overline{l}, Act, \mathcal{X}, I, prob)$, initial symbolic state and the target set $R \in L$, the portion of the state space that is generated by the ForwardReachability algorithm takes the form of a Markov decision process. The symbolic states in set $ResultSet$ form the states of the Markov decision process and the edges between a symbolic state and its successor symbolic state are used to define the required probabilistic transitions. Once this has been done, conventional probabilistic reachability analysis is then performed on the finite-state MDP model through computation of probabilities using linear programming.

From the algorithm in Figure 5.2, we observe that a symbolic state (l, ζ) is to be added to two different sets, the set $FrontSet$ and the set $ResultSet$. This can be made more efficient by adding each symbolic state once in an explicit implementation: the set of $FrontSet$ and $ResultSet$ do not need to be stored separately and could be realised efficiently by using one common data structure, such as a list.

The algorithm in Figure 5.2 is quite similar to the forward algorithm for the non-probabilistic timed systems discussed earlier Section 4.3. The states in the probabilistic systems generated by this algorithm have the form of a pair (l, ζ), where l is the node and ζ is the zone. However, unlike reachability analysis in non-probabilistic timed systems, there are three differences:

- The on-the-fly technique can not be applied, and the whole state space has to be traversed in order to construct an MDP model over zones.
- Edges between generated symbolic states need to be constructed.
- In the non-probabilistic case, if we have two symbolic states, for example, (l, ζ_1) and (l, ζ_2), and one of them is a subset of the other, that is, they contain the same discrete parts and one continuous part is a subset of the other, one only needs to store the union of both, that is $(l, \zeta_1 \cup \zeta_2)$ instead of two separate states. However, in the case of probabilistic timed automata, we have to distinguish two sets of generated states even if one of them is a subset of the other. In other words, we have to store many small sets of states separately instead of simply their union.

Example 5.1 We use the simple door control example of Figure 5.1 and assume the initial node is *close*. We consider the property $\mathcal{P}_{<0.9}[\Diamond open]$ which states that whether the maximal probability to reach node *open* from the initial state is 0.9. Figure 5.3 shows the zone graph obtained via the forward exploration algorithm in Figure 5.2. By using an integer variable s_1 ranging over [0,1], where $s_1 = 0$ encodes the symbolic state $(close, x \geq 0)$ and $s_1 = 1$ encodes the symbolic state $(open, 0 \leq x < 3)$, the zone graph in Figure 5.3, which is an MDPs over zones, can be translated into the PRISM language shown in Figure 5.4. Using PRISM software, we obtain the probability of this property is 1.

5.4.2 The Backward Algorithms

As pointed out in [KNSS99], the maximum probability of reaching the target set, when computed by the forward reachability algorithm in Figure 5.2, is an upper bound on the

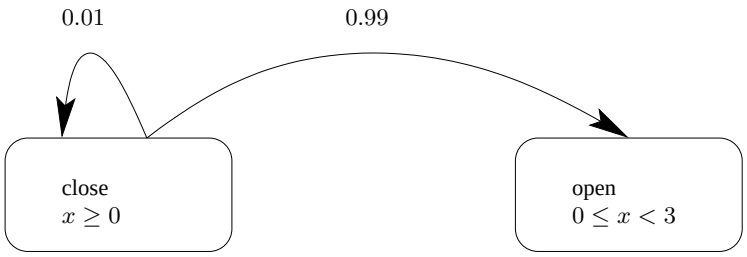

Figure 5.3: Zone graph obtained via the forward exploration of the PTA in Figure 5.1

nondeterministic

module M_1
$s_1 : [0..1]$ **init** 0;
[] $(s_1 = 0) \rightarrow 0.01 : (s_1' = 0) + 0.99 : (s_1' = 1)$;
endmodule

Figure 5.4: The PRISM description of the Figure 5.3

actual maximum probability of the probabilistic timed automaton reaching the target set. However, by using the backward algorithm of [KNS00] the exact maximum probability of reaching the target set can be obtained. Furthermore, the exact minimum probability of reaching the target set can be obtained using backward algorithm of [KNS03b]. However, compared with forward algorithm, the backward algorithms can produce exact probabilities and be applied to full Probabilistic Timed Computation Tree Logic (PTCTL) [KNSS02] but at a cost of higher computational complexity. The computation complexity of minimum probability is even higher than that of backward maximum probability due to the constraint on universal time divergent adversaries over paths [KNS03b]. For computing maximum probability, the constraint on time divergent adversaries oner paths can be safely remove because we only consider finite paths which can alway be safely enpanded to time divergent infinite paths. This cannot be applied to the case of minimum probability because the probability can be different between under divergent adversaries and under non-divergent adversaries.

Next we describe these algorithms which are the subject of our implementation.

The Maximum Probability Backward Algorithm Figure 5.5 shows the algorithm MaxUntil [KNS03b] used to compute the exact maximum probability of reaching the target set V via set U. Actually, the algorithm MaxUntil proceeds through the computation of least fixpoint [CGP99] starting with the set of symbolic states satisfying formula V and iteratively computing larger sets of symbolic states through backward graph search by first computing predecessor states and then keep those in set U by set intersection operations. Because during backward search not all edges will be detected [KNS00], the edge set needs to be extended and this is done by explicitly considering all the relevant intersections of symbolic states and induced edges. Two symbolic states are *relevant* if and only if the intersection of their zone part is not empty and they have the same node part. The nodes in result set Z are used to define the states of a Markov decision process and the edges in result set $E_{(l,g,p)}$ are used to define the generated probabilistic transitions which are induced by probability distribution (l, g, p). Each edge in $E_{(l,g,p)}$ takes the form of $(z, (X, l'), y)$, where z and y are symbolic states and X is the set of clocks to be reset when p is chosen.

MaxUntil(U, V)

1. $Z := tpre_{U \cup V}(V)$
2. **for** $(l, g, p) \in prob$
3. $\quad E_{(l,g,p)} := \emptyset$
4. **end for**
5. **repeat**
6. $Y := Z$
7. **for** $y \in Y \wedge (l, g, p) \in prob \wedge e = (l, g, p(X, l')) \in edges(l, g, p)$
8. $\quad z := U \cap dpre(e, tpre_{U \cup V}(y))$
9. $\quad$ **if** $(z \neq \emptyset) \wedge (z \notin tpre_{U \cup V}(V))$
10. $\quad\quad Z := Z \cup z$
11. $\quad\quad E_{(l,g,p)} := E_{(l,g,p)} \cup \{(z, (X, l'), y)\}$
12. $\quad\quad$ **for** $(\bar{z}, (\bar{X}, \bar{l'}), \bar{y}) \in E_{(l,g,p)}$
13. $\quad\quad\quad$ **if** $(z \cap \bar{z} \neq \emptyset) \wedge (X, l') \neq (\bar{X}, \bar{l'}) \wedge (z \cap \bar{z} \notin tpre_{U \cup V}(V)))$
14. $\quad\quad\quad\quad Z := Z \cup (z \cap \bar{z})$
15. $\quad\quad\quad$ **end if**
16. $\quad\quad$ **end for**
17. $\quad$ **end if**
18. **end for**
19. **until** $Z = Y$
20. **construct** $PS = (Z, Steps)$ **where** $(z, \rho) \in Steps$ **if and only if there exists** $(l, g, p) \in prob$ **and** $E \subseteq E_{(l,g,p)}$ **such that**
    - $z \in \{z' \mid (z', e, z'') \in E\}$
    - $(z', e, z'') \in E \;\Rightarrow\; z' \supseteq z$
    - $(z'_1, e, z') \neq (z'_2, e', z'') \in E \;\Rightarrow\; e \neq e'$
    - E **is maximal**
    - $\rho(z') = \sum\{\!|\, p(X, l') \mid (z, (X, l'), z') \in E \,|\!\}\;\; \forall z' \in Z$
21. **return** $\bigcup\{\mathsf{tpre}_{U \vee V}(z) \mid z \in Z \wedge p_z^{max}(\Diamond\, \mathsf{tpre}_{U \vee V}(V)) \gtrsim \lambda\}$

Figure 5.5: The MaxUntil algorithm

```
pre1(U, V)
1.  Y := ∅
2.  for (l, g, p) ∈ prob
3.      Y_0 := true
4.      Y_1 := ∅
5.          for e ∈ edges(l, g, p)
6.              Y_0 := dpre(e, U) ∩ Y_0
7.              Y_1 := dpre(e, V) ∪ Y_1
8.          end for
9.      Y := (Y_0 ∩ Y_1) ∪ Y
10. end for
11. return Y
```

Figure 5.6: The $pre1$ algorithm

```
MaxU_{≥1}(U, V)
1.  Z_0 := true
2.  repeat
3.      Y_0 := Z_0
4.      Z_1 := ∅
5.      repeat
6.          Y_1 := Z_1
7.          Z_1 := V ∪ (U ∩ pre1(Y_0, Y_1))
8.          Z_1 := Z_1 ∪ tpre_{U∪V}(Y_0 ∩ Y_1)
9.      until Z_1 = Y_1
10.     Z_0 := Z_1
11. until Z_0 = Y_0
12. return Z_0
```

Figure 5.7: The $MaxU_{\geq 1}$ algorithm

$MaxV_{\geq 1}(c, U, V)$
1. $Z := \mathbf{true}$
2. **repeat**
3. $\quad X := Z$
4. $\quad Z := V \cap z.MaxU_{\geq 1}(X, (U \cap X) \cup \{z > c\})$
5. **until** $Z = X$
6. **return** Z

Figure 5.8: The $MaxV_{\geq 1}$ algorithm

The algorithm accepts two parameters: the symbolic sets U and V. Lines 1-4 initialise both the result set and edge set with the empty set. Lines 5-19 perform the fixed point calculation as described above. Line 8 generates the symbolic states by the *tpre* first and then *dpre* operation and keeps those that are included in the set U. Lines 12-16 generate relevant states. Line 20 constructs the probabilistic system PS using the probabilistic edges of the probabilistic timed automaton and the computed edge sets. The states of PS are the symbolic states generated by the previous steps (Lines 1-19) of the algorithm, and the probabilistic transition relation of PS is constructed by grouping the graph edges generated by the same probabilistic edge of the probabilistic timed automaton under study. Finally, in line 21, the maximum probability of reaching $\mathsf{tpre}_{U \vee V}(V)$ is computed for each $z \in Z$. Note that $z \neq \emptyset$ means if and only if z encodes at least one state and formula clock valuation pair.

Example 5.2 We still consider the property $\mathcal{P}_{>0.9}[\Diamond open]$, but this time we are using the backward algorithm. Figure 5.9 shows the zone graph obtained via the maximum backward exploration of the simple door control example of Figure 5.1, where the initial node is *close*. We obtain the probability of this property is 1 after the graph is translated into PRISM language and checked using PRISM software.

The Minimum Probability Backward Algorithm When computing minimum probability, due to the constraint on universal time divergent adversaries over paths, techniques

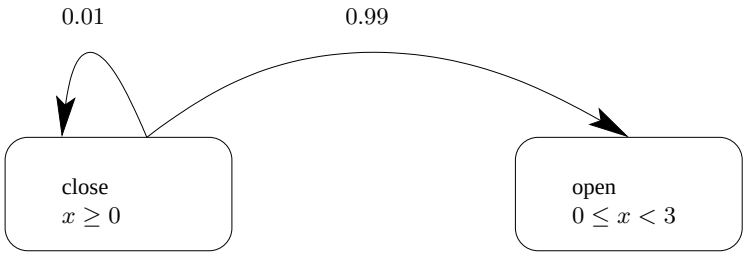

Figure 5.9: Zone graph obtained via the maximum probability backward exploration of the PTA in Figure 5.1

which are based on Propostions 5.4.1- 5.4.3 [KNS03b] are introduced in order to avoid the universal time divergent adversaries over paths.

Proposition 5.4.3 is based on [HNSY92] where it is shown that verifying $\phi\ \forall\mathcal{U}\ \psi$ ('all divergent paths satisfy $\phi\ \mathcal{U}\ \psi$') reduces to computing the fixpoint:

$$lgp\ Y.\big(\ \psi \vee \neg z.\big(\ \neg Y\ \exists\mathcal{U}\ (\neg(\phi \vee Y) \vee (z{>}c))\ \big)\ \big) \tag{5.1}$$

for any $c \in \mathbb{N}$ greater than 0. The important point is that the universal quantification over paths has been replaced by an existential quantification, combined with a constraint enforcing that more than c time units must elapse repeatedly.

Proposition 5.4.1 $p^{min}_{s,\mathcal{E}}(U\ \mathcal{U}\ V) = 1 - p^{max}_{s,\mathcal{E}}(\neg U\ \mathcal{V}\ \neg V)$

Proposition 5.4.2 $p^{max}_{s,\mathcal{E}}(\neg U\ \mathcal{V}\ \neg V) = p^{max}_{s,\mathcal{E}}(\neg V\ \mathcal{U}\ \mathcal{P}_{\geq 1}[\neg U\ \mathcal{V}\ \neg V])$

Proposition 5.4.3 $\mathcal{P}_{\geq 1}(\neg U\ \mathcal{V}\ \neg V) = \nu X_{\cdot}(\neg V \wedge z_{\cdot}\neg\mathcal{P}_{<1}[X\ \mathcal{U}\ ((X \wedge \neg U) \vee \{z > c\}])$

Propositions 5.4.1-5.4.3 in [KNS03b] give the way to compute the set of symbolic states satisfying $z.\mathcal{P}_{>p}[U\ \mathcal{U}\ V]$, which is given by the following set of symbolic states: $\{true\} \setminus MaxUntil(\neg V, MaxV_{\geq 1}(c, \neg U, \neg V))$.

Combined with the algorithm in Figure 5.5, algorithms in Figure 5.6, 5.7 and 5.8 are used to compute the exact minimum probability of reaching the target set.

The intuition of the algorithm $pre1(U, V)$ in Figure 5.6 is to calculate the set of states which could reach states in both sets V and U of states via a single discrete transition

but is restricted to those that can reach from set U only with probability one, which is ensured in Line 6 by set conjunction on all branching of same distribution.

The algorithm $MaxU_{\geq 1}(U, V)$ in Figure 5.7 computes the set that can reach set V via set U with maximum probability one. It contains a double fixpoint which is implemented as two nested loops. This algorithm is based on [dAKN$^+$00], but with the consideration of timed transitions in Line 8. The intuition of the algorithm in Figure 5.7 is to calculate the set of states which could reach states in states of set V via set U, but only via U with probability one. The outer loop computes the set of states Z_0 which is initially set to the whole state space, and will contain the final set of states as required when this algorithm terminates. In each iteration of the outer loop, states which cannot reach U with probability one are removed; this is achieved by the inner loop which calls the algorithm in Figure 5.6.

The algorithm $MaxV_{\geq 1}(U, V)$ in Figure 5.8 is a greatest fixpoint [CGP99] calculation which will return the set of states that satisfy $U\ \mathcal{V}\ V$ with maximum probability one. The parameter c can be any integer value. Because $MaxV_{\geq 1}$ contains $MaxU_{\geq 1}$, it means that the computation of $MaxV_{\geq 1}$ has 3 nested fixpoints which is implemented by 3 nested loops.

Example 5.3 We consider the property $\mathcal{P}_{>p}[\Diamond open]$, which is to calculate the minimum probability of reaching open given the model in 5.1 and *close* is the initial node. Here the set U is $\{true\}$ and the set V is $\{open\}$. In order to compute $\{(true)\} \setminus MaxUntil(\neg V, MaxV_{\geq 1}(c, \neg U, \neg V))$, we first compute $MaxV_{\geq 1}(c, \{false\}, \{(close)\})$ because $\neg\{true\} = \{false\}$ and $\neg\{(open)\} = \{(close)\}$. The details on the computations of the set of symbolic states S satisfying
$MaxV_{\geq 1}(c, \{false\}, \{(close)\})$ is given in Appendix A.4. The returning set of symbolic states S is $\{(close)\}$. Next we compute the set of symbolic states satisfying
$MaxUntil(\{(close)\}, \{(close)\})$, which only returns $\{(close)\}$. As the $\{(true)\}\setminus\{(close)\} = \{(open)\}$, it means that the minimum probability of this property is 0 because there is no initial node found in the resulting set of symbolic states.

Chapter 6

Explicit Implementation of the Forward Exploration Algorithm

The forward reachability was first implemented using KRONOS [KRO] and PRISM in [DKN02, DKN04]. In this chapter we consider an explicit implementation of the forward algorithm. This means we do not exploit the mechanism of or benefit from symbolic techniques. We use explicit data structures such as an adjacency list or array to store the set of states, and there is no sharing among them. Although we are aware that this is subject to the well-known problem of state-space explosion, the purpose of this implementation is, firstly, to provide a prototype implementation for use on case studies, and, secondly, to identify the possible bottlenecks of the problem of model checking probabilistic timed automata via the forward algorithm. This chapter begins with the implementation of the forward algorithm in an explicit manner described in Section 6.1. In Section 6.2, we present experimental results obtained from the explicit version.

6.1 Implementation

The implementation that we have developed first reads in a model of a probabilistic timed automaton from a file which is in the textual notation described in Section 6.1.1. Then the information about nodes and the transition relation between the nodes, as well as the discrete probability, are parsed and the product model is constructed and stored in a

temporary data structure. Next, the graph searcher implementing the forward reachability algorithm analyses the product automaton starting from the initial node until all of the reachable states are found. Meanwhile, the reachable states are used to generate the model which takes the form of an MDP. Finally, the specification which is dependent on what state we want to reach is expressed in PCTL and the resulting model are translated into the PRISM input language, and then are handed over to the PRISM model checker to perform probability calculations.

6.1.1 Explicit Construction of the Model

The first step to model check a PTA is to construct an internal representation of the model. This involves explicitly enumerating the nodes to build the product model of the system if the system consists of more than one module as defined in Section 5.2.3. The explicit method proceeds through a breadth-first search of the nodes of the PTA modules. We start with the initial set of nodes, enumerating all possible combined transitions (synchronised or independent) as defined in Section 5.2.3, and compute all the nodes which could be reached via a single transition. Then we enumerate each of the newly found reachable nodes in the previous iterations to find out all the transitions and their corresponding reachable nodes. The iterated searching continues until there are no new reachable nodes.

Input Language for the PTA Model. Before describing our explicit implementation approach, a simple example of our language for modelling a probabilistic timed system is given in Figure 6.1. This notation is based on the non-deterministic guarded command language of [LPY97a] extended with probabilities. The textual description of a probabilistic timed system consists of a list of nodes. Except the last node which is closed with "*" instead of }, each node is enclosed in { } and includes its name and invariant. If a node is labelled with the keyword "init", this means that the node is a start node. Underneath each node, there are transitions. For each transition, we have the target node's name followed by a guard condition, the list of reset clocks and probability. Transitions are grouped together if they belong to the same distribution. Any line which begins with the character "#" is a comment and is ignored by the parser.

```
#comments
{
init
    node close; true
    [
    tran close; true; x=0; 0.01
    tran open; true; x=0; 0.99
    ]
}
{
    node open; x<3
    [
    tran open; x>2; x=0; 0.005
    tran close; x>2; x=0; 0.995
    ]
*
```

Figure 6.1: The textual description of the PTA in Figure 5.1

As mentioned in Chapter 5, the parallel composition of several interacting sub-components is often useful when defining complex systems. However, in order to support such a parallel composition of modelling, both the notation for describing the systems and the program that inputs its textual description have to be expanded. Instead of reading one single model file, the program will accept a set of files, each containing a single probabilistic timed automaton described in the above language. However, if there is synchronisation between any two automata, the textual representation of transitions should be modified to represent this. Below we describe how to expand the notation by adding additional labels to transition commands. Note that the components of each transition are the same as those of non-synchronised transitions and thus do not need to change. The only modification is the addition a synchronisation action before the keyword "tran" in both model files for which synchronisation is needed. The actions must be the same name as defined in Section 5.2.3 so that two or more automata can synchronise on the same synchronisation actions.

6.1.2 Explicit Implementation of the Forward Algorithm

In this section we describe the explicit implementation, which is given in Figure 6.2, of the forward algorithm in Figure 5.2. The algorithm takes four parameters: a list of files for model description (*modelDescription*), max constant appearing in the model (*MaxConst*), the initial node (*initialStates*) and target node (*targetStates*). Line 1 constructs the product model from the text files of our input language in the format given above. Lines 2-4 initialise the result set of reachable states with the initial states. Lines 5-15 generate the finite-state graph in the form of a Markov decision process, where each generated state is obtained by iterating the *post* operation in line 9. The edges of the graph are inherited from the constructed model, for example, carrying the same discrete probability distribution, and stored in the current processed states in line 11. Line 12 appends the newly found states to the tail of the result set for further exploration. Finally, line 16 writes the resulting MDP and the reachability property as a PCTL formula in the form of the PRISM input language.

```
ForwardModelChecking(modelDescription, MaxConst, initialStates, targetStates)
1.  M := constructModel(modelDescription)
2.  ResultList := ∅
3.  ResultList.append(initialStates)
4.  index2FrontList := 0;
5.  while (index2FrontList < ResultList.size())
6.      currentState := ResultList.get(index2FrontList, M)
7.      index2FrontList + +
8.      for each e ∈ getTransitionOfPTA(currentState, M)
9.          nextState := currentState.post(e, MaxConst)
10.         if ResultList.notIn(nextState) == True
11.             currentState.addTransition(e, nextState)
12.             ResultList.append(nextState)
13.         end if
14.     end for each
15. endwhile
16. return WriteAsPrism(ResultList, initialStates, targetStates)
```

Figure 6.2: Forward probabilistic reachability algorithm

6.1.3 Model Checking For Reachability Properties

The reachability checking is very easy in our explicit method. We just hand over the generated resulting output of the forward implementation to PRISM, which then computes the maximum probabilities. The issue of integration with the PRISM software system is to be addressed in our symbolic approach for the implementation in Chapter 8.

6.2 Experimental Results

We have implemented the explicit forward reachability algorithm in Java and applied it to some case studies. The timing information is represented as DBMs. All case studies in this thesis, including in this and the following chapters, are performed on a Linux machine (Kernel 2.4.18) with configuration of 2792 MHz CPU and 1 Gigabyte of memory. The time units are seconds.

In Table 6.1 and Table 6.2, we present results for two case studies, the abstract and full model of FireWire root contention protocol (the Tree Identify Protocol of the IEEE 1394 High Performance Serial Bus modelled in [SV99]). The abstract model contains only one module, whereas the full model consists of four sub-components modelled as probabilistic timed automata composed in parallel see appendix A.2.

In the tables in this and the following chapters, "-" denotes that the data is not attempted and "*" that the data is not available because the memory capability of PRISM is exceeded. The probability values are omitted since they all agree with those calculated previously by other methods [DKN02, KNS03a, DKN04].

In both cases, the property verified is the minimum probability that, from the initial state, a leader (root) is chosen before the deadline is reached. The algorithm can only compute an upper bound on the maximum probability. The minimum probability is obtained by, firstly, observing that the probability to reach the leader being elected is one. Secondly, a modification of the original model is made: a node is added to the model and a transition from the node labelled with leader elected to the new node, and the enabling condition for the new transition is when the global clock reading is greater than the deadline. Finally we take one minus the calculated maximum probability for the new

node from the initial node. A proof that this is what the required minimum probability is given in [KNS03a].

For the case studies, we give the size of the MDP (number of states generated), the time spent on generating the MDP via the forward algorithm, the time for constructing the MDP from the textual file of the PRISM input language and the time for model checking the property. In Table 6.1 and Table 6.2, the first column gives different parameters for the deadline; the second column is for the size of MDP and the remaining columns are for the time spent on verification, which includes three sub-columns for time spent on forward exploration to generate the reachable graph in a textual file, model construction using PRISM from textual files and model checking probabilistic reachability using PRISM.

From the table for these examples, we see that model checking is very fast, which is due to the efficiency of the symbolic approach adopted by PRISM and because the size of MDP are small. The time for the generation of the reachable graph for both models increases steadily for small values of deadline. However, it would increase greatly for larger deadline. For example, an increase from the deadline 10000 to 20000 in the case of the full model, results in more than a 5-fold increase in the time spent on the generation of the reachable graph and a 4-fold increase in the number of states. This increase in time is explained as follows. One characteristic of the forward algorithm is that each newly generated symbolic state has to be compared with previously generated ones. The function to compare two symbolic states consists of two steps: one for comparison of the discrete parts and the other for timing information which is represented as DBMs. The dominant factor of comparing symbolic states are determinated by the comparison of DBMs because, firstly, one drawback of our DBM implementation is that there is no hash function for quickly locating the same DBM, and secondly, the cost of comparison operation on a $n \times n$ matrix for DBMs would have time-complexity $O(n^2)$.

The process of constructing the MDP from the textual file uses more time than the other parts of the verification process. Both cases show that the time spent on model construction increases as the number of the generated states increases. When the number of states is small, for example, less than 2000, the construction time increases steadily. However, it increases sharply when the number of states is more than 6500. This would

further worsen, as shown in the case of the full model, where the process of construction would fail because PRISM runs out of the memory when the number of states is as high as 96592.

The fact that PRISM runs out of memory during the model construction shown in Table 6.2 is due to the following factors. Firstly, the generated MDP in the form of the PRISM input language does not take any account of the structure information contained in the states. Secondly, the model has 96592 states and more than 60000 lines of transitions. It is known that PRISM can deal with as many as 10^{13} states if the model is expressed in such a way that the structure is exploited within the corresponding symbolic representation in MTBDDs, which is not addressed in this chapter, see Chapter 8. Our explicit approach is divided into two steps. However, the second step could be merged with the first one by directly constructing the MDP in the form of an MTBDDs as done in the PRISM model checker. This justifies our motivation to adopt a symbolic technique and implement the full process in a single software tool.

Discussion The other papers concerning the forward probabilistic reachability implementation are [DKN02, KNS03a, DKN04]. [DKN02], which was using KRONOS [KRO] for the generation of the reachable graph, only contains the test results for the abstract model of FireWire. In [KNS03a], HYTECH [HYT] was used to generate the reachable graph, but it also included the results for the full model of FireWire using the digital clock semantics. Both papers show a similar pattern for the abstract model of FireWire, that is, the number of states of the generated reachable graph grows when the value of the deadline increases. Although [DKN04] includes the verification results for the full model of FireWire, the difference from our case studies is that our experiment presents more results concerning the time spent on the model construction from the generated reachable graph into MTBDD representation using PRISM and shows that large generated reachable graph without any reduction would cause PRISM to fail during model construction phase. One contribution of [DKN02, DKN04] is that the authors proposed an encoding method to reduce the number of the transitions between symbolic states in order to exploit the structural information among those symbolic states having same discrete parts. In Chapter 8, we will describe how this idea can be adapted into our symbolic imple-

mentation and discuss the adaptation. Another method adopted by [DKN04] is using bi-simulation to further reduce the size of generated reachable graph which is not covered by out work. Finally, we remark that the explicit approach to construct the product model used in this chapter is known to be inefficient and infeasible for large models. However, we are interested in the process and implementation of the forward state exploration and construction of the MDP, rather than the process construction of the models for PTA, which is to be addressed in our symbolic approach in Chapter 8.

6.3 Summary

The explicit implementation of the forward algorithm can be summarised as follows. Although the results are based on two examples, the implementation demonstrates that forward model checking of maximum probabilistic reachability for probabilistic timed automata is feasible for small examples. If the generated MDP model contains many states and transitions without considering the reduction of the state space, the use of intermediate textual files to store the descriptions of zone graph can cause PRISM to run out of memory because of the size of the MTBDDs.

Table 6.1: Verification of the abstract model I_1^p with wire delay set to 360 ns

Deadline	States	Time(Explicit)		
		Forward	Construct.	M.C.
2000	64	0.151	0.094	0.009
2500	88	0.182	0.141	0.012
3000	88	0.184	0.140	0.011
3500	124	0.243	0.234	0.013
4000	162	0.270	0.433	0.016
4500	159	0.274	0.418	0.016
5000	208	0.328	0.676	0.019
5500	244	0.367	0.910	0.020
6000	253	0.376	0.973	0.022
7000	348	0.454	2.152	0.025
8000	438	0.525	3.371	0.030
9000	506	0.601	4.567	0.033
10000	609	0.710	7.402	0.043
20000	2124	2.944	117.724	0.128
30000	4546	9.953	612.743	0.296
40000	7851	23.691	1853.573	1.546
50000	12094	48.485	4992.301	2.760
60000	17231	86.477	11245.264	1.367
70000	23305	141.767	-	-
80000	30251	227.807	-	-

Table 6.2: Verification of the full model $Impl^p$ with wire delay set to 360 ns

Deadline	States	Time(Explicit)		
		Forward	Construct.	M.C.
2000	951	5.583	15.020	0.050
2500	1415	8.671	37.799	0.067
3000	1425	8.768	38.125	0.067
3500	2092	12.960	95.042	0.106
4000	2803	17.480	170.284	0.140
4500	2799	18.036	168.851	0.147
5000	3725	24.155	301.363	0.175
5500	4432	28.989	475.884	0.228
6000	4675	31.052	528.828	0.255
7000	6545	45.309	1044.781	0.347
8000	8437	60.079	1963.446	0.447
9000	9879	72.999	2687.657	0.558
10000	11988	91.097	3987.859	0.709
20000	44335	490.927	75078.248	4.181
30000	96592	1551.899	*	-
40000	168514	3820.440	*	-

Chapter 7

Explicit Implementation of the Backward Exploration Algorithm

In this chapter we still consider an explicit implementation, but now for the implementation of the backward algorithm for the first time (a preliminary version was reported in [KNSW04]). As in the previous chapter, the purpose of this implementation is to provide a proof-of-concept implementation and identify the possible bottlenecks of the problem of model checking for probabilistic timed automata via the backward algorithm. We again use explicit data structures such as an adjacency list or array to store the set of states and there is no sharing among them as before. There are two goals we aim to achieve when applying the backward algorithms. One is to obtain the exact maximum probability, as opposed to just an upper bound, and the other is to obtain the minimum probability. However, when we turn to model checking the property of the minimum probabilistic reachability via the backward algorithm, we discover that non-convex zones are commonly encountered during the calculation. Difference Bound Matrices (DBMs) have no native support for non-convex zones. One data structure which supports non-convex zones is Difference Decision Diagrams (DDDs) which has been made available for us to use and is provided as a C library. We have implemented the explicit backward reachability algorithm in both Java and the C programming language using two representations of timing information as DBMs and DDDs. The Java version uses DBMs while the C version uses DDDs for representing the timing information.

This chapter starts with the description of the implementation of the backward algorithm in an explicit manner in Section 7.1. The results are presented in Section 7.2.

7.1 Implementation

Firstly, as in the previous chapter describing the explicit implementation of the forward algorithm, our explicit implementation of the backward algorithm reads in a model of a probabilistic timed automaton from one or more files in the notation described in Section 6.1. Secondly, the same method is applied to parse the model files and store the model in a temporary data structure. Then the tool decides which algorithm to call for calculating the maximum or the minimum probability according to the command line arguments. Next, the backward algorithm is used to generate the model in the form of an MDP, which is translated into the PRISM input language. Finally, model checking is performed using PRISM on the property against the generated model .

7.1.1 Explicit Implementation of the Backward Algorithm

The implementation of the backward maximum probability algorithm in Figure 5.5 for generating the reachable graph is given in Figure 7.1. It accepts three parameters: the model M and two set of symbolic states U and V stored as lists. Line 1 creates the union of sets U and V. Lines 2-3 initialise the result set with timed predecessor of the set V. Line 5 sets the index value to 0. Lines 6-29 generate the finite-state graph, the states of which are obtained in lines 10-14 and lines 17-27. The states generated in lines 10-14 are obtained by first iterating timed and discrete predecessor operations, and then removing those are not in set U. Lines 17-27 calculate the *relevant* states by set conjunction operation on states newly obtained in line 10-14 and states explored so far. The edges between states are added at the same time when the states are generated. Line 30 writes the result in the form of the PRISM input language.

The implementation of the $MaxV_{\geq 1}$ algorithm in Figure 5.8 is given in Figure 7.2. It accepts the following parameters: the integer value c, and two set of symbolic states U and V stored as lists. Line 1 initialises the result set to **true**. Line 2 initialises the

temporary set X to the empty set. Lines 3-9 contain the loop for computing the greatest fixpoint. Inside the loop, the sets X and U will be enhanced with zone $\{z \geq 0\}$ in lines 5 and 6. Line 7 invokes the $MaxU_{\geq 1}$. The states generated in Line 8 are obtained by firstly, computing $z.\phi$ where ϕ is the result of $MaxU_{\geq 1}$ in Line 7 through performing the conjunction of result of Line 7 and set $\{z = 0\}$ and removing the atomic zone involving clock z, and secondly, removing those not in set V from result of previous step. Finally, Line 10 returns the result set Z.

The implementation of the $MaxU_{\geq 1}$ algorithm in Figure 5.7 is given in Figure 7.3. It accepts two set of symbolic states U and V stored as lists. Line 1 gets the union of sets U and V. Line 2 initialises result set Z_0 to set V. Line 3 initialises set Y_0 to empty one. Line 4-16 is the two nested loops for the computation of double fixpoint. Note that the first iteration of both inner and outer loops will always return set equal to V, thus the implementation can be speeded up by initialising both sets Z_0 and Z_1 to set V, which is reflected in Lines 2 and 6. Lines 10 and 12 compute the discrete and time predecessor states, by calling $pre1$ and $tpre$ algorithm, respectively. Line 17 returns the result set.

The implementation of the $pre_{\geq 1}$ algorithm in Figure 5.6 is given in Figure 7.4. It accepts two set of symbolic states U and V stored as lists. Line 1 initialise the result set to empty one. The outer loop in Lines 2-10 goes through each distribution in the model and computes those states can reach set V and set U only with probability one, which is achieved by inner loop in Lines 5-8. Line 11 returns the result Y.

7.2 Experimental Results

In this section, we provide the experimental results for model checking the property of maximum and minimum probability of reachability via the backward approach, which are included in subsection 7.2.1 and 7.2.2 respectively.

We present results for the abstract and full model of FireWire root contention protocol, as in the previous chapter. We also give results for the IEEE 802.3 CSMA/CD (Carrier Sense, Multiple Access with Collision Detection) protocol see appendix A.2.

For the case studies, we give the size of the MDP and the time spent on generating the MDP via the backward algorithm, which are shown in the table under the column

```
BackwardModelChecking(M, StatesList_U, StatesList_V)
1.  StatesList_UV = union(StatesList_U, StatesList_V)
2.  StatesList_Z = tpre(StatesList_UV, StatesList_V)
3.  StatesList := StatesList_Z
5.  index := 0;
6.  while (index < StatesList.size())
7.      currentState := StatesList.get(index, M)
8.      index++
9.      for each e ∈ getTransitionOfPTA(currentState, M)
10.         tmpState := dpre(e, tpre(StatesList_UV, currentState))
11.         preState := conjunct(tmpState, StatesList_U)
12.         if StatesList.notIn(preState) == True
13.             preState.addTransition(e, currentState)
14.             StatesList.append(preState)
15.         end if
16.         idx := 0;
17.         while (idx < StatesList.size())
18.         tmpState := StatesList.get(idx, M)
19.         andState := conjunct(tmpState, preState)
20.         for each e2 ∈ getTransitionOfPTA(tmpState, M)
21.             tmp2State := StatesList.get(tmpState, e2)
22.             StatesList.append(andState)
23.             andState.addTransition(e, currentState)
24.             andState.addTransition(e2, tmp2State)
25.         end for each
26.         idx++;
27.         endwhile
28.     end for each
29. endwhile
30. return WriteAsPrism(StatesList, initialStates, targetStates)
```

Figure 7.1: Backward probabilistic reachability algorithm

$MaxV_{\geq 1}(c, StatesList_U, StatesList_V)$	
1.	$StatesList_Z := \{\mathbf{true}\}$
2.	$StatesList_X := \emptyset$
3.	**while** $(StatesList_X \neq StatesList_Z)$
4.	$\quad StatesList_X := StatesList_Z$
5.	$\quad StatesList_{X_z} := conjunct(z \geq 0, StatesList_X)$
6.	$\quad StatesList_{UX_z} := conjunct(z \geq 0, conjunct(StatesList_U, StatesList_X))$
7.	$\quad StatesList_{Z_z} := MaxU_{\geq 1}(StatesList_{X_z}, disjunct(StatesList_{UX_z}, \{z > c\}))$
8.	$\quad StatesList_Z := conjunct(StatesList_V, Exists(z, conjunct(z = 0, StatesList_{Z_z}))$
9.	**endwhile**
10.	**return** $StatesList_Z$

Figure 7.2: The $MaxV_{\geq 1}$ algorithm

labelled "States" and column labelled "Time" respectively. The unit of time is second. In the tables, the term "DDD" refers to the implementation which uses DDDs and "DBM" refers to the implementation which uses DBMs for representing the timing information. For the convenience of comparison, we also list both the results obtained from backward and forward algorithm, which are under the column of "Forward" and "Backward" respectively.

7.2.1 Maximum Probabilistic Reachability

The abstract and full FireWire models used in this section are same as in the Chapter 6. The property verified is also same, which is the minimum probability that, from the initial state, a leader (root) is chosen before the deadline D is reached. Once again, the minimum probability is calculated by taking one minus the calculated maximum probability to reach the added node from the initial node by using the same modified model used in previous chapter. However, one difference from the Chapter 6 is that the algorithm to calculate the maximum probability here proceeds by the backward search.

The property verified for the CSMA/CD case study is the maximum probability of

$MaxU_{\geq 1}(StatesList_U, StatesList_V)$	
1.	$StatesList_{UV} := disjunct(StatesList_U, StatesList_V)$
2.	$StatesList_{Z_0} := StatesList_V$
3.	$StatesList_{Y_0} := \emptyset$
4.	**while** $(StatesList_{Y_0} \neq StatesList_{Z_0})$
5.	$\quad StatesList_{Y_0} := StatesList_{Z_0}$
6.	$\quad StatesList_{Z_1} := StatesList_V$
7.	$\quad StatesList_{Y_1} := \{\mathbf{true}\}$
8.	$\quad$ **while** $(StatesList_{Y_1} \neq StatesList_{Z_1})$
9.	$\quad\quad StatesList_{Y_1} := StatesList_{Z_1}$
10.	$\quad\quad StatesList_{tmp} := conjunct(StatesList_U, pre1(StatesList_{Y_0}, StatesList_{Y_1})$
11.	$\quad\quad StatesList_{Z_1} := disjunct(StatesList_V, StatesList_{tmp})$
12.	$\quad\quad StatesList_{tmp} := tpre(StatesList_{UV}, conjunct(StatesList_{Y_0}, StatesList_{Y_1}))$
13.	$\quad\quad StatesList_{Z_1} := disjunct(StatesList_{Z_1}, StatesList_{tmp})$
14.	$\quad$ **endwhile**
15.	$StatesList_{Z_0} := StatesList_{Z_1}$
16.	**endwhile**
17.	**return** $StatesList_{Z_0}$

Figure 7.3: $TheMaxU_{\geq 1}$ algorithm

$pre1(StatesList_U, StatesList_V)$

1. $StatesList_Y := \{\textbf{false}\}$
2. **for each Distribution** $dist \in getDistributionOfPTA(M)$
3. $\quad StatesList_{Y_0} := \{\textbf{true}\}$
4. $\quad StatesList_{Y_1} := \{\textbf{false}\}$
5. $\quad$ **for each** $e \in getTransitionOfPTA(M, dist)$
6. $\quad\quad StatesList_{Y_0} := conjunct(dpre(e, StateList_U), StateList_{Y_0})$
6. $\quad\quad StatesList_{Y_1} := disjunct(dpre(e, StateList_V), StateList_{Y_1})$
8. $\quad$ **endfor**
9. $\quad StatesList_Y := disjunct(StatesList_Y, conjunct(StatesList_{Y_0}, StatesList_{Y_1}))$
10. $\quad$ **endfor**
11. **return** $StatesList_Y$

Figure 7.4: The $pre1$ algorithm

both stations correctly delivering their packets by the deadline D. In order to compare the results with those obtained from the forward algorithm, the same model and manner are used as in the Chapter 6, where the model is modified by adding a node, for which the maximum probability is calculated from the initial node.

From Table 7.1, we see that the number of states generated in the MDP using the backward approach is less than that for the forward approach. However, it is not always the case that the backward approach generates smaller MDPs than the forward approach. As we can see from Table 7.2, which contains results for verifying the full model of FireWire, the outcome can be the opposite, in that with the same value of the parameter (deadline), the number of states in the generated MDP for the backward approach is greater than that for the forward approach.

In both cases of the abstract and full model of FireWire, the time spent on the backward search was more than the time spent on the forward search with the same value of the parameter (deadline). This is not surprising because, in the forward approach, when a new zone is found it is only compared with the list of previously found zones to

determine whether it is already explored. The required operation is only zone equality checking. However, in the backward approach, when a new zone is found, it is not only compared with the list of previously found zones, but it must also be used to obtain the relevant zones via the operation of zone conjunction. Then the resulting zones can only be appended to the list if it is not empty set by performing the zone emptiness checking. The implementation of the operation of zone emptiness checking is time-consuming in both DDDs and DBMs. In DBM, it is achieved through bringing the zones into canonical form. In DDDs, similar methods are adopted and this is achieved through checking that there exists at least one path which is feasible, as each path in DDDs corresponds to a DBM. These greatly decrease the performance of the backward traversal approach as the time spent on zone conjunction and emptiness checking depends on the number of states.

An exception to the above is the case of CSMA/CD, in which the backward approach generates fewer states and takes less time than the forward approach. One possible reason behind this case is that the number of states generated from the backward approach is around one thirtieth of that from the forward approach. In the other two cases, the difference in size of the generated MDP between the two approaches is around two to three times.

As the DBM-based version is implemented in Java while the DDD-based is implemented in C, it is not fair to compare the two. However, from the experimental results, we cannot conclude that the performance of the DDD-based version is better than that of DBM-based one in terms of time consumption, or vise versa.

7.2.2 Minimum Probabilistic Reachability

Table 7.4 and 7.5 give the results for model checking the minimum probability directly for the abstract model of FireWire and the full model of CSMA/CD.

Although the property verified for the abstract model of FireWire is the same as in the section 7.2.1, one difference is that the model used here is the original one, that is, the model has not been modified with an added node. This is the same for the full model of CSMA/CD. However, the property verified for CSMA/CD is the minimum probability of both stations correctly delivering their packets by the deadline D instead of the maximum

probability.

From Table 7.4 and Table 7.5, we can see that the generation time using DBMs is considerably less than that for DDDs.

We notice that the number of states is different between the DDD- and DBM-based versions. The version based on DBMs generated fewer states than that based on DDDs. This is because, firstly, one of the characteristics of the backward algorithm is to calculate the zone conjunction when a new zone is encountered, which means that zones are broken down into smaller zones. Secondly, when performing the minimum probability model checking, the zones passed into the algorithm are often non-convex. However, there is no canonical way to break down non-convex zones into a list of DBMs [Tri98]. In DBM representation, the non-convex zones are stored as a list of DBMs, which is non-canonical. On the other hand, in the DDDs representation, the non-convex zones are not broken down at the beginning of the algorithm. This explains the different number of states generated by the algorithms.

We have demonstrated the feasibility of our implementation in Table 7.4 and Table 7.5. However, the models used in these two cases are quite small. When we apply our implementation to the case of the full model of FireWire, it turns out that both implementations based on DBMs and DDDs perform very badly. The computation for DBM-based implementation did not terminate for more than 10 hours for small deadlines, such as 2000; while DDD-based implementation always ran out memory. The reason why the performance of DBM-based version is poor is that the backward algorithm involves three nested fixpoint calculations and, within each iteration, zone equality checking has to be performed. However, for non-convex zones represented as a list of DBMs, the complexity of this operation is very high as it is achieved via zone complementation and conjunction.

In Table 7.5, the number labelled with $*$ means that garbage collection occurred and the data under column "DDD" is collected with good variable ordering on clocks. The same case study, but with bad variable ordering on clocks, can lead to worse performance due to higher occurrence of garbage collection or the system running out of memory.

As the results shown in Table 7.5 indicate, even with good variable ordering on clocks

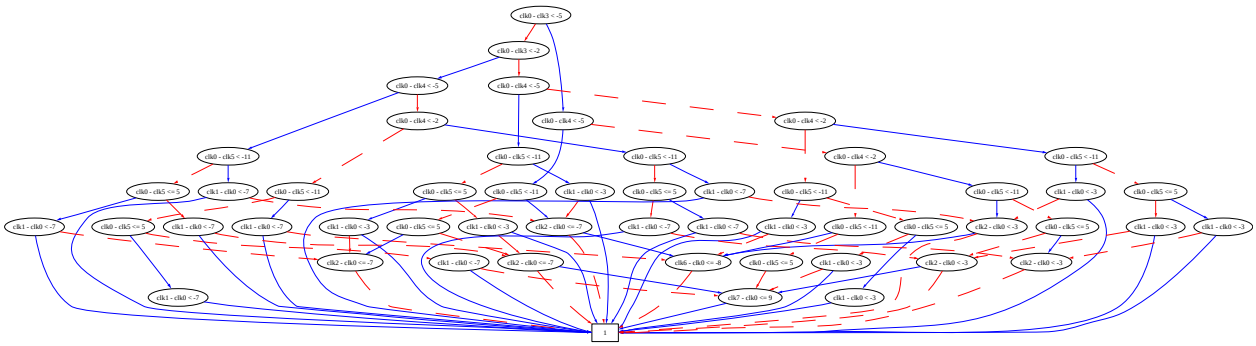

Figure 7.5: DDD ordering example

the the DDD-based version performs much worse than the the DBM-based version. Below we discuss the possible reasons.

7.2.3 Causes of Memory Exhaustion

We notice that besides the high complexity of the backward algorithm, especially the minimum probabilistic reachability algorithm which involves three nested fixpoint calculations, our running machine has only one Gigabyte of memory. In the backward algorithm, the *tpre* operation is a critical operation, especially in the case of computation of the minimum probability. The DBM-based implementation of the *tpre* operation contains double complementation operations. On the other hand, the DDD-based implementation is memory-intensive because it generates many intermediate nodes. DDDs are proposed mainly for dealing with the issue that non-convex zones cannot be directly represented by DBMs. We are interested in understanding why DDDs lead to garbage collection and run out of memory configuration during performing the minimum probabilistic reachability algorithm.

For better understanding of DDDs, below we also compare them to BDDs (ordered). Like BDDs, DDDs can be ordered and are also very sensitive to the ordering of the variables which label the nodes in the DDDs. Figures 7.5 and 7.6 show that the same DDD with different orderings can result in structures where one has nearly three times as many nodes as the other. However, an important difference between the ordered BDDs and ordered DDDs is that ordered BDDs have a canonical form, while ordered DDDs

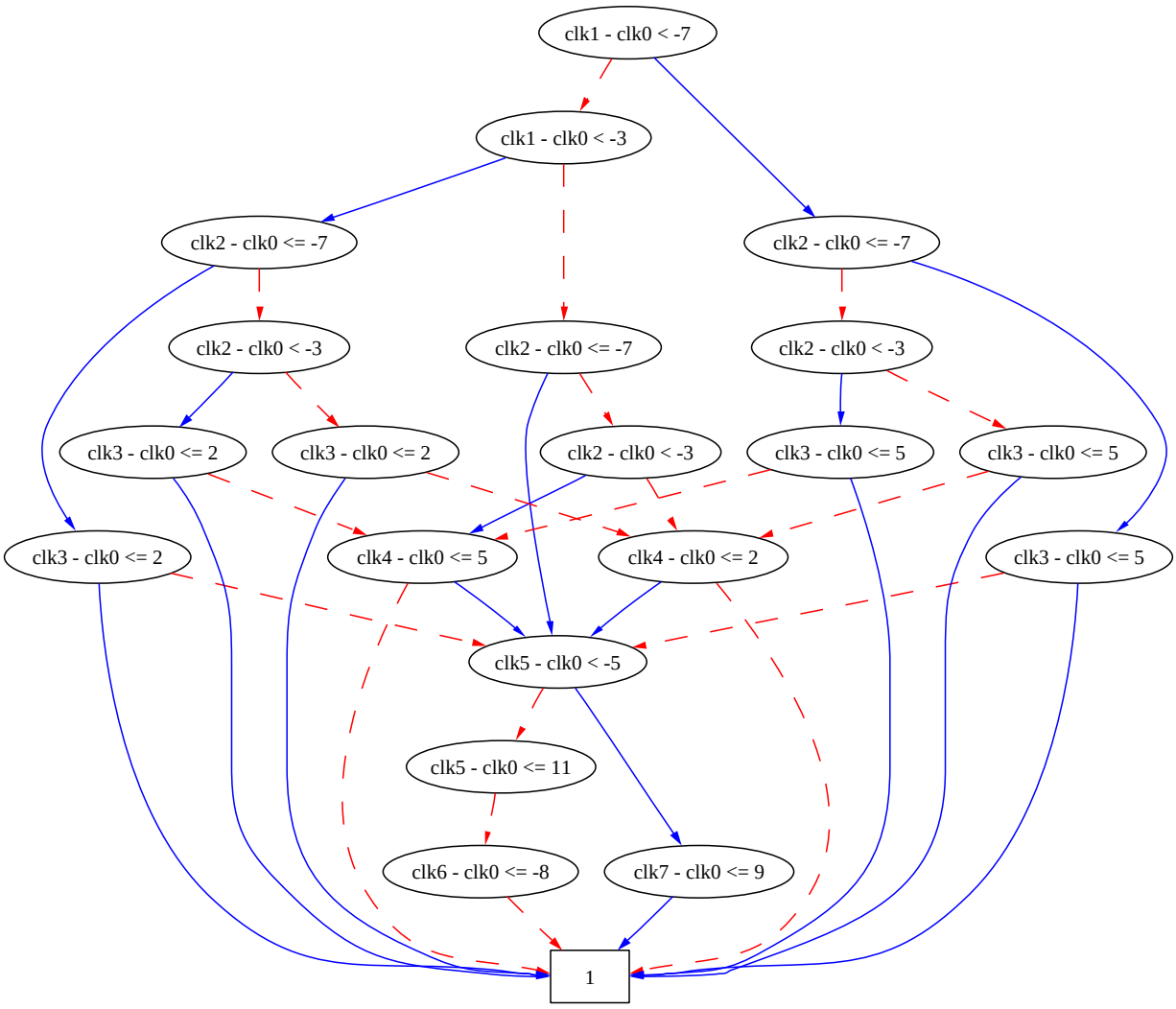

Figure 7.6: DDD ordering example

do not. A Boolean variable is used for labelling the nodes of BDDs, while DDD nodes are labelled by clock difference, which contains a pair of clocks and the upper bounds between these two clocks. However, it is this differences on the node labelling that incurs a significant effect on performance in the manipulation of BDD-like data structures.

Our experiments indicate that, besides the lack of canonicity, DDD operations, such as set union, existential quantification and *tpre*, are the main culprits for generating large numbers of nodes. These would lead to memory exhaustion for DDDs usage.

- The domain of DDDs

  In BDDs, the length of 1-path is bounded by $(n + 1)$ and the size of a BDD is bounded by 2^n where n is the number of Boolean variables or labelling nodes. In DDDs, it is different: the length of 1-path can be arbitrarily long because there can be arbitrarily many tests on the same clock pairs. If we only consider DDDs for representing the zones, the number of labelling nodes along a 1-path in DDDs is determined by both the number of clocks and the maximal constant appearing in the model and the specification, which is bounded by $(2C + 1)^{\frac{1}{2}n(n-1)}$ where n is the number of clocks and C is the maximal constant. Figure 7.7 shows a DDD, one of whose paths labelled with a dashed line originating at the root node shows that the difference between clock clk1 and clock clk0 could have the value in the range $[-2, 2]$.

- The set union in DDDs

  Set union on DDDs could reduce the number of DDD nodes in some cases. However, it could also increase the number of DDD nodes: the more complex non-convex zones are, the more DDD nodes are needed. For example, Figures 7.8, 7.9, 7.10 and 7.11 show four DDDs, each of which has only one 1-path and only four nodes (actually, only two nodes for each as the terminal nodes 1 and 0 are shared by all DDDs). The union of these four DDDs is given as Figure 7.12, which has 18 nodes and 12 paths. This simple union example shows that the number of nodes after union has doubled and the number of paths could be as many as three times the total of the original ones. As the number of variables along one 1-path could be exponential, a DDD after union could also be exponential, which in the worst case could have the

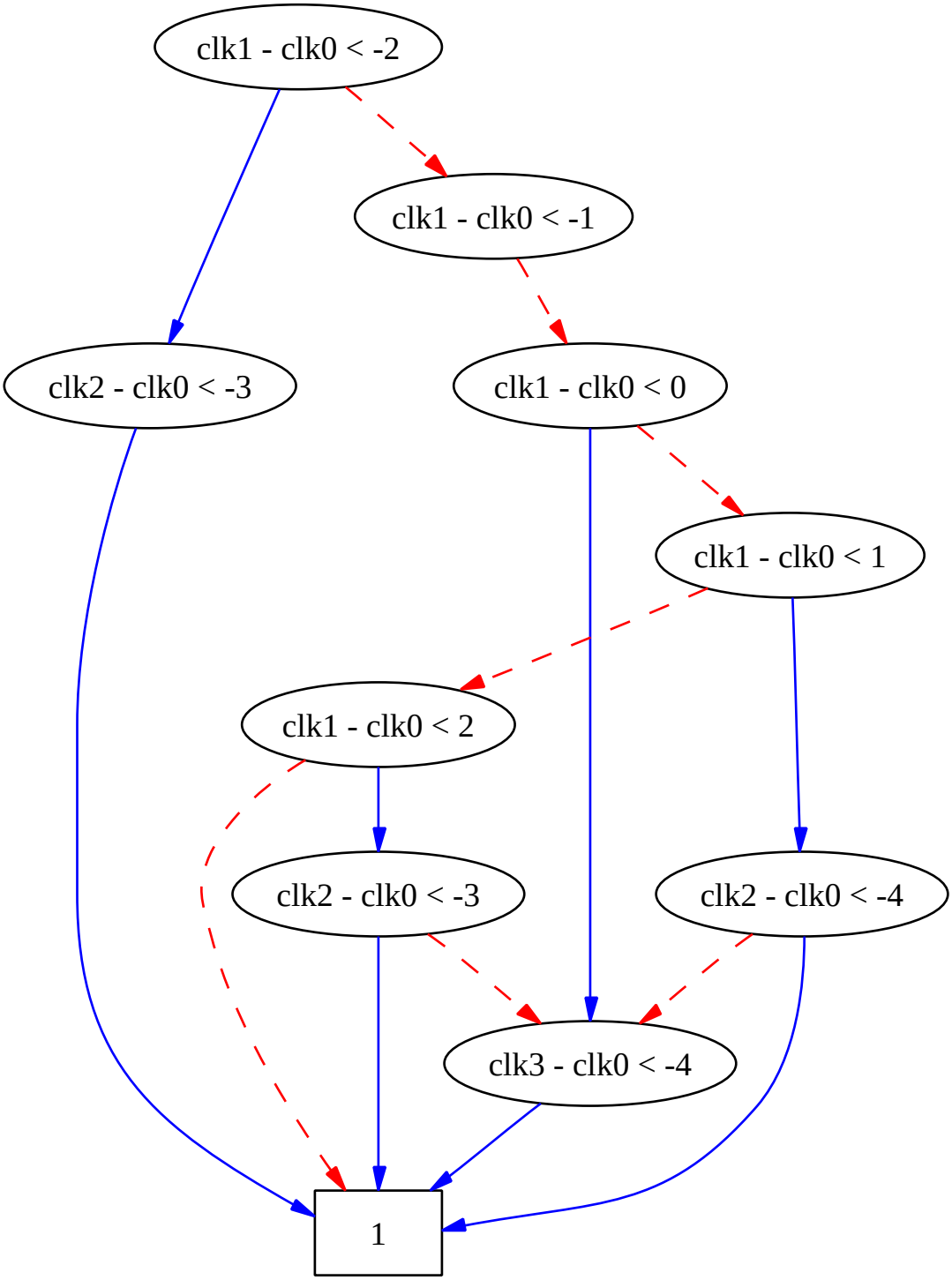

Figure 7.7: DDD domain example

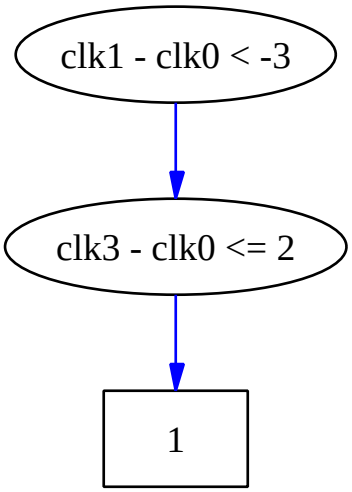

Figure 7.8: DDD union example: single DDD-1

size of $2^{(2C+1)^{\frac{1}{2}n(n-1)}}$.

- Existential quantification

  Existential quantification of a variable x in an expression ϕ, which removes x from ϕ, cannot be done by simply removing all terms from ϕ containing x. We must keep all the implicit clock differences induced by x among the other variables. Suppose we are given an expression ϕ of clock constraints $(x - y \leq 12) \wedge (y - z \leq -5)$ and we want to remove clock y, which is expressed as $\exists y.\phi$, then the resulting expression is $(x - z \leq 7)$. The example given reduces the number of clock constraints from two to one because it it very simple. Below we will give another example which will increase the number of the clock constraints. Figures 7.13 and 7.14 show two DDDs corresponding to the constraints before and after existential quantification. The one after quantification is obtained by removing clock clk4 from the one before.

- Redundant nodes along paths

  In BDDs, node variables are unique along BDD paths leading to terminal node true, and redundant nodes are not allowed along any paths. Unlike BDDs, DDDs have no canonical form; furthermore, the labelling nodes along the DDD paths leading to terminal node true could be redundant. In DDDs, both local reduction and path

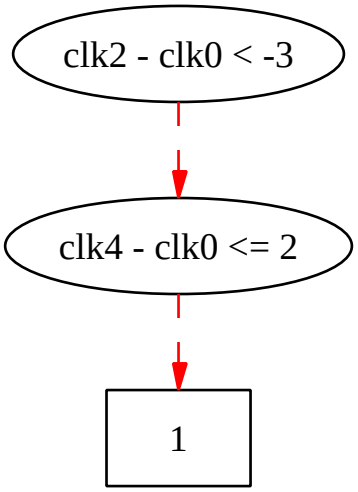

Figure 7.9: DDD union example: single DDD-2

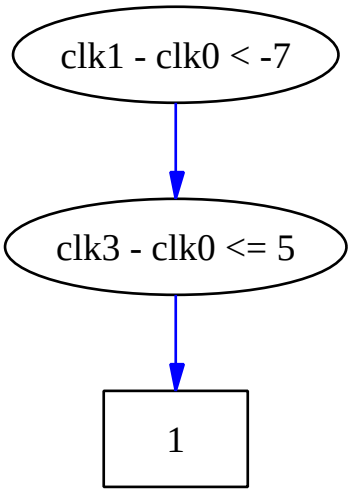

Figure 7.10: DDD union example: single DDD-3

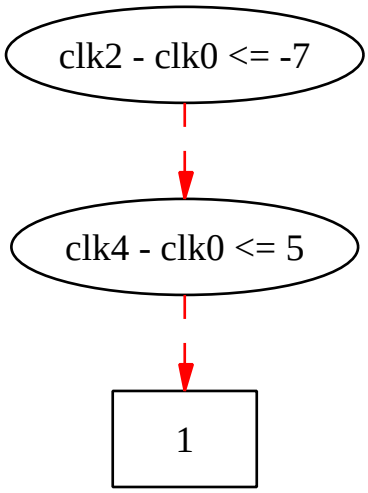

Figure 7.11: DDD union example: single DDD-4

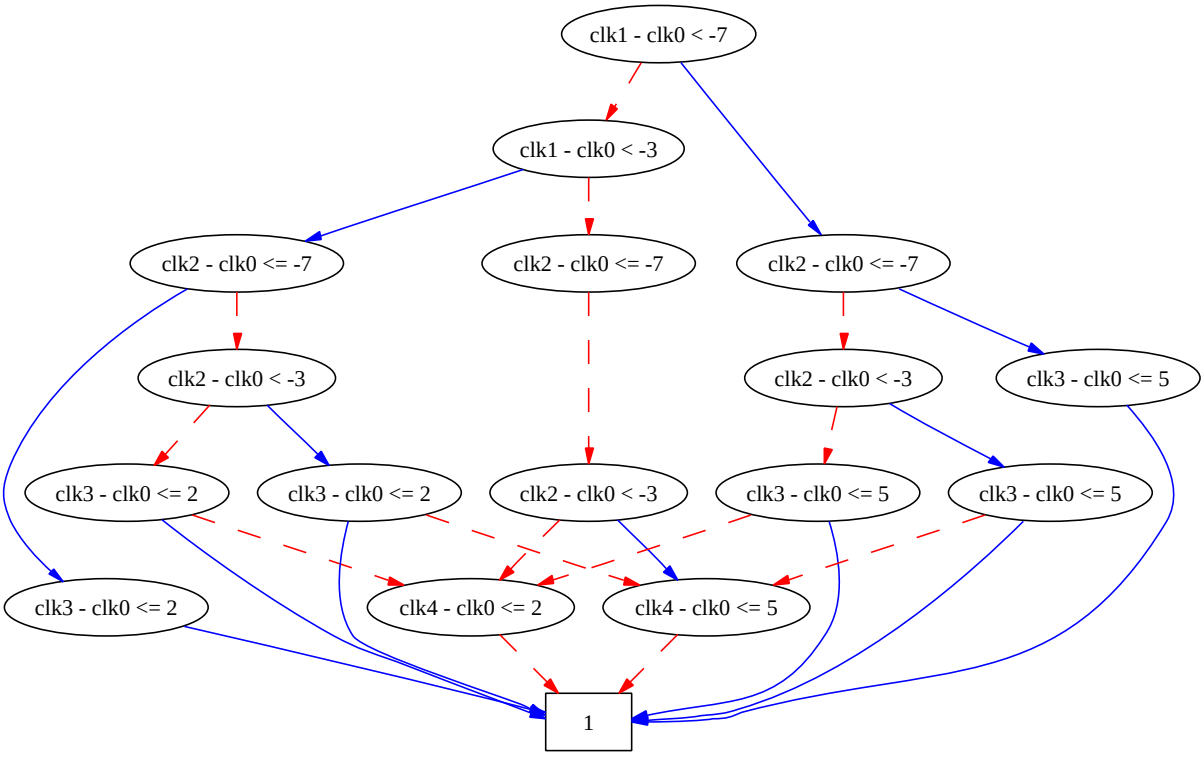

Figure 7.12: DDD union example

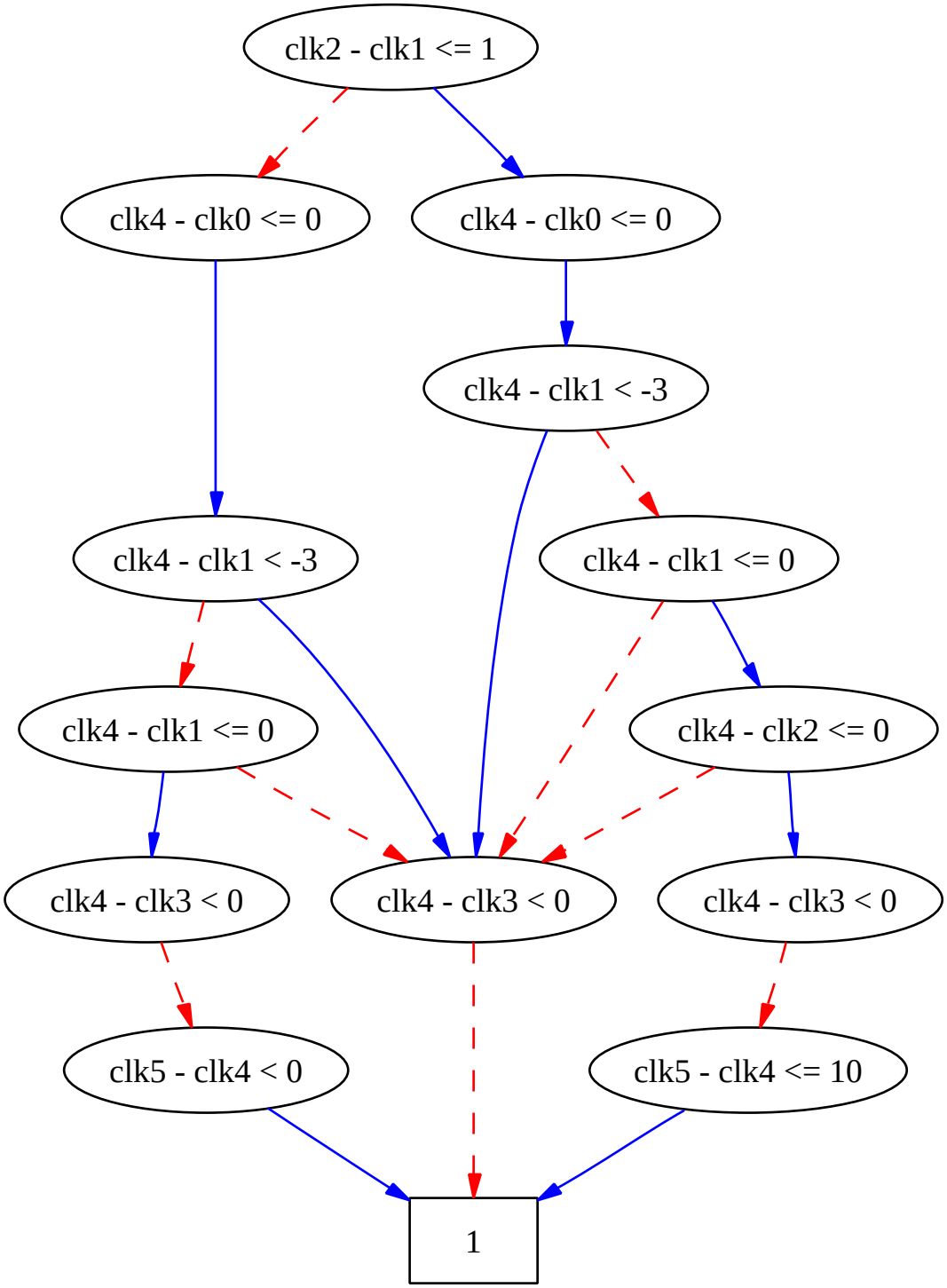

Figure 7.13: DDD example: before existential quantification

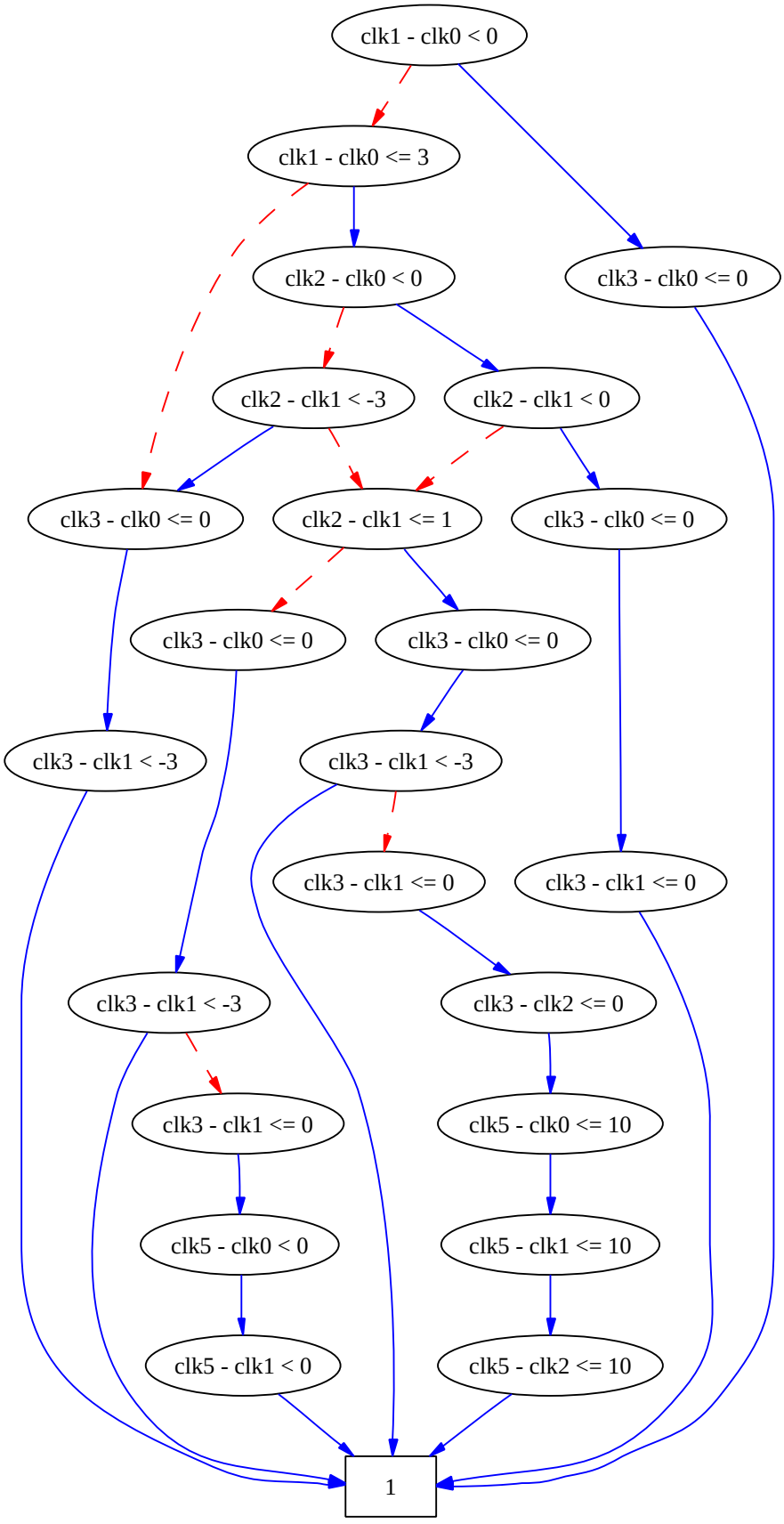

Figure 7.14: DDD example: after existential quantification

reduce [ML98] are defined, which at some level could remove certain redundant nodes along paths. However, during operations on DDDs, redundant nodes could still be generated and it is not possible to remove all the redundant nodes. The reason that not all the redundant nodes along a path can be removed is because, although a node could be regarded as redundant along a path, it is not along another path. The dashed line path originating at the root node in Figure 7.7 shows that, except for the last node, all other nodes are redundant along that path. However, these nodes have to be kept because they are not redundant in other paths. As each path in a DDD can be viewed as a disjunct in the Disjunctive Normal Form (DNF) representation of the DDD, the semantics of a DDD can be treated as the union of each path. In other words, it is the union of DDDs that results in the redundant nodes which could not be totally removed.

Figure 7.15 and Figure 7.16 are two DDDs corresponding to before and after path reduce.

- The DDD-based *tpre* operation is obtained in terms of the logical operations of complementation and set union, or set intersection and existential quantification. However, when the expression is more complex, keeping and making these clock differences explicit means introducing many new nodes in DDDs, which could lead to exponential blow-up.

- The complexity of the *tpre* operation
  The complexity of set union on DDDs is $O(|u|\ |v|)$, while the complexity of existential quantification is in the worst case $O(2^{|u|} \times n^3)$ [ML98]. As the time spent on existential quantification is the dominating factor, the complexity of *tpre* is determined by the complexity of existential quantification, where $|u|$ is the number of nodes in the DDD u and n is the number of clock variables.

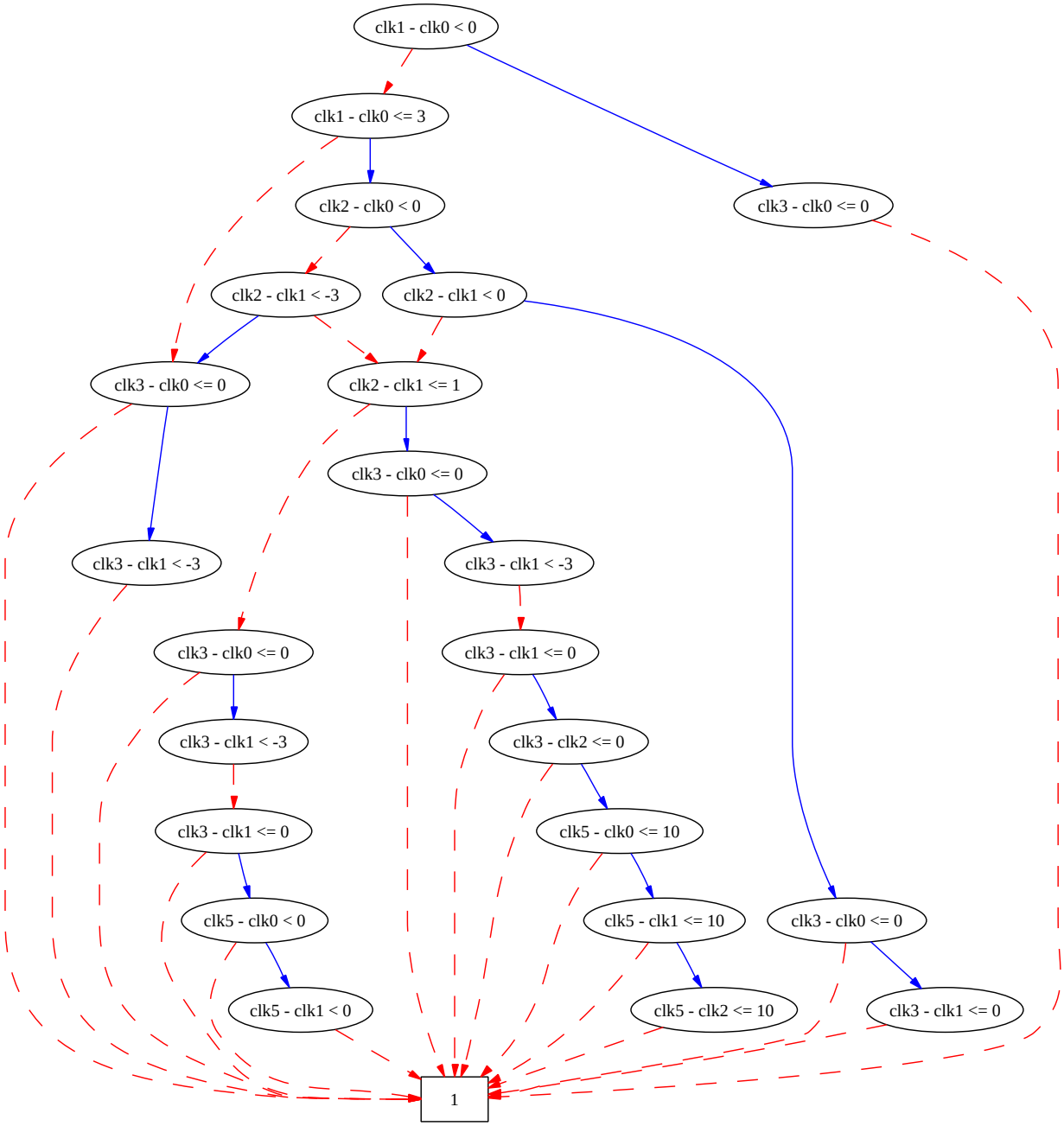

Figure 7.15: DDD example: before path reduce

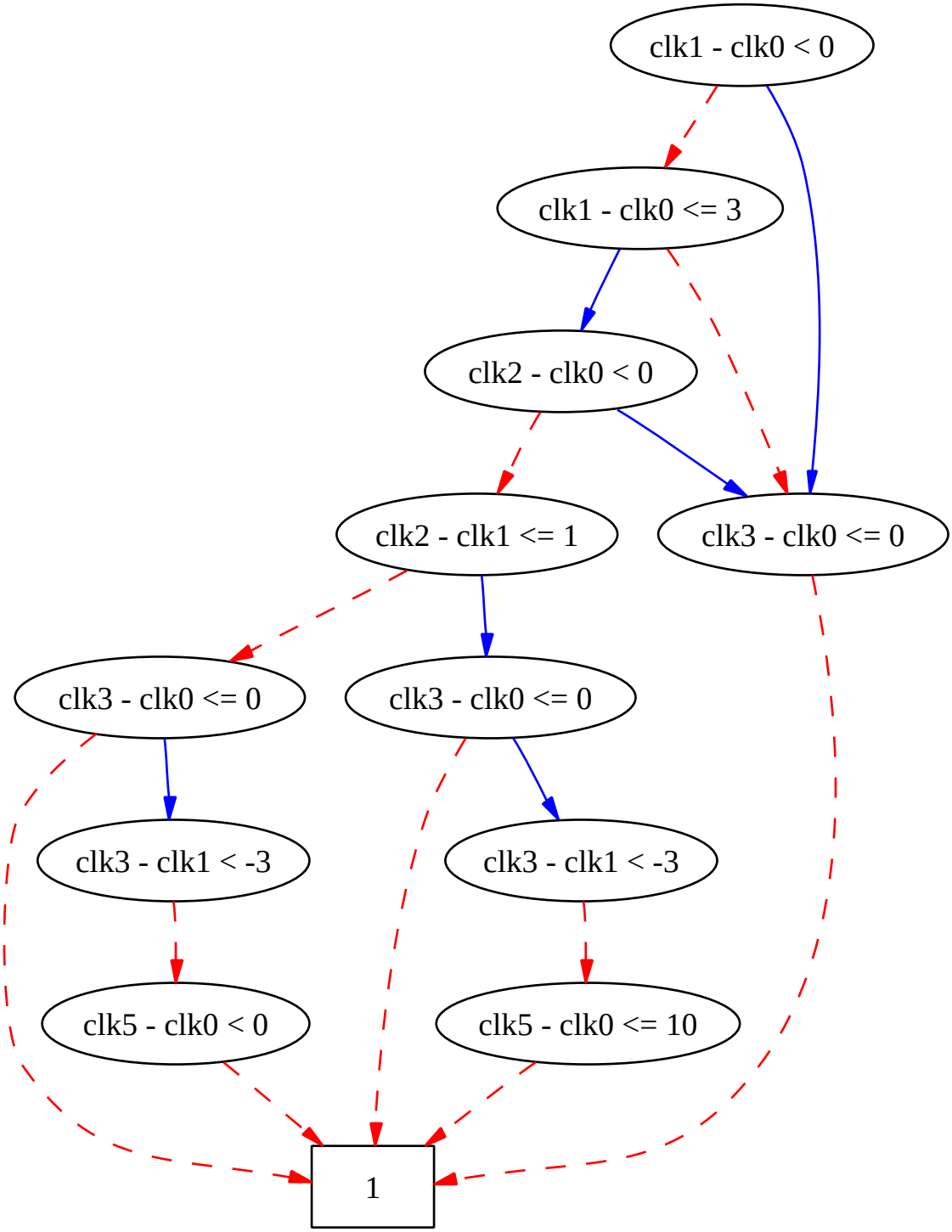

Figure 7.16: DDD example: after path reduce

7.3 Summary

In conclusion, the explicit implementation of the backward algorithm can be summarised as follows.

When calculating the maximum probability, the computational complexity of the backward algorithm is much greater than that of the forward algorithm. This extra complexity is caused by the characteristics of the algorithm by which additional relevant zones have to be calculated via zone conjunction.

When calculating the minimum probability, both DBM-based and DDD-based implementations can only deliver results for small models. For larger models, both perform badly and could fail to deliver results. The experiments also demonstrate that under our running environment, even with the best variable ordering, DDDs require large memory configuration for quite small models.

The poor performance in the case of calculating the minimum probability is due to the high complexity of the algorithm and operations required, especially the *tpre* operation. Thus, a suitable data structure for representing and operating on non-convex zones is key to efficient implementation.

Table 7.1: Verification maximum probability of the abstract model I_1^p with wire delay set to 360 ns

Deadline	Forward		Backward		
	State	Time	State	Time	
				DDD	DBM
2000	64	0.151	25	0.000	0.482
2500	88	0.182	32	1.000	0.261
3000	88	0.184	37	1.000	0.280
3500	124	0.243	42	1.000	0.300
4000	162	0.270	47	1.000	0.323
4500	159	0.274	59	1.000	0.412
5000	208	0.328	66	1.000	0.442
5500	244	0.367	73	1.000	0.474
6000	253	0.376	81	1.000	0.508
7000	348	0.454	107	1.000	0.676
8000	438	0.525	126	1.000	0.811
9000	506	0.601	159	1.000	1.105
10000	609	0.710	183	1.000	1.366
20000	2124	2.944	643	11.000	24.261
30000	4546	9.953	1380	59.000	267.066
40000	7851	23.691	2400	215.000	1726.970
50000	12094	48.485	3714	609.000	9848.939
60000	17231	86.477	5310	1426.000	-
70000	23305	141.767	-	-	-
80000	30251	227.807	-	-	-

Table 7.2: Verification maximum probability of the full model $Impl^p$ with wire delay set to 360 ns

Deadline	Forward		Backward		
	State	Time	State	Time	
				DDD	DBM
2000	951	5.583	1218	365.000	381.969
2500	1415	8.671	2363	4071.000	2694.162
3000	1425	8.768	2952	8798.000	5474.705
3500	2092	12.960	3692	19508.000	-
4000	2803	17.480	-	-	-
4500	2799	18.036	-	-	-

Table 7.3: Verification maximum probability of the full CSMA/CD model (backoff=1)

Deadline	Forward		Backward		
	State	Time	State	Time	
				DDD	DBM
1000	6404	24.145	92	1.000	1.383
1200	9034	39.782	212	2.000	7.793
1400	11771	60.644	332	4.000	20.818
1600	15329	92.919	452	7.000	40.282
1800	19453	137.779	644	8.000	58.730
2000	23468	188.489	758	10.000	61.970
2200	28516	263.348	902	22.000	68.040
2400	34023	353.003	1046	40.000	76.408
2600	39970	467.441	1186	57.000	87.155
2800	45654	591.693	1328	80.000	100.476
3000	52561	756.939	1472	107.000	115.595

Table 7.4: Verification minimum probability of the abstract model I_1^p with wire delay set to 360 ns (min)

Deadline	DBM		DDD	
	States	Time	States	Time
2000	15	3.165	15	2.000
2500	17	2.902	17	2.000
3000	20	3.600	24	2.000
3500	22	3.639	28	2.000
4000	25	3.617	41	2.000
4500	32	3.653	78	2.000
5000	37	3.659	93	2.000
5500	42	3.699	155	3.000
6000	47	3.713	206	4.000
7000	66	3.838	472	37.000
8000	81	3.916	753	186.000
9000	107	4.119	1339	1369.000
10000	126	4.128	1988	4876.000
20000	528	17.874	-	-
30000	1206	165.886	-	-

Table 7.5: Verification minimum probability of the full model CSMA/CD (backoff=1)

Deadline	DBM		DDD	
	States	Time	States	Time
1000	391	21.811	47	225.000
1200	439	25.515	95	250.000
1400	487	31.373	143	291.000
1600	535	39.100	191	361.000
1800	641	43.707	307	481.000
2000	763	47.463	679	804.000
2200	923	56.057	1419	1936.000*
2400	1083	68.606	2417	7008.000*
2600	1223	82.718	3511	17855.000*
2800	1383	104.007	4865	43852.000*
3000	1543	131.009	-	-

Chapter 8

Symbolic Implementation of the Forward Exploration Algorithm

In Chapter 6 and 7, we described an explicit implementation of model checking for probabilistic timed automata. In this chapter, we consider a symbolic implementation. The implementation is based on Multi-terminal Binary Decision Diagrams (MTBDDs). As model checking for probabilistic timed automata involves both probabilistic and timing information, the difficulty is how to efficiently combine the three steps, that is, how to represent the models, perform the necessary operations to generate MDPs over zones, and, finally, perform probabilistic analysis against previously generated MDPs in terms of MTBDDs.

A fundamental issue in symbolic model checking is how to formulate a symbolic representation for both the set of states and the transitions between them, and ensure that this kind of representation admits all the necessary operations which are required to explore the state space.

The outline of this chapter is as follows. In Section 8.1, symbolic implementation of the forward algorithm is introduced. In Section 8.2, we consider how to exploit the facility provided by PRISM to make the model construction feasible and efficient based on our method. In particular, we consider a model description language which is adapted from the PRISM language.

8.1 Symbolic Implementation of Model Checking Probabilistic Timed Automata with MTBDDs

In Section 8.1.1, an encoding method to symbolically represent the syntax of probabilistic timed automata (PTAs) is introduced. We will consider how to use MTBDD functions to encode transitions of PTAs. In Section 8.1.2, we present an MTBDD-based implementation of the forward algorithm. In Section 8.1.3, we consider how to model check MDPs. In Section 8.1.4, we present experimental results for a symbolic implementation of the forward algorithm based on our method. Experimental results are presented and compared with those methods based on explicit model construction.

8.1.1 Symbolic Representation of PTAs with MTBDDs

In this section we describe a symbolic encoding method for probabilistic timed automata based on MTBDDs.

We first discuss possible ways of symbolic implementation of model checking for PTAs and the reason why we choose MTBDDs as our symbolic representation for the models. Next we consider how to use MTBDDs to encode PTAs. In Section 8.1.2, we consider the implementation of the forward algorithm based on our method.

In verifying timed automata, the states usually contain two different parts: the discrete part and the continuous part. The discrete part is usually finite while the continuous part is infinite. In addition, the requirements for operations on these two parts are different. The continuous part requires operations of advancing and resetting clocks. Using DDDs to encode the set of states and the transition relation between states, all the required operations on both parts can be obtained by using only the data structure of DDDs.

However, in the case of probabilistic timed automata, we also need to encode the information about probability distributions in the transition relation, which is not supported by just using DDDs. In the case of the forward algorithm, one needs to inherit the probability for generating the probabilistic transition relation between generated sets of pairs of symbolic states according to the original probabilistic distribution.

Furthermore, there are three problems that need to be dealt with when generating the

probabilistic system representation:

- The size of the state space and number of transitions between the states of the generated probabilistic system is unknown before the algorithm, which dynamically generates these states, terminates. In other words, we need a data structure which could deal with dynamic size of the state space.

- Like non-probabilistic timed systems, the states in the probabilistic systems generated by this algorithm have the form of a pair (l, ζ), where l encodes the discrete part and ζ encodes the continuous part. However, when performing reachability analysis of non-probabilistic timed systems, if we reach two sets of states, for example, (l, ζ_1) and (l, ζ_2), and one of them is a subset of the other, that is, they contain the same discrete parts and one continuous part is a subset of the other, one only needs to store the union of both, that is $(l, \zeta_1 \cup \zeta_2)$ instead of two separate states. However, in the case of probabilistic timed automata, we have to distinguish two sets of generated states even if one of them is a subset of the other. In other words, we have to store many small sets of states separately instead of simply their union.

- The guards on clocks and clock resetting associated with the transition relation in the syntactic models of probabilistic timed automata are different, although the source and target discrete component could be same. This means that the implicit way of applying probabilistic transition relation could not be applied, as in the case of untimed probabilistic model checking. In other words, we can no longer apply the probabilistic transition relation in one go, but have to apply only one probabilistic transition each time and explicitly maintain such information in the generated probabilistic systems.

Thus, there are key problems that have to be addressed in order to use symbolic model checking for probabilistic timed automata:

- How to define a data structure for representing and manipulating the sets of states which contains both a discrete part and a continuous part.

- How to define a data structure for representing the timed transition relation between states.

- How to define a data structure for representing the probabilistic transition relation between states.
- How to define a data structure for constructing the dynamically generated information, states and probabilistic transitions between states.

The explicit implementations in Chapter 6 and 7, MTBDDs are only exploited by applying PRISM tool in the final step of verification. The following section is addressing the above problems by using the data structure that is a combination of MTBDDs and DDDs/DBMs. This is possible since MTBDDs could be treated as functions mapping from indices to reals. For the final step, once the model has been translated into MTBDDs, model checking the generated probabilistic system can be achieved via PRISM.

MTBDDs have been successfully applied for symbolic model checking of untimed probabilistic systems [PRI, KNP02], and specifically Markov decision processes (MDPs). One of the most interesting applications of MTBDDs is their ability to represent both vectors and matrices. Markov decision processes (MDPs) can be treated as non-square matrices. The use of MTBDDs for the representation of MDPs has been discussed in [Par02]. Our motivation is to use MTBDDs for representing both the syntactic models of probabilistic timed automata and the generated MDP from forward exploration because the underlying semantics of labelled probabilistic timed transition systems associated with probabilistic timed automata is a Markov decision process (MDP).

We distinguish two types of models: the syntactic models for probabilistic timed automata and the generated MDPs models obtained via the forward or backward algorithms. The former are static and latter are dynamic.

Representation for Syntactic Models We start by considering how to symbolically represent the syntactic models for probabilistic timed automata using MTBDDs.

For finite untimed probabilistic systems, the method relies on encoding sets of states as BDDs and the (probabilistic) transition relation between these states using MTBDDs, which can be done compactly if there is sufficient regularity in the model.

This method can be adapted to represent the syntactic models for probabilistic timed automata. However, we have to consider the difference in syntax and semantics between

models for finite untimed probabilistic systems and models for probabilistic timed automata.

Firstly, the complete information for the state space of finite untimed probabilistic systems is explicitly contained in the models, while only partial information, which consists of the discrete component of the state space, is explicitly contained in the models of probabilistic timed automata. Secondly, the syntax of models for probabilistic timed automata additionally contains information about timing of the transitions, which consists of guards on clock constraints and clock resetting.

Our starting point is to use MTBDDs to represent a single transition (probabilistic edge) in the syntactic models of probabilistic timed automata.

Recall that each probabilistic edge has the form of the tuple (l, l', g, X, p) where l is the current node, l' is the next node, g is the guard, X is the set of clocks to be reset and p is the probability value. The edges of the generated reachable graph are derived from the corresponding edges of probabilistic timed automata. For convenience, we also include the information about non-deterministic choice, both invariants for current node and next node into our encoding for edges. Thus, each new probabilistic edge has the form of the tuple $(n, l, inv(l), l', inv(l'), g, X, p)$, n is (the encoding of) the non-deterministic choice, $inv(l)$ is the invariant of the current node, $inv(l')$ is the invariant of the next node.

The basic idea behind the classical Boolean logarithmic encoding is that 2^n elements of a finite set could be encoded using n bits.

Our method is to use Boolean vectors to encode all the information appearing in a probabilistic edge. We use separate Boolean vectors which correspond to the discrete component of the state space, the invariants associated with discrete components, the set of guards on clocks (clock constraints) and the set of clocks to be reset.

For the systems consisting of more than one module, the Boolean vector encoding the discrete component, which is finite, is further divided into several groups according to the structure of the system, for example, the number of subcomponents and values of the non-clock variables following well-known heuristics established in [dAKN$^+$00, HKN$^+$03], which groups together those variables that belong to the same module.

Both invariants and guards on clocks appearing in the probabilistic transition are clock

constraints. We store them together. The sets of clock constraints are explicitly stored as a list, which is a separate data structure from the discrete components. Since the set of clock constraints is finite, informally, we use a one-to-one function to assign a unique index to each set of clock constraints and similar logarithmic encoding is applied.

The number of non-deterministic choices in each state is finite and bounded. The maximum number of non-deterministic choices for all states is determined through parallel composition, and can therefore be encoded using the logarithmic encoding.

It remains to encode the set X of clocks to be reset. There are a number of issues to consider when encoding the clock reset operation in the transition relation:

- Recall that each clock $x \in X$ could be set to different values and not simply zero. If we encode each reset in the transition, this means we need two sets of Boolean BDD variables for it: one for the clock and the other for the value that the clock should be set to.

- The total number of clocks to be reset appearing in the transition could vary.

Thus, we opt for a simple approach: a Boolean vector is reserved in the transition for assigning a unique index value to each distinct set X and each set X is explicitly stored as a list.

We have described informally how to encode the syntactic models of PTAs. First, we consider formally the encoding of a PTA without non-deterministic choices. In this case, the syntactic models of PTAs can be treated as a square matrix. Formally, assuming given an encoding of $2^{(l+gr)}$ integers into $(l+gr)$ Boolean variables, i.e. a one-to-one function $enc : \{0, \ldots, 2^{(l+gr)}-1\} \rightarrow \mathbb{B}^n$, we can represent a $2^{(l+gr)} \times 2^{(l+gr)}$ square matrix, $\mathbf{M}$, as a mapping from $\{0, \ldots, 2^{(l+gr)}-1\} \times \{0, \ldots, 2^{(l+gr)}-1\}$ to R by an MTBDD M over $2(l+gr)$ variables, $(l+gr)$ of which encode row indices and $(l+gr)$ of which encode column indices. Using *row variables* $\underline{\mathsf{x}} = (\mathsf{x}_1, \ldots, \mathsf{x}_{(l+gr)})$ and *column variables* $\underline{\mathsf{y}} = (\mathsf{y}_1, \ldots, \mathsf{y}_{(l+gr)})$, we say that M represents $\mathbf{M}$ if and only if $f_{\mathsf{M}}[\underline{\mathsf{x}} = enc(i), \underline{\mathsf{y}} = enc(j)] = \mathbf{M}(i,j)$ for $0 \leq i, j \leq 2^{(l+gr)}-1$. It is convenient to divide both row and column vectors $\underline{\mathsf{x}}$ and $\underline{\mathsf{y}}$ into 3 sub-vectors: $\underline{\mathsf{x}} = (\underline{\mathsf{x}_\mathsf{l}}, \underline{\mathsf{x}_\mathsf{gr}})$ and $\underline{\mathsf{y}} = (\underline{\mathsf{y}_\mathsf{l}}, \underline{\mathsf{y}_\mathsf{gr}})$. Here we use l Boolean variables to encode the set of 2^l nodes, gr Boolean variables to encode the set of 2^g guards or

clock resettings. We use same number of Boolean varibles (gr) for both guards and clock resettings in order to reduce the number of required variables. The upper bound values is the maximum value of the two indices. The interpretation of elements of $\mathbf{M}$ $f_{\mathsf{M}}[\underline{\mathsf{x}_{\mathsf{l}}} = enc(l), \underline{\mathsf{x}_{\mathsf{gr}}} = enc(gr), \underline{\mathsf{y}_{\mathsf{l}}} = enc(l'), \underline{\mathsf{y}_{\mathsf{gr}}} = enc(gr')] = \mathbf{M}(i,j)$ is as follows: if the current node is $[\underline{\mathsf{x}_{\mathsf{l}}} = enc(l)]$ and guard $[\underline{\mathsf{y}_{\mathsf{gr}}} = enc(gr)]$ is satisfied, the transition could be performed to go to next node $[\underline{\mathsf{y}_{\mathsf{l}}} = enc(l')]$ and the clock in the $[\underline{\mathsf{y}_{\mathsf{gr}}} = enc(gr')]$ is reset.

Then we extend this idea to PTAs with non-deterministic choices between probabilistic transitions in their nodes. The syntactic models in this case can be treated as a non-square matrix, and the underlying semantics is an MDP. The idea to use a third index to represent and encode nondeterministic choice in an MDP representing as an MTBDD was proposed in [Bai98] and discussed in [Par02]. Formally, assuming that the maximum number of nondeterministic choices in any node is bounded by 2^n and given an encoding of the 2^n into $\underline{\mathsf{z}} = (\mathsf{z}_1, \ldots, \mathsf{z}_n)$ Boolean variables, i.e. a one-to-one function $enc : \{0, \ldots, 2^n - 1\} \rightarrow B^n$, we can represent a $2^{(l+gr)} \times 2^n \times 2^{(l+gr)}$ non-square matrix, $\mathbf{M}$, as a mapping from $\{0, \ldots, 2^{(l+gr)} - 1\} \times \{0, \ldots, 2^n - 1\} \times \{0, \ldots, 2^{(l+gr)} - 1\}$ to R, by an MTBDD M over $2(l + gr) + n$ variables, $(l + gr)$ of which encode row indices and $(l + gr)$ of which encode column indices and n of which encode the indices to nondeterministic choices.

Figure 8.1 is the MTBBD representation for transition relation after applying our encoding method for probabilistic timed automaton in Figure 5.1. In Figure 8.1, the MTBDD variables "x", "y", "g" and "r" are for source node, target node, guard and clock resetting, respectively. The second left path represents the probabilistic transition of going to node *open* with probability 0.99 from node *close* at any time and clock x to be reset.

Representation for Dynamic Models Now we consider how to symbolically represent the models dynamically generated via the forward algorithms using MTBDDs. For finite untimed probabilistic systems the potential state space is known in advance of constructing the symbolic representation of the model, as it can be deduced from the syntactic model description.

The difficulty with model checking of PTAs is that the size of the state space is unknown beforehand, and states (node-zone pairs) are generated dynamically through the

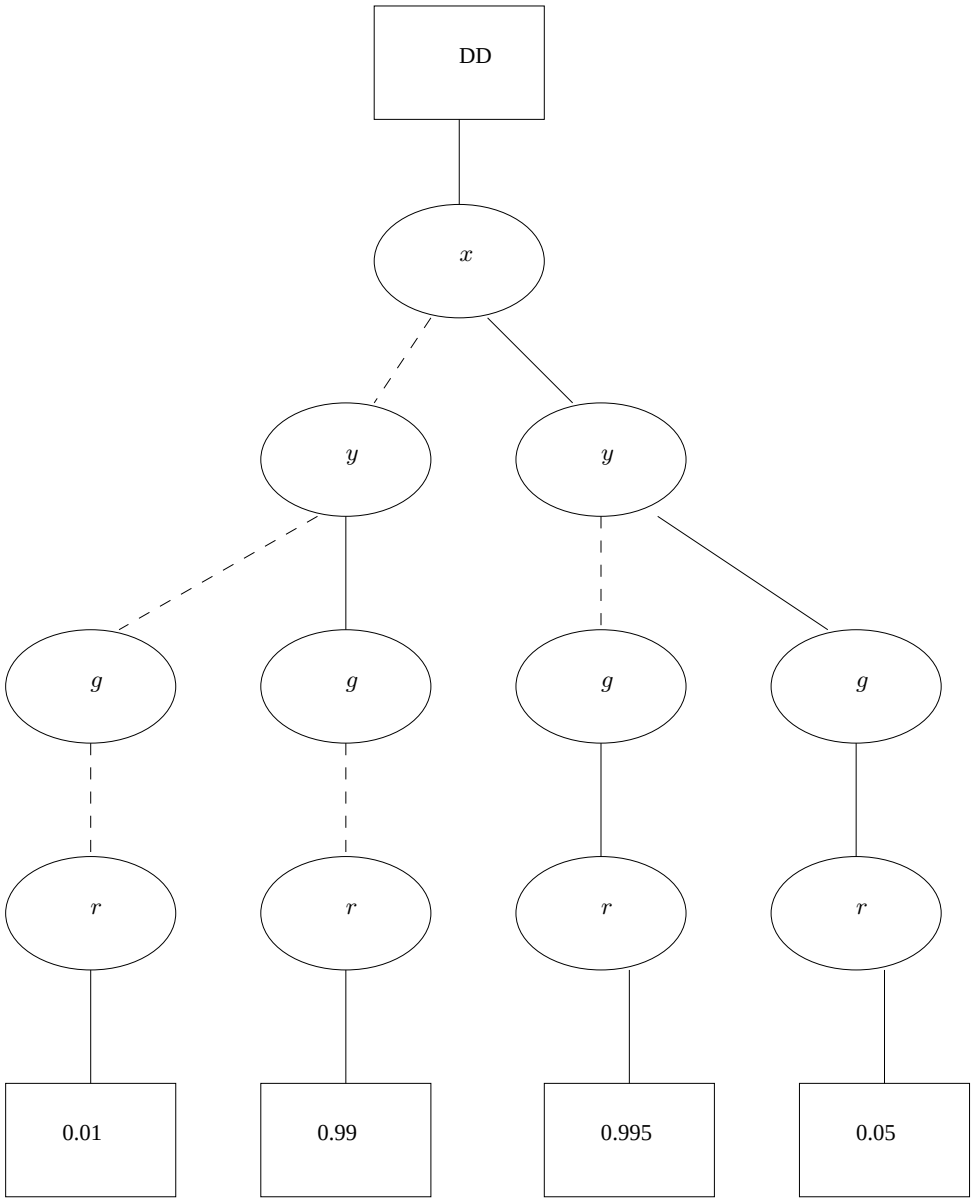

Figure 8.1: Transition relation of Figure 5.1 in MTBDD

process of exploration of the zone graph using timed predecessor or successor operations. The resulting symbolic representation has to be amenable to such dynamic manipulation of the state space.

Below we describe how to encode the *states* and *probabilistic edges* for the generated MDP via the forward algorithm.

Let us first consider how to encode the state space. Each state in the state space has the form of a pair (l, ζ) where $l \in L$ is the discrete part and $\zeta \in Zones(\mathcal{X})$ is the zone. Like the method for encoding the syntactic models of PTAs, we use the same Boolean vectors to encode the discrete component of the pair and a separate Boolean vector for the zone part. However, the number of zones, unfortunately, could be infinite when forward analysis is used. The technique in [BY04] guarantees the termination of the forward reachability search, which means that a finite set of zones could be obtained. Since the set of zones is finite, informally, we use a one-to-one function to assign a unique index to each zone and a logarithmic encoding as above is applied.

After we have encoded the state space, it is easy to derive the encoding for the probabilistic edge for the generated MDP. Each probabilistic edge has the form of the tuple $(n, l, l', \zeta, \zeta', p)$, where n is (the encoding of) the non-deterministic choice, l and l' are the current and next node, ζ, ζ' is the current and next zone and p is the probability value.

8.1.2 Forward Implementation

After presenting our symbolic encoding method for probabilistic timed automata, we consider the application of our method to the probabilistic reachability algorithms. In this section we focus on forward probabilistic reachability analysis. Forward probabilistic reachability has very good properties, that is, all the zones generated are convex and the generated probability edges of MDPs are directly inherited from the syntactic models of PTAs.

The algorithm in Figure 8.2 is an MTBDD-based implementation of the algorithm in Figure 5.2 with respect to our encoding. The algorithm $ModelCheckingPTA$ accepts three parameters: the model of the probabilistic timed automaton PS_{PTA}, which is an MTBDD-encoded representation of the syntax of the original probabilistic timed automa-

ton, the initial set of states ϕ_{init} and the set of target states ϕ_{target}. Lines 1-4 deal with the initialisation: line 1 initialises the generated set of probabilistic transitions with the empty set, and lines 2-3 assign the initial set to both the front set and the reachable set. Lines 5-21 generate the finite-state graph, the edges of which are obtained in lines 8-11 by iterating timed and discrete successor operations. Each generated edge has the form of a tuple $(n, l, l', \zeta, \zeta', p)$, where n is the encoding of non-deterministic choice, (l, ζ) corresponds to current symbolic state, (l', ζ') is the next symbolic state in the generated transition, and p is the probability value. Line 6 constructs a temporary MTBDD with the information necessary for the timed and discrete successor operations by restricting to the front set: each path (Line 8) of the temporary MTBDD has the form of a tuple $(n, l, l', \zeta, g, X, p)$, where l and l' are the current and next nodes, ζ is the current zone associated with current node, g is the guard, X is the set of clocks to be reset and p is the probability. Lines 9.1-9.3 give the MTBDD-based pseudo-code of the construction of a single probabilistic edge of the generated MDP. Line 9.1 obtains the next zone by using standard zone successor operation. Line 9.2 uses the technique in [BY04] to obtain the unique normal form of the next zone and adds it to the list of zones if it is a new one, and otherwise it returns the unique index to it in the list. Line 9.3 constructs and returns the probabilistic edge of the generated MDP. Lines 12-19 extract the reachable states from the generated probabilistic transition set and check whether the fixed point is reached. Line 20 adds the set of newly generated edges to the old one. Finally, in line 22, model checking is performed on the resulting finite-state probabilistic system to obtain the maximum probability of reaching the set of target nodes.

8.1.3 Symbolic Model Checking of the Generated MDP

Having presented techniques to encode the syntactic models of probabilistic timed automata and to generate the MDP via the forward algorithm with MTBDDs in an efficient manner, we now consider the problem of model checking PTCTL reachability formulae against PTAs using the same data structure, which gives a upper bound on maximum probability.

As we saw in Chapters 6 and 7, the problem of model checking PTCTL formulae for

$ModelCheckingPTA(\mathsf{PS}_{PTA}, \phi_{init}, \phi_{target})$	
1.	$\mathsf{PS}_{MDP} := \emptyset$
2.	$\phi_{frontset} := \phi_{init}$
3.	$\phi_{reach} := \phi_{init}$
4.	$done :=$ **false**
5.	**while** $(done =$ **false**$)$
6.	$\quad \mathsf{TmpPS} := \phi_{frontset} \times \mathsf{PS}_{PTA}$
7.	$\quad \mathsf{Edges} := \emptyset$
8.	$\quad$ **for each non-zero path** $(n, l, l', \zeta, g, X, p)$ *of* TmpPS
9.1.	$\quad\quad \zeta' = ZoneSuccessor(l, \zeta, l', g, X)$
9.2.	$\quad\quad \zeta' = \textsc{AddZone}(\textsc{Normalise}(\zeta', k))$
9.3.	$\quad\quad tr = n \times l \times \zeta \times l' \times \zeta' \times p$
10.	$\quad\quad \mathsf{Edges} := \mathsf{Edges} + \mathsf{tr}$
11.	$\quad$ **endfor**
12.	$\quad \mathsf{T}_{01} := \textsc{Threshold}(\mathsf{T}, >, 0)$
13.	$\quad \phi_{tmp} := \textsc{ThereExists}(\underline{\mathsf{z}}, \mathsf{T}_{01})$
14.	$\quad \phi_{tmp} := \textsc{ThereExists}(\underline{\mathsf{x}}, \phi_{tmp})$
15.	$\quad \phi_{tmp} := \textsc{ReplaceVars}(\phi_{tmp}, \underline{\mathsf{y}}, \underline{\mathsf{x}})$
16.	$\quad \phi_{reach'} := \phi_{reach} \vee \phi_{tmp}$
17.	$\quad$ **if** $(\phi_{reach} = \phi'_{reach})$ **then** $done :=$ **true**
18.	$\quad \phi_{frontset} := \phi_{reach'} \setminus \phi_{reach}$
19.	$\quad \phi_{reach} := \phi_{reach'}$
20.	$\quad \mathsf{PS}_{MDP} := \mathsf{PS}_{MDP} + \mathsf{Edges}$
21.	**endwhile**
22.	**return** $MaxProbReach(\phi_{init}, \phi_{target}, PS_{MDP})$

Figure 8.2: The MTBDD version of the forward probabilistic reachability algorithm

PTAs is reduced to model checking a corresponding PCTL formula, which is translated from the PTCTL formula on the MDP generated via the forward or backward algorithm.

In Chapters 6 and 7, we have used PRISM to carry out model checking the PCTL reachability formulae against MDPs. However, there the technique is achieved by first translating the generated MDP and PCTL reachability formulae into the PRISM input language, and then calling PRISM via the command interface to perform the remaining task. In this section we consider reusing the existing algorithms based on MTBDDs. The MTBDD-based model checking algorithms for PCTL verification for MDPs have been implemented in PRISM. The model checking algorithms take a formula in the logic PCTL and a model which is an MDP, and return the set of states which satisfy the formula.

The requirement of invoking these model checking algorithms is that the models can be represented as MTBDDs and the parameters of formulae (ie, sets of states) can be represented as BDDs. We have seen in Section 8.1.1 how the generated MDP model can be represented as an MTBDD by encoding its state space with Boolean variables. What has not been discussed so far is how to represent PCTL formulae as required by PRISM software.

For the reachability properties supported by the forward algorithm, the corresponding translated until operator in PCTL is $\mathcal{P}_{\bowtie p}[\diamond\phi]$, over MDPs, where ϕ is the target set.

The MTBDD function $MaxProbReach$ (in Figure 8.2), which is implemented in PRISM for the numerical computation for the PCTL until operator, takes as input an MTBDD $\mathsf{PS}_{\mathsf{MDP}}$ representing the transition probability matrix of the MDP, two BDDs, phi_{init} and phi_{target}, representing the sets of states $Sat(\phi_1)$ and $Sat(\phi_2)$, respectively, and returns an MTBDD representing the vector of probabilities for each state, assuming that $\textsc{Sat}(\phi)$ is the function which computes the BDD for the set of states satisfying the formula ϕ.

Thus, the problem of representing PCTL formulae is reduced to how to represent two subformulaes as BDDs, which is already done as the subformulae representing the sets of states can be represented as an BDD (special case of MTBDDs) with Boolean variables.

8.1.4 Experimental Results

Table 8.1: Time consumption of the full model $Impl^p$ with wire delay set to 360 ns

Deadline	States	Time(Explicit)			Time(Symbolic)			
					MTBDD/DDD		MTBDD/DBM	
		Forward	Construct.	M.C.	S.F.C.	M.C.	S.F.C.	M.C.
2000	951	5.583	15.020	0.050	27.031	0.125	1.101	0.065
2500	1415	8.671	37.799	0.067	57.391	0.136	1.723	0.144
3000	1425	8.768	38.125	0.067	57.830	0.137	1.708	0.146
3500	2092	12.960	95.042	0.106	123.811	0.320	2.681	0.242
4000	2803	17.480	170.284	0.140	220.011	0.291	3.734	0.282
4500	2799	18.036	168.851	0.147	218.481	0.325	3.743	0.322
5000	3725	24.155	301.363	0.175	386.091	0.418	6.700	0.517
5500	4432	28.989	475.884	0.228	543.811	0.614	6.700	0.517
6000	4675	31.052	528.828	0.255	607.936	0.579	7.177	0.568
7000	6545	45.309	1044.781	0.347	1180.776	0.714	11.317	0.743
8000	8437	60.079	1963.446	0.447	1974.860	0.883	16.621	1.016
9000	9879	72.999	2687.657	0.558	2701.027	1.036	20.930	1.043
10000	11988	91.097	3987.859	0.709	-	-	32.060	1.545
20000	44335	490.927	75078.248	4.181	-	-	238.890	5.800
30000	96592	1551.899	*	-	-	-	922.844	13.436
40000	168514	3820.440	*	-	-	-	2545.692	24.036

The MTBDD-based forward reachability model checking algorithm has been implemented in our software tool using the Java programming language. We have applied this tool to some case studies. In this section, we present some of the results and assess the performance of our symbolic implementation and compare it to the explicit implementation. One property of our symbolic approach is that it is generic, in that it can support different data structures for timing information. Our implementation can support two kinds of representation: DBMs and DDDs. We also present results using these two dif-

Table 8.2: Time consumption of the full CSMA/CD model (max, backoff=1)

Deadline	States	Time(Explicit)			Time(Symbolic)			
					MTBDD/DDD		MTBDD/DBM	
		Forward	Construct.	M.C.	S.F.C.	M.C.	S.F.C.	M.C.
1000	6404	24.145	943.441	0	475.420	0.003	9.115	0.003
1200	9034	39.782	2094.178	0	948.097	0.003	14.976	0.003
1400	11771	60.644	3581.585	0	1613.910	0.003	22.082	0.003
1600	15329	92.919	6289.276	0	2751.622	0.003	32.620	0.004
1800	19453	137.779	10947.995	0.828	4451.248	1.245	47.311	1.264
2000	23468	188.489	15936.895	1.665	6589.583	2.975	69.739	2.831
2200	28516	263.348	23828.224	3.304	9666.598	5.550	94.787	5.777
2400	34023	353.003	-	-	13746.622	9.630	125.467	9.689
2600	39970	467.441	-	-	19074.778	14.625	163.256	14.504
2800	45654	591.693	-	-	24926.546	20.359	201.990	20.322
3000	52561	756.939	-	-	33002.538	27.906	256.135	28.262

ferent representations of the timing information for a comparison between DBMs and DDDs.

We begin with a comparison between explicit and symbolic implementation. In Table 8.1, Table 8.2, Table 8.3 and Table 8.4, we present results for two case studies: the IEEE 1394 FireWire root contention protocol and the IEEE 802.3 CSMA/CD (Carrier Sense, Multiple Access with Collision Detection) protocol.

The results obtained from verifying the full model of the FireWire root contention protocol are shown in Table 8.1 and Table 8.3, which show the time and memory consumption respectively. The property verified is the minimum probability that, from the initial state, a leader (root) is chosen before the deadline is reached.

Table 8.2 and Table 8.4 include results for the time and memory consumption, respectively, of the CSMA/CD protocol when computing the maximum probability of both stations correctly delivering their packets by the deadline D.

In the tables, the term “Explicit” refers to the explicit implementation and “MTBDD-

Table 8.3: Memory consumption of the full model $Impl^p$ with wire delay set to 360 ns

Deadline	MDP Nodes (Explicit)	MDP Nodes (Symbolic)	Mem. (Zone)		
			DDD		DBM
			Peak	Estimated	
2000	1719	3206	46343.66	324.46	143.75
2500	2556	4280	93414.54	492.84	207.75
3000	2585	4401	94993.58	502.61	210.25
3500	3384	5630	184895.18	728.27	299.50
4000	4610	6570	318451.16	977.65	398.75
4500	4476	6430	322968.19	991.48	397.75
5000	5291	7402	546041.32	1311.11	525.25
5500	6003	8857	800000^+	1559.77	623.50
6000	6404	8919	800000^+	1653.37	657.75
7000	8449	11163	800000^+	2317.22	916.25
8000	9818	12543	800000^+	2983.175	1179.25
9000	11856	14511	800000^+	3519.195	1381.75
10000	13126	16211	-	-	1673.50
20000	31526	36069	-	-	6201.75
30000	-	57990	-	-	13534.50
40000	-	78684	-	-	23638.00

/DDD" refers to the implementation which uses MTBDDs for encoding the discrete part and DDDs for timing information; and "MTBDD/DBM" refers to the implementation which differs from the "MTBDD/DDD" by using DBMs instead of DDDs. The data and sub-columns under column "Time(Explicit)" are copies of those in Chapter 6. The left-most column of these tables gives the different deadlines. The second column shows the number of symbolic states generated via the forward construction. The column "S.F.C." refers to the symbolic forward and construction. The column "M.C." refers to the computation time for model checking the given properties against the MDP model encoded as an MTBDD. The unit for all columns under "Nodes" is the number of the nodes in the

Table 8.4: Memory consumption of the full model CSMA (max, backoff=1)

Deadline	MDP Nodes (Explicit)	MDP Nodes (Symbolic)	Mem. (Zone)		
			DDD		DBM
			Peak	Estimated	
1000	7457	10805	577832.75	1099.57	399.71
1200	9653	13681	800000$^+$	1558.76	564.45
1400	11388	16122	800000$^+$	2039.87	737.11
1600	14110	19446	800000$^+$	2667.49	962.01
1800	16697	23077	800000$^+$	3397.76	1223.73
2000	19276	26110	800000$^+$	4108.15	1478.71
2200	22260	30364	800000$^+$	5005.77	1800.20
2400	-	34330	-	-	2151.37
2600	-	38264	-	-	2530.37
2800	-	41497	-	-	2894.82
3000	-	45743	-	-	3337.11

MTBDD where each node occupies 20 bytes. For memory used for zones, we give them in terms of those for DBM and DDD. The memory consumption for DBMs is that actually used. For DDDs, we cannot give the actual memory consumption, and instead give both the estimated and peak time memory consumption. The column "Estimated" refers to an estimation of the memory consumption of all zones based on DDDs when algorithm terminates. The column "Peak" refers to the highest value of memory consumption by DDDs when algorithm terminates. All units for time and memory are seconds and kilobytes.

Here we recall that in order to model check certain properties, the explicit implementation involves two steps: first, it generates the reachable states, and next it represents those as a model in the PRISM input language, which is then passed to PRISM to finish the verification process. In the case of explicit implementation, we give three kinds of measures of time consumption: the time spent on forward computation of the whole set of reachable states, the time for construction of the model as MDP in PRISM, and the time spent on model checking, as well as two kinds of memory usage for discrete and

continuous components. On the other hand, for the symbolic implementation, which is labeled "MTBDD/DDD" and "MTBDD/DBM", as they have already been partly integrated with PRISM, the overall checking process does not go through the PRISM input language: the tool constructs the target MDP models in MTBDDs and directly calls functions provided by PRISM. Thus, for example, for the symbolic implementation, we give two kinds of time consumption measures: the time spent on symbolic construction of the MDP model, which corresponds to the sum of those two times spent in the case of the explicit one (forward computation and construction), and the time spent on model checking. For memory usage it is same.

From the tables, we see that the times for model checking are nearly the same for the three kinds of implementation.

Compared to the explicit implementation, the symbolic implementation based on DBMs has a significant advantage: the time spent on generating the MDP models is no longer a problem, since it took 238 seconds to perform both the forward construction and to generate the MDP in MTBDDs for the full model $Impl^p$ with deadline 20000 ns, whilst it took 75708 seconds to generate the MDP model alone for the same deadline with the explicit version.

The DDD-based symbolic implementation performs worse than the explicit one. It is slower due to a large number of intermediate DDD nodes being generated, which forces the DDD run-time library to invoke garbage collection. The main reason behind this is that DDDs have no canonicity property. The canonicity property of zones is critical to the model checking, especially for model checking PTAs, because each time a state is found the algorithm is forced to check whether it has already been processed before. This is one of the main operations that the algorithm has to perform and many such equality checks to see if two states are equal are performed. BDDs, which have a canonical form under a given variable ordering, allow equality checking which is just a simple comparison of two unique ids of BDDs and there are no temporary nodes which are generated. Since DDDs have no canonical form, the equality checking on DDDs is realised through checking whether one DDD is a subset of another in both directions. Furthermore, the operation of checking membership is based on recursive calls on the nodes along the DDDs. Like

BDD operations, the recursive calls generate some temporary nodes and these temporary nodes are put in the cache in case other operations need them in the future. Memory for the generated temporary nodes is only released periodically via garbage collection when there is no more space for new nodes.

What we can see from the experimental results is that the DBM-based symbolic implementation performs very well. There is no surprise why DBM-based symbolic implementation does so well in the case of the forward approach. Here we are more interested in why the DDD implementation is outperformed by DBMs. There are two factors which lead DBMs to perform well in comparison with DDDs. The first is that DBMs have a canonical form. Although, in the worst case, the complexity for comparing whether two DBMs are equal is $(n+1)^2$ where n is the number of the clocks, in practice it could be speeded up whenever two elements in the same position of matrices are different. The second is that, when using DBMs, operations are merely performed on two-dimensional matrices, the memory is fixed and there is no ordering on them.

As mentioned in Chapter 6, there are other papers concerning the forward implementation, [DKN02, KNS03a, DKN04]. Among these, only [KNS03a, DKN04] consider the full model of FireWire, and only [DKN04] gives the information such as time spent on the construction of the product model. The results from [KNS03a] for the full model of Firewire is based on digital clocks. The other source of experimental results can be found on the PRISM website [PRI], where the time spent on the construction of the product model of FireWire is given for the method used in [KNS03a] and compared with the results obtained based on digital clocks. In [DKN04], two major reduction techniques are adopted in order to reduce the size of generated MDP. One is based on the instance encoding [DKN02, DKN04] and the other is based on bi-simulation. The data used here for comparison are based on our method using DBMs and the better of [DKN04] and [PRI]. For example, the instance encoding method has two variants. The performance of the instance encoding method "instance with relative and absolute compaction" is the better variant. Below we use "compaction" as an abbreviation for "instance with relative and absolute compaction".

We will focus on comparison of our symbolic implementation with results obtained

based on the digital clocks [KNS03a], results based on different reduction techniques adopted in [DKN04], compaction and bi-simulation. In general, our symbolic method performs better than both the digital clocks and compaction methods. Although our symbolic approach and the approach based on digital clocks are MTBDD-based, the approaches compared are different. Thus, the criteria we used are the total time spent on the verification and memory consumption for the MTBDD representation of the generated reachable graph. We observe that our symbolic approach is better than the approach based on digital clocks in terms of both the time consumption and the memory consumption. For example, for deadline 10000, our method takes about 32 seconds to perform the verification, which is 600 times faster than the method of digital clocks which takes about 18944 seconds to finish, and our method generated 16211 nodes which is 80 times fewer than 1244511 for digital clocks. The increase in value of the deadline parameters will result in an increase in both the time consumption and memory consumption for both approaches. However, the digital clocks approach is more affected by this increase. For example, an increase from the deadline 8000 to 10000 in case of the full model, results in an increase of 16 seconds in the time spent on verification and an increase of 3500 in the number of generated states using our method, while an increase of 10000 seconds in the time and an increase of 43453600 in the number of states. This increase in both time and the number of states for method of digital clocks is explained as follows. In the digital clocks' method, the size of the model is determined by the range of the variables, which are used to encode the state of the model. The bigger value of the variable for the deadline implies the greater number of states, which hence consumes more verification time. In contrast to the digital clocks approach, the change of the deadline parameter will be reflected in the computation of DBMs, but not directly in the size of the generated reachable graph. Compared with the compaction method, for deadline 40000, our method takes about 2570 seconds to perform the verification, which is 10.8 times faster than the compaction method which takes about 27890 seconds to finish. The reason why the compaction method is not as good is that, although the compaction method can reduce the size, the main obstacle to verifying the full model against large deadlines is the time required by PRISM to build the model. The exception is the bi-simulation

method which performs better than ours; for the deadline of 40000 in FireWire, the bi-simulation method takes about 1309 seconds to perform the verification against reduced state space of about 10000 states, which is round one time faster than our method which takes about 2570 seconds against state space of 168514 states. The reason behind why bi-simulation method performs better is that it can reduce greatly the size of the generated reachable graph, for deadline 40000 in FireWire, the size of the reduced state space is less than one sixteenth of the size of the state space generated by our method. However, the bi-simulation method employs two other tools, KRONOS and CADP [GH02], and the approach involves the following steps as follows: modify the output from KRONOS by identifying the states where a leader has been elected after the deadline has passed, represent the reachability graph in a format suitable for the CADP tool set, use CADP to construct the bi-simulation quotient, and finally use a translator to obtain the PRISM language format from the CADP output.

8.1.5 Discussion

Variable Ordering

The MTBDD variable ordering has been discussed in [Par02]. In our symbolic implementation, we exploited the result of [Par02] concerning the heuristics for determining the BDD variable ordering. However, we are more interested in the DDD variable ordering in the forward algorithm. The behaviour of different DDD variable ordering in the backward algorithm has been described in the Chapter 7, where the zones could be non-convex. Figure 8.3 and Figure 8.4 show that different variable ordering for DDDs with deadline 1000 and 1200, respectively. The x-axis is for different variable orderings on clocks. There are a total of 120 (5!) different ordering since the model of CSMA has only five clocks. The y-axis is for the time spent on the forward exploration. From the figures, we can see that the forward algorithm is less sensitive to the DDD variable orderings and different parameters show a similar pattern in terms of the time spent on calculation. The possible explanation for this is that unlike, in the backward algorithm, the zones encountered in the forward algorithm are convex. Although the different parameters for deadline show a similar pattern, increasing the value of the parameter for the deadline will typically result

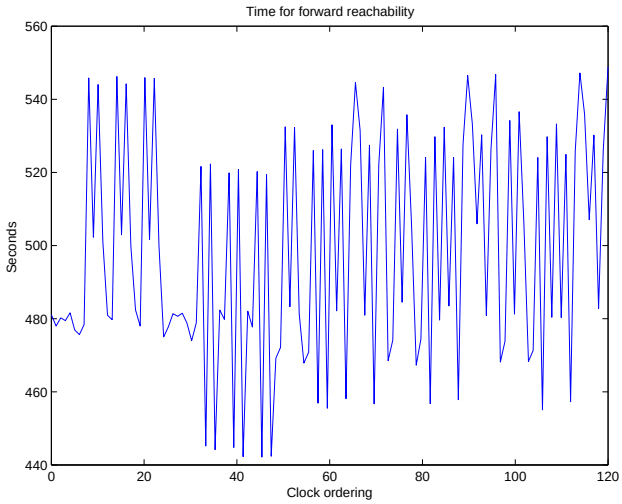

Figure 8.3: DDD ordering for CSMA with deadline 1000

in an increase in the verification time due to more states which are generated.

Estimation for the Upper Bound for Size of Zones

Below we summarise the main issues that have to be addressed when applying our encoding method:

- Unlike in the case of non-probabilistic timed systems, in which the on-the-fly technique [BTY97a] could be applied to make search algorithms finish as early as possible, the forward probabilistic reachability search has to construct the whole reachable zone graph in order to obtain the probability value.

- The size of the state space and number of transitions between these states of the generated probabilistic system is uncertain before the algorithms terminate. Although, if using the region graph, the size of the state space of a PTA can be established in advance of the model construction, such an approach is impractical due to the region graph being exponential in the number of clocks and the maximal constant appearing in the model.

As a result, we cannot fix the size of the vector of Boolean variables needed to encode the zone part in advance. Instead, we pre-allocate the vector based on an estimate. One

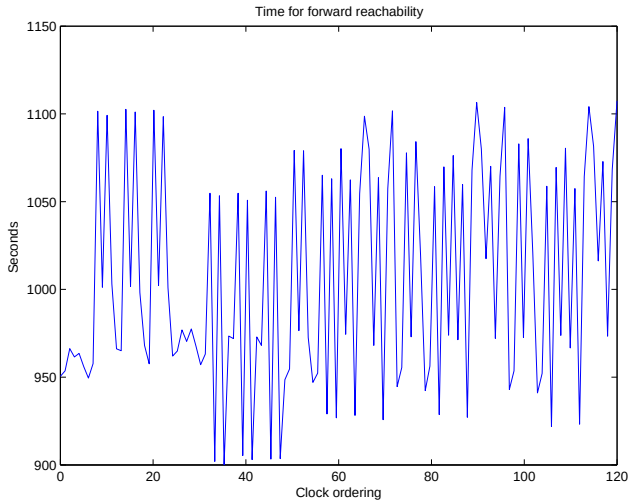

Figure 8.4: DDD ordering for CSMA with deadline 1200

factor that will affect the size of zones is the parameters for the model, such as the largest constant appearing in the model. An estimation of the size of the zones could be explained as follows. We pick a value p and s for the parameter of the largest constant and upper bound of zone size. If the test is successful, then we obtain the real value for the zone size. Next we fix a step value d and run a few tests for different parameters of the largest constant starting from value p, and find the corresponding value for real zone size. Based on the pattern of values (p, d, s), a function can be formulated and used for predicting the upper bound for the zone size.

Zone Structure

In Section 8.1.1, it is shown that the number of zones increases as the value of the deadline increases. A simple method is adopted for allocating indices to the encountered zones. The range of the values for the indices is determined by the total number of zones to be obtained via the forward algorithm. Such a simple method does not exploit the potential implicit structural information among states. Furthermore, this could destroy the regularity of the MTBDDs representing the state space. The successful application of MTBDDs depends on its compact representation or sharing mechanism. However, in the

case of simple allocation of indices, the continuous component becomes the dominating factor which overwhelms the discrete component because there is less sharing among the discrete components as the range for the indices becomes larger. As shown in [DKN02], there is some sharing among the discrete component which could be exploited to decrease the number of lines of transitions in the input language of PRISM. The basic idea of [DKN02] is to figure out the maximum number n of distinct zones for each node and then relabel each zone by assigning a value in the range of $[0, n-1]$. In other words, the form of the state pair becomes (l, n) where l is the discrete component and n is the new label for the corresponding zone associated with the discrete component. In this section, we discuss how this idea can be adapted for our symbolic implementation of the forward algorithm based on MTBDDs. We first present, informally, how the adaptation could be implemented. A direct way to achieve this is to apply the relabelling after the forward search is finished, which is done in [DKN02]. However, this means that the algorithm will traverse the whole MTBDDs tree representing the generated MDP. The aim of the adaptation is that it must be efficient. The traversal could be avoided as follows. We can make the adaptation more efficient by applying the relabelling at the same time as a new zone is found for the same node instead of through the traversal of the generated MTBDDs. This could be realised with the help of additional memory for storing the intermediate information. The requirement for this intermediate information is that it is minimal in that it should hold sufficient information so that the forward search could proceed without losing any information. The new form of a state pair contains only labels for the zones instead of the zones themselves. We need a way to build the link between the label and the corresponding zone through nodes, thus the minimal information in the intermediate storage consists of the nodes, the associated zones and the labels.

The original algorithm could be modified as follows. When a zone is found, it is compared against the set of zones associated with the same discrete component: if the zone is not previously associated with it, a label value is assigned to the zone and the zone is added to the zone set associated with the same discrete component; otherwise the zones set remains unchanged. In both cases the pair of node and label is used as the representation of symbolic states.

Formally, the transitions in the generated MDP $f(n, l, lp, old, oldp) = p$ are replaced with $f(n, l, lp, new, newp) = p$, where f is the function encoding the transition, n is nondeterministic choice, l and lp denotes the current and next nodes, old and $oldp$ is current and next zones, and new and $newp$ is the current and next label corresponding the zones associated with nodes. The intermediate storage can be represented as a function $f(l, new) = old$, where l is a node, new is the label and old is the corresponding zone associated with l. Using the same Boolean encoding within the BDD, the function $f(l, new) = old$ could be encoded as an MTBDD.

Figure 8.5 gives the new MTBDDs algorithm. Compared with the original MTBDD-based algorithm in Figure 8.2, the main differences in Figure 8.5 are: an intermediate storage in MTBDDs is introduced in line 0, and function Label performs the function of relabelling which assigns a value to the zone associated with the corresponding discrete component.

The results obtained from verifying the full models of CSMA/CD protocol and FireWire based on the new implementation are shown in Table 8.5, 8.6, 8.7 and 8.8.

For the purpose of reducing both the memory usage for the generated MDP and the time spent on model checking, the results are encouraging. However, this is not the case if we consider the total consumption of both memory and time. From the tables, we can see that, compared with the original implementation, the new implementation is less efficient because it uses more memory and time in total. The difference between these two implementation is that, although less time and memory is used for model checking the generated MDPs, more time and memory is spent on the construction of the intermediate data. This is for the same reason as before: the zones are still the dominating factor which destroys the regularity of the MTBDDs for the representation of the function of $function f(l, new) = old$, which is the mapping between the labels and the zones.

8.2 Kronecker-based Model Construction

In Section 8.1.1, we demonstrated a general encoding method for probabilistic timed automata by using MTBDDs. We also showed the application of our method to model checking for PTAs. In Section 8.1.4, we see that our method based on MTBDDs proved

ModelCheckingPTA($\mathsf{PS}_{PTA}, \phi_{init}, \phi_{target}$)

0. $\mathsf{Intermedia} := \emptyset$
1. $\mathsf{PS}_{MDP} := \emptyset$
2. $\phi_{frontset} := \textsc{Label}(\mathsf{Intermedia}, \phi_{init})$
3. $\phi_{reach} := \textsc{Label}(\mathsf{Intermedia}, \phi_{init})$
4. $done := \textbf{false}$
5. **while** ($done = \textbf{false}$)
6. $\quad \mathsf{TmpPS} := \phi_{frontset} \times \mathsf{PS}_{PTA}$
7. $\quad \mathsf{Edges} := \emptyset$
8. $\quad$ **for each non-zero path** $(n, l, l', \zeta, g, X, p)$ *of* TmpPS
9.1. $\quad\quad \zeta' = ZoneSuccessor(l, l', \zeta, g, X)$
9.2. $\quad\quad \zeta' = \textsc{AddZone}(\textsc{Normalise}(\zeta', k))$
9.3. $\quad\quad \phi_{next} = \textsc{Label}(\mathsf{Intermedia}, l', \zeta')$
9.4. $\quad\quad tr = n \times \textsc{Label}(l, \zeta) \times \phi_{next} \times p$
9.5. $\quad\quad \mathsf{Intermedia} = \mathsf{Intermedia} + \textsc{ReplaceVars}(\phi_{next}, \underline{\mathsf{y}}, \underline{\mathsf{x}})$
10. $\quad\quad \mathsf{Edges} := \mathsf{Edges} + \mathsf{tr}$
11. $\quad$ **endfor**
12. $\quad \mathsf{T}_{01} := \textsc{Threshold}(\mathsf{T}, >, 0)$
13. $\quad \phi_{tmp} := \textsc{ThereExists}(\underline{\mathsf{z}}, \mathsf{T}_{01})$
14. $\quad \phi_{tmp} := \textsc{ThereExists}(\underline{\mathsf{x}}, \phi_{tmp})$
15. $\quad \phi_{tmp} := \textsc{ReplaceVars}(\phi_{tmp}, \underline{\mathsf{y}}, \underline{\mathsf{x}})$
16. $\quad \phi_{reach'} := \phi_{reach} \vee \phi_{tmp}$
17. $\quad$ **if** ($\phi_{reach} = \phi'_{reach}$) **then** $done := \textbf{true}$
18. $\quad \phi_{frontset} := \phi_{reach'} \setminus \phi_{reach}$
19. $\quad \phi_{reach} := \phi_{reach'}$
20. $\quad \mathsf{PS}_{MDP} := \mathsf{PS}_{MDP} + \mathsf{Edges}$
21. **endwhile**
22. **return** $MaxProbReach(\textsc{Label}(\phi_{init}), \textsc{Label}(\phi_{target}), PS_{MDP})$

Figure 8.5: The MTBDD version of the forward probabilistic reachability algorithm adapted according to [DKN02]

Deadline	PTA nodes	Nodes		
		Kronos-Adaptation		
		None	Yes	
		MDP	temporary	MDP
1000	2290	10805	10602	3078
1200	2290	13681	14858	3632
1400	2290	16122	19335	4230
1600	2290	19446	25171	5008
1800	2290	23077	31903	5666
2000	3010	26110	38587	6596
2200	3010	30364	46866	7646
2400	3010	34330	55919	8413
2600	3010	38264	65648	9159
2800	3010	41497	75002	10124
3000	3010	45743	86347	10961

Table 8.5: Memory comparison for verifying CSMA with/without Kronos adaptation

Deadline	PTA nodes	Nodes		
		Kronos-Adaptation		
		None	Yes	
		MDP	temporary	MDP
2000	7893	3206	1490	2103
2500	7893	4280	2019	2439
3000	7893	4401	2041	2490
3500	8463	5630	2828	3062
4000	8463	6570	3637	3244
4500	8463	6430	3630	3238
5000	9033	7402	4751	3872
5500	9033	8857	5558	4383
6000	9033	8919	5807	4343
7000	9033	11163	7986	5510
8000	9603	12543	10152	6010
9000	9603	14511	11838	6876
10000	14163	16211	14535	8186
20000	9603	36069	51908	17224
30000	9603	57990	112319	27801
40000	9603	78684	195503	39076

Table 8.6: Memory comparison for verifying FireWire with/without Kronos adaptation

Deadline	Time			
	Kronos-Adaptation			
	None		Yes	
	F.W.	MC	F.W.	MC
1000	9.115	0.003	17.837	0.001
1200	14.976	0.003	31.357	0.002
1400	22.082	0.003	49.341	0.002
1600	32.620	0.004	78.624	0.002
1800	47.311	1.264	123.142	0.328
2000	69.739	2.831	184.135	0.714
2200	94.787	5.777	269.775	1.484
2400	125.467	9.689	383.128	2.680
2600	163.256	14.504	541.738	3.848
2800	201.990	20.322	719.885	5.304
3000	256.135	28.262	954.831	7.268

Table 8.7: Time comparison for verifying CSMA with/without Kronos adaptation

Deadline	Time			
	Kronos-Adaptation			
	None		Yes	
	F.W.	MC	F.W.	MC
2000	1.101	0.065	1.382	0.035
2500	1.723	0.144	2.048	0.082
3000	1.708	0.146	2.058	0.090
3500	2.681	0.242	3.304	0.135
4000	3.734	0.282	4.702	0.121
4500	3.743	0.322	4.601	0.235
5000	5.374	0.391	6.771	0.169
5500	6.700	0.517	8.397	0.348
6000	7.177	0.568	9.032	0.215
7000	11.317	0.743	14.212	0.297
8000	16.621	1.016	21.040	0.362
9000	20.930	1.043	26.342	0.423
10000	32.060	1.545	39.673	0.562
20000	238.890	5.800	303.099	2.249
30000	922.844	13.436	1210.245	5.199
40000	2545.692	24.036	3494.244	10.652

Table 8.8: Time comparison for verifying FireWire with/without Kronos adaptation

to behave extremely well in terms of the computational complexity compared to the explicit method. Furthermore, our experimental results of real case studies could also be performed very efficiently. When we turned our attention to analysis of artificial cases, for example, in the case of the Milner's scheduler [Mil89] which we are using to test the scalability of our method, we witnessed that if the size of the systems to be analysed is small or medium, the model construction process performs well, but the performance decreased quickly when the size of the systems grows large. As shown in our paper [WK05], the explicit method for constructing the product automata is inefficient because the time to build it increases as the number of nodes increases.

The main cause of this problem is easy to identify and obvious: the method we adopted is explicit, that is, we explicitly enumerated the nodes to build the product model of the system which consists of more than one module. The explicit method could be described informally as follows. Basically, it involves two steps: the first step is based on a kind of Breadth-First Search of the PTA models, while the second one is just a simple application of the encoding method. We started with the set of initial nodes, enumerating all the possible combined transitions (synchronised or asynchronous) and calculating all of the next nodes which could be reached via a single transition. Then we enumerated each of the newly found reachable nodes in the previous iterations to find out all the transitions and their corresponding reachable nodes. The iterated searching continued until there was no new reachable node found. Finally, we applied our encoding method to the search result of the set of reachable nodes and transitions among them.

In real life, applications of model checking will always lead to models which contain huge numbers of states. The inefficiency of explicit methods to construct product models is well-known because it is often infeasible to enumerate each state and transitions between states. The Kronecker representation, which was originally introduced in [Pla85], has been shown to be successful when applied to the model construction of stochastic systems. The power behind the Kronecker approach is that it is only necessary to store the small, component matrices and the structure of the expression which combines them. However, there is very little work concerning the Kronecker-based technique for the model construction of systems involving real-time clocks.

Although the Kronecker representation has been applied in [BCDK00, KL97, MCD00, PRI], there is little work for systems involving real-time clocks, for example, the real-time systems modeled as timed automata. Although the Kronecker-based technique has been applied to Continuous-Time Markov Chains (CTMCs) [BKH99b], there is no explicit timing information in those models, for example, real-time clocks and the corresponding clock constraints appearing in the models.

In this section, we consider a Kronecker-based efficient representation for the PTA model. Our method to build the PTA model is based on that implemented in the tool PRISM, which employs the Kronecker representation. The outline of this section is as follows. Firstly, we describe how to exploit Kronecker-based model construction that has been implemented in PRISM for our purpose. Secondly, results are presented and compared with those methods based on explicit model construction.

8.2.1 Synthesis with the Kronecker-based Model Construction

We first recall the PRISM input language. Then we describe our type-based PRISM language for the model description of probabilistic timed automata. Last we describe the algorithm, which is synthesised with our symbolic forward algorithm, for the construction of the typed variables from the MTBDDs model representation.

PRISM Language In the input language of PRISM [PRI], there are two fundamental elements in the model description language: modules and variables. The system which is considered here is a composition of n modules $M_1, \ldots M_n$. Each module M_i contains a set of variables Var_i; these variables are called local. Each local variable in module M_i, $x \in Var_i$ has its own finite range of values $range(x)$. A transition from one state to another in a module can be made by changing the value of its local variables. A global transition of the whole model from one global state to another comprises transitions from one or more of its component modules. Two kinds of global transitions are defined. One is the asynchronous transition, in which case a single module makes a transition independently, while the other modules remain unchanged in their current state; the other is the synchronous transition, which is two or more modules making a transition

simultaneously. The behaviour of each module is defined through a set of commands which gives the semantics of the transition. The global behaviour of the whole model is given by the parallel composition of its component modules.

Typed-based Model Description Language Although the encoding method for PTAs has already been introduced in [WK05], it is not applied to the process of the model construction. In this part, we describe the model description language which is based on the PRISM input language for the model type of Markov decision process, on which the semantics of PTAs is built.

In the PRISM language, the variables are just used to encode the state of the model. The PRISM input language at present cannot support real-time information encountered in the transitions of PTAs, such as real-time clocks and real-time clock constraints. However, we need a way to encode the timing information: the invariants associated with nodes, clock constraints and clock resets encountered in the transitions of probabilistic timed automata. The solution is, firstly, to use Boolean variables to encode the index to each of them and use a separate description for the set of clocks, clock constraints and clock resettings, and, secondly, to introduce four types for variables and require that all variables are typed. The four types of variables are nodes, guards, resets and misc. All variables in a system belong to one of the four types in our language. The name of a variable has a prefix of "loc", "guard", "reset" and "misc" for the type nodes, guards, resets and misc, respectively. We require the set of variables in each group to be disjoint. The variables of the type nodes are only used for encoding the node part of the discrete component appearing in the PTAs. The variables of the type misc are only used for encoding the remaining information of the discrete component. The variables of the type guards and resets are used for encoding the enabling condition (clock constraints) and clock resets in the transitions of PTA, respectively. We describe our language using the PTA example in Figure 5.1. Figure 8.6 and Figure 8.7 give its corresponding model description in our language and the corresponding description language for the encoding for the timing information.

First, we describe the language introduced for timing information. The description in Figure 8.7 starts with the list of all clocks appearing in the model and the property

or specification. This is followed by a list of timing constraints for each module. Inside each module, a list of invariants (set of clock constraints) is declared. Each invariant corresponds to a node in that module, that is, each invariant is assigned a unique integer value which is also assigned to the corresponding node. Next is a list of enabling conditions (set of clock constraints) used in the model. The final component in the module is a list of clock resets. Similarly to the invariants, a unique integer value is assigned to the element of the list of the enabling conditions and clock resets.

Second, we introduce the form of a single command of a system module. Our command bears similarities to those used in the PRISM input language for MDPs, and takes the following form:

$$[]\ g \rightarrow \lambda_1 : u_1 \wedge g_1 \wedge r_1 + \cdots + \lambda_n : u_n \wedge g_n \wedge r_n;$$

However, the interpretation of the command here is quite different from that of for normal PRISM command, where the guard is always put at the left hand side of $\rightarrow$. The reason why we use updated variables (primed version of MTBDDs variables) for both guards and clock resettings is because we need a way to distinguish the possible nondeterministic choices at certain nodes. Each command consists of two parts which are separated by the $\rightarrow$. On the left hand side of the $\rightarrow$ is the guard for untimed information. The guard g is a predicate over all the variables, which include four kinds of variables defined in the whole model, namely the nodes, misc, guards and reset variables. On the right hand side, there are one or more updates which modify the values of some or all of the variables. Each update u_i describes a transition which the module can make. A transition corresponding to updates u_i with probability λ_i can be made if and only if both guard g and g_i are true and the transition sets the set of clocks encoding in the variable r_i, where g_i is the enabling condition which involves the clock constraints.

Consider the second command in Figure 8.6 as an example:

$$[]\ (loc_1 = 1) \rightarrow 0.005 : (loc'_1 = 1) \wedge (guard'_1 = 1) \wedge (reset'_1 = 1) + \\ 0.995 : (loc'_1 = 0) \wedge (guard'_1 = 1) \wedge (reset'_1 = 1);$$

The interpretation of the above command is as follows: once the system is at the

nondeterministic

module M_1
loc_1 : [0..1] **init** 0;
$guard_1$: [0..1] **init** 0;
$reset_1$: [0..1] **init** 0;
[] $(loc_1 = 0) \rightarrow 0.01 : (loc_1' = 0) \wedge (guard_1' = 0) \wedge (reset_1' = 1) +$
$0.99 : (loc_1' = 1) \wedge (guard_1' = 0) \wedge (reset_1' = 1)$;
[] $(loc_1 = 1) \rightarrow 0.005 : (loc_1' = 1) \wedge (guard_1' = 1) \wedge (reset_1' = 1) +$
$0.995 : (loc_1' = 0) \wedge (guard_1' = 1) \wedge (reset_1' = 1)$;
endmodule

Figure 8.6: A textual description of the PTA in Figure 5.1

current node *open* ($loc_1 = 1$), and it can remain at the current node *open* for 2 to 3 time units (guard is $guard_1' = 1$ and invariant is $x < 3$), then it will take a transition either to node *close* ($loc_1 = 0$) with probability 0.995 or remains at node *open* with probability 0.005 and clock x ($reset_1' = 1$) is to be reset upon the transition being taken.

Algorithm for the Construction of the Typed Variables After the introduction of the input description language for probabilistic timed automata, we can exploit the built-in Kronecker mechanism of PRISM for the construction of the product model of PTAs into an MTBDD representation. However, the MTBDD representation of the product model differs from that constructed from the explicit method in that the explicit method uses a single variable for enabling conditions and its primed version for clock resettings, while the Kronecker-based representation has more than one variable for enabling conditions, namely, one variable for each module for representing a sub-condition. These sub-conditions work together as conjuncts in order to form an enabling condition of a transition. Similarly, clock resettings are represented using more than one variable, one for each module.

Take a single module without nondeterministic choices as an example, in this case, the syntactic models of PTAs can still be treated as a square matrix. Formally, assuming

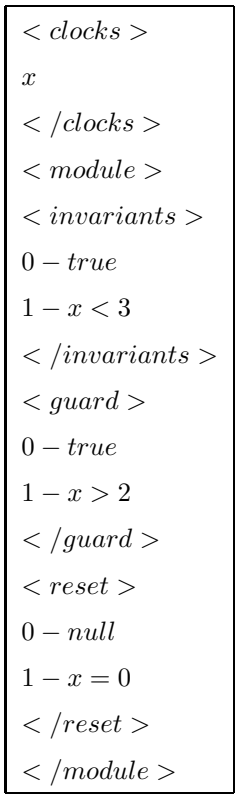

Figure 8.7: The textual description of the timing information for the PTA in Figure 5.1

given an encoding of $2^{(l+g+r)}$ integers into $(l+g+r)$ Boolean variables, i.e. a one-to-one function $enc : \{0,\ldots,2^{(l+g+r)}-1\} \rightarrow \mathbb{B}^n$, we can represent a $2^{(l+g+r)} \times 2^{(l+g+r)}$ square matrix, $\mathbf{M}$, as a mapping from $\{0,\ldots,2^{(l+g+r)}-1\} \times \{0,\ldots,2^{(l+g+r)}-1\}$ to R by an MTBDD M over $2(l+g+r)$ variables, $(l+g+r)$ of which encode row indices and $(l+g+r)$ of which encode column indices. Using *row variables* $\underline{\mathsf{x}} = (\mathsf{x}_1,\ldots,\mathsf{x}_{(l+g+r)})$ and *column variables* $\underline{\mathsf{y}} = (\mathsf{y}_1,\ldots,\mathsf{y}_{(l+g+r)})$, we say that M represents $\mathbf{M}$ if and only if $f_{\mathsf{M}}[\underline{\mathsf{x}} = enc(i), \underline{\mathsf{y}} = enc(j)] = \mathbf{M}(i,j)$ for $0 \leq i,j \leq 2^{(l+g+r)}-1$. It is convenient to divide both row and column vectors $\underline{\mathsf{x}}$ and $\underline{\mathsf{y}}$ into 3 sub-vectors: $\underline{\mathsf{x}} = (\underline{\mathsf{x}}_\mathsf{l}, \underline{\mathsf{x}}_\mathsf{g}, \underline{\mathsf{x}}_\mathsf{r})$ and $\underline{\mathsf{y}} = (\underline{\mathsf{y}}_\mathsf{l}, \underline{\mathsf{y}}_\mathsf{g}, \underline{\mathsf{y}}_\mathsf{r})$. Here we use l Boolean variables to encode the set of 2^l nodes, g Boolean variables to encode the set of 2^g guards (clock constraints), and r Boolean variables to encode the set of 2^r clock resettings. The interpretation of elements of $\mathbf{M}$ $f_{\mathsf{M}}[\underline{\mathsf{x}}_\mathsf{l} = enc(l), \underline{\mathsf{x}}_\mathsf{g} = enc(g), \underline{\mathsf{x}}_\mathsf{r} = enc(r), \underline{\mathsf{y}}_\mathsf{l} = enc(l'), \underline{\mathsf{y}}_\mathsf{g} = enc(g'), \underline{\mathsf{y}}_\mathsf{r} = enc(r')] = \mathbf{M}(i,j)$ is as follows: if the current node is $[\underline{\mathsf{x}}_\mathsf{l} = enc(l)]$ and guard $[\underline{\mathsf{y}}_\mathsf{g} = enc(g')]$ is satisfied, the transition could be performed to go to next node $[\underline{\mathsf{y}}_\mathsf{l} = enc(l')]$ and the clock in the $[\underline{\mathsf{y}}_\mathsf{r} = enc(r')]$ is reset.

In order to exploit our symbolic forward algorithm, we have to construct the typed variables of nodes, enabling condition and clock resetting from the MTBBDs paths. Figure 8.8 gives the algorithm for the construction of the typed variables. The algorithm accepts an MTBDD and recursively traverses the MTBDD path to retrieve the variable value corresponding to the module. Line 2 constructs the module variable. Lines 3-10 fetch the index value for module variable according to different type. Lines 11 and 12 recursively traverse child nodes. Line 14 returns the values of module variables.

8.2.2 Experimental Results

The Kronecker-based model construction and the textual input language for the model of probabilistic timed automata described in the previous section have been implemented in our software tool. This section presents the experimental results of some case studies to which the above techniques have been applied using the tool. The performance of our symbolic implementation of model construction are assessed and compared to that of equivalent and explicit approaches. Since the comparison is only meaningful for the systems containing more than one module, our analysis focuses on this kind of system.

ConstructionOfTypedVariable(v)	
1.	**if** (v is a non-zero terminal vertex)
2.	**then** construct module variable
3.	**if** variable is type of node
4.	**then** fetch index value of sub-node
5.	**elseif** variable is type of guard
6.	**then** fetch index value of sub-guard
7.	**elseif** variable is type of resetting
8.	**then** fetch index value of sub-resetting
9.	**else** fetch index value of sub-misc
10.	**endif**
11.	$ConstructionOfTypedVariable(else(v))$
12.	$ConstructionOfTypedVariable(then(v))$
13.	**endif**
14.	**return** value of nodes, guard, resetting, misc

Figure 8.8: The construction of typed variable

With regard to the performance, we list the results for both time and memory consumption for the generation of the product models.

As before, we present experimental results based on two case studies: the IEEE 1394 FireWire root contention protocol and the IEEE 802.3 CSMA/CD protocol. In addition, we present results for Milner's scheduler [Mil89], see Appendix A.2. The models for FireWire and CSMA/CD are the same as those used in previous chapters. The model for Milner's scheduler is that used in [MLAH99c] with only one clock.

The results obtained from verifying the full models of the FireWire root contention protocol are shown in Table 8.10 and Table 8.12, which are for memory and time consumption respectively. The property verified is the maximum probability that, from the initial state, a leader (root) is chosen before the deadline is reached.

The results obtained from verifying the full models of CSMA/CD protocol are shown in Table 8.9 and Table 8.11, which include memory and time consumption respectively for the CSMA/CD protocol when computing the maximum probability of both stations correctly delivering their packets by the deadline D.

From the tables, we can see that the model construction based on Kronecker performs better than the explicit approach. This is because an explicit enumeration of the nodes is infeasible for larger models. The increase in memory usage for PTAs is caused by two factors: firstly, we do not exploit structure when encoding the guard and reset, which are combined into a single guard or reset; and secondly, both guard and reset encoding depends on the value of the upper bound on the size of the set of zones. The time spent on model construction is almost the same, and memory usage to store the model of a PTA is also the same for different parameters for deadlines. This is because the clock constraints, which are expressed in a separate file for input, are already encoded into integer values which are indifferent to the size of the clock constraints or values used in them. From the experimental results, we can see that when the number of modules in the systems is small the difference between explicit and Kronecker-based approaches is not great with regard to the time consumption of the model construction. However, the situation is rather different when the number of modules, the state space of the model and the transitions between states increase considerably as shown in the artificial example

Deadline	Nodes			
	Explicit		Kronecker	
	PTA	MDP	PTA	MDP
1000	2290	10805	578	11095
1200	2290	13681	578	14142
1400	2290	16122	578	16614
1600	2290	19446	578	20124
1800	2290	23077	578	24013
2000	3010	26110	578	27511
2200	3010	30364	578	31734
2400	3010	34330	578	36244
2600	3010	38264	578	41692
2800	3010	41497	578	45875
3000	3010	45743	578	51408

Table 8.9: Memory comparison for verifying CSMA with/without Kronecker

of Milner's scheduler. For the explicit approach, it could become infeasible to handle the process of model construction, but the Kronecker-based approach could still maintain linear growth in terms of the time consumption on the model construction. The reason for the linear growth based on Kronecker-based model construction can be explained as follows. Firstly, each cycle in Milner's scheduler is very small and contains only two or four nodes, and hence its corresponding MTBDDs representation is also very small. Secondly, our method is based on PRISM which adopt efficient MTBDD-based implementation of Kronecker expression.

8.3 Summary

In summary, we have implemented model checking of probabilistic timed automata via the forward algorithm in a symbolic way, in which all the steps involved in the verification

Deadline	Nodes			
	Explicit		Kronecker	
	PTA	MDP	PTA	MDP
2000	7893	3206	1408	3473
2500	7893	4280	1408	4597
3000	7893	4401	1408	4674
3500	8463	5630	1408	5922
4000	8463	6570	1408	6758
4500	8463	6430	1408	6701
5000	9033	7402	1408	7829
5500	9033	8857	1408	9324
6000	9033	8919	1408	9348
7000	9033	11163	1408	11672
8000	9603	12543	1408	13014
9000	9603	14511	1408	15016
10000	14163	16211	1408	16895
20000	14163	36069	1408	36953
30000	14163	57990	1408	59272
40000	14163	78684	1408	81307
50000	-	-	1408	103179

Table 8.10: Memory comparison for verifying FireWire with/without Kronecker

Deadline	Time					
	Explicit			Kronecker		
	Constr.	F.W.	MC	Constr.	F.W.	MC
1000	0.123	9.115	0.003	0.043	9.177	0.002
1200	0.123	14.976	0.003	0.036	14.446	0.002
1400	0.115	22.082	0.003	0.035	20.735	0.003
1600	0.121	32.620	0.004	0.035	30.040	0.004
1800	0.119	47.311	1.264	0.035	44.283	1.418
2000	0.120	69.739	2.831	0.035	63.745	2.970
2200	0.124	94.787	5.777	0.035	88.059	5.681
2400	0.128	125.467	9.689	0.036	117.772	9.494
2600	0.121	163.256	14.504	0.035	155.722	14.818
2800	0.126	201.990	20.322	0.036	194.190	20.531
3000	0.121	256.135	28.262	0.035	245.192	27.662

Table 8.11: Time comparison for verifying CSMA with/without Kronecker

Deadline	Time					
	Explicit			Kronecker		
	Constr.	F.W.	MC	Constr.	F.W.	MC
2000	0.351	1.101	0.065	0.087	1.799	0.069
2500	0.335	1.723	0.144	0.088	2.578	0.180
3000	0.364	1.708	0.146	0.087	2.615	0.181
3500	0.337	2.681	0.242	0.087	3.995	0.253
4000	0.342	3.734	0.282	0.089	5.322	0.322
4500	0.354	3.743	0.322	0.087	5.271	0.357
5000	0.314	5.374	0.391	0.088	7.440	0.372
5500	0.315	6.700	0.517	0.087	9.164	0.500
6000	0.327	7.177	0.568	0.088	9.739	0.588
7000	0.327	11.317	0.743	0.088	14.680	0.716
8000	0.322	16.621	1.016	0.089	20.627	0.903
9000	0.331	20.930	1.043	0.087	25.422	0.990
10000	0.484	32.060	1.545	0.089	36.076	1.487
20000	0.353	238.890	5.800	0.087	252.194	5.205
30000	0.364	922.844	13.436	0.087	945.575	12.564
40000	0.464	2545.692	24.036	0.089	2576.327	23.055
50000	0.365	-	-	0.089	5808.306	37.588

Table 8.12: Time comparison for verifying FireWire with/without Kronecker

N cycles	Nodes			
	Explicit		Kronecker	
	PTA	MDP	PTA	MDP
3	1777	1135	342	564
4	5325	3173	584	996
5	12833	7634	886	1530
6	28105	14313	1248	2166
7	58300	29843	1670	2904
8	122861	60424	2151	3744
9	252159	124872	2694	4686

Table 8.13: Memory comparison for verifying Milner's scheduler with/without Kronecker

N cycles	Time	
	Explicit	Kronecker
	Constr.	Constr.
3	0.192	0.059
4	0.532	0.053
5	1.600	0.058
6	11.528	0.065
7	19.387	0.076
8	71.926	0.085
9	241.336	0.102

Table 8.14: Time comparison for verifying Milner's scheduler with/without Kronecker

process, the construction of the product model, the forward searching and probabilistic verification, are based on MTBDDs. This is efficient because, firstly, it avoids explicit construction of the model by directly translating the description of the model into MTBDDs via the Kronecker expression; secondly, it completes the verification in a single step rather than invoking several steps via textual files. Furthermore, the implementation is generic as it can support different data structures for the representation of timing information. However, our attempt to exploit the structure inside the zones, namely, to adapt the idea from [DKN02] proved to be a disadvantage compared with that with no such adaptation.

Chapter 9

Conclusions

9.1 Summary and Evaluation

The aim of this work was to develop an efficient model checker for probabilistic timed automata. We started with an explicit implementation of both forward and backward algorithm for model checking probabilistic timed automata and identified the bottleneck of the explicit implementation. Then we investigate how symbolic model checking technique based on MTBDDs can be exploited to achieve our goal.

Our MTBDD-based symbolic implementation is generic with respect to the support of the real-timed information. We have shown that two data structures representing the real-timed information could be supported. Besides the DBMs and DDDs used in our implementation, other existing data structures, such as CDDs and CRDs which were invented for storing zones, can also be supported.

Due to the lack of existing symbolic implementations of model checking for probabilistic timed automata, we are concerned with the comparison between our explicit implementation and symbolic implementation. The comparison is fair in that tests are carried out on identical examples and operations on zones and under the same running environments.

In Chapter 6, we presented an explicit implementation for the forward algorithm. We used the explicit data structure for storing the state space and implemented our own DBM package for storing zones. However, the real focus of this chapter is to identify the

bottleneck. We showed that a dominant bottleneck arises at the last step, that is when the tool PRISM was applied to construct the generated MDP model from the textual files in PRISM language into MTBDDs when the generated MDP model is large and contains many states and transitions.

We continued the analysis of an explicit implementation for the backward algorithm in Chapter 7. Unlike the forward algorithm, the backward algorithm is more complex in terms of the computational complexity. In the case of model checking the maximum probabilistic reachability, the algorithm repeatedly performs zone conjunction operations in order to detect the missing edges between states. In the case of model checking the minimum probabilistic reachability, the computation complexity is even worse because it involves the computation of 3-nested loops and non-convex zones are common during computation. In terms of the equivalence checking of two non-convex zones, neither DBMs nor DDDs have efficient means of doing this because both of them lack the mechanism to support the canonicity property. In DBMs, equivalence checking involves the complementation operation, which again results in non-convex zones, while in DDDs, it is through set inclusion. In the backward algorithm, the *tpre* operation is a critical operation, especially in the case of the computation of the minimum probability. The DBM-based implementation of the *tpre* operation contains double complementation operations. On the other hand, in the DDD-based implementation, it is memory-intensive because it generates many intermediate nodes. One of the most important lessons we learnt from the implementation for the backward algorithm is that the backward algorithm, especially the one for computing the minimum probability, could not be implemented efficiently without a suitable data structure which not only can efficiently represent non-convex zones via a canonical form, but has an efficient support of the requirement of the *tpre* operation as well.

In Chapter 8, we present a symbolic implementation of the forward algorithm. Compared with the explicit implementation, the symbolic implementation is more efficient with respect to both time and space consumption. We also showed that DBMs outperforms DDDs in the forward algorithm. The reason behind this is that DDDs lack the canonical form and the operations required generate many intermediate nodes. One limi-

tation of the symbolic implementation is that it has not exploited the potential structure within the zones. We demonstrated that a relabelling approach could result in a smaller state space in terms of the generated MDP model, but the overall time and space requirement is greater than the approach without relabelling. Furthermore, we presented how the Kronecker-based technique can be combined with the MTBDD-based implementation to directly construct the probabilistic timed automata into MTBDDs and introduced an model description language for the probabilistic timed automata based on the PRISM language. The exploitation of Kronecker-based technique makes the process of model construction efficient. In terms of the model construction, the Kronecker-based technique ensures the feasibility of the analysis of extremely large models with a structured pattern.

9.2 Conclusions and Discussion of Future Work

In conclusion, we have successfully demonstrated that the MTBDD-based symbolic implementation of model checking probabilistic timed automata is feasible in practice and can be efficient on some real-world protocols. Our MTBDD-based symbolic implementation is generic with respect to the support of the real-timed information. The exploitation of MTBDD-based and Kronecker-based techniques could be extended to model checking non-probabilistic timed systems, for example, timed automata. Similarly to other work on model checking non-probabilistic timed systems, the conclusion we draw from the results is that a data structure with the canonical form for representing and operating non-convex zones is the most important factor. Among all existing data structures supporting non-convex zones in literature, the CRDs [Wan03] seem to have best performance at the moment, and we conjecture that our implementation would perform better if DBMs or DDDs was replaced with CRDs. One future research direction is to use our symbolic implementation and generalise it to the probabilistic hybrid system models.

9.2.1 Consideration of Symbolic Implementation of the Backward Algorithm

In Chapter 8, we have symbolically implemented the algorithm for the forward reachability. However, no symbolic implementation of the backward reachability has been developed in this thesis. In order to efficiently implement the algorithm for the backward reachability, there are two issues we have to address. One issue is mentioned above, that is that a suitable data structure which can not only efficiently represent non-convex zones via a canonical form, but also has an efficient support for the *tpre* operation. The other issue is how to efficiently represent the non-deterministic choices dynamically generated from the backward algorithm. For the forward algorithm, such problem does not arise because the generated non-deterministic choices are directly derived from the original syntactical models of PTAs, and there is a one-to-one correspondance between the generated MDP models and the original PTA models. However, in the case of the backward algorithm, more than one non-deterministic choice could be generated for each non-deterministic choice appearing in the original PTA model. In the case of explicit implementation of the backward algorithm, the construction of non-deterministic choices is done only after all the reachable symbolic states and transitions among them are generated. However, in the case of symbolic implementation of the backward algorithm, the non-deterministic choices are dynamically generated whenever a new transition is encountered. The dynamic nature of generating non-deterministic choices has two effects on the construction process. One effect is that the number of the non-deterministic choices is unknown until the algorithm terminates, which is similar to the number of zones in the case of our symbolic implementation of the forward algorithm. The other effect is that one has to find an efficient way to construct the MTBDD variables representing the non-deterministic choices. Figure 9.1 shows a subgraph of a probabilistic timed automaton with one distribution of two branches, where A, B and C are nodes, p_1 and p_2 are probability values when transitions are taken from node A to node B and to node C respectively, and $p_1 + p_2 = 1$ (the guard and clock resetting are omitted). Figure 9.2 shows a subgraph of the zone graph that could be generated, with two non-deterministically chosen distributions, where zone A_1 is derived from node A, zones B_1 and B_2 are derived from node B and zone C_1 is derived

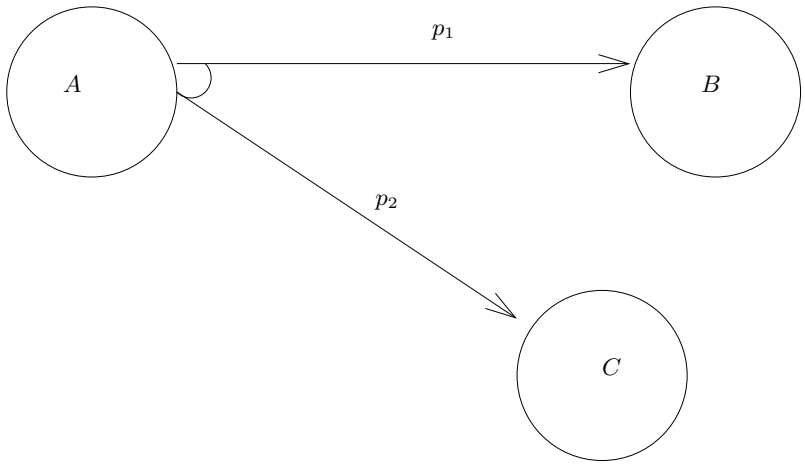

Figure 9.1: A example of a subgraph of a timed automaton with one distribution

from node C. In general, if a distribution has n branches, and the number of zones derived from the i-th branch is b_i (if i-th branch does not appear, b_i is replaced with 1), $1 \leq i \leq n$, the total number of non-deterministic choices for the zone derived from the source node is $b_1 \times b_2 \cdots \times b_i \cdots \times b_n$.

9.2.2 How to Achieve DDD-based Operation *Normalise*

The forward algorithm for the (non-probabilistic) timed systems does not always terminate. Hence, the *c-equivalence* and its corresponding operation (*Normalise*) play an important role to guarantee the termination of the forward reachability search. However, the operation *Normalise* is only defined on the DBMs representing the convex zones.

For the forward probabilistic reachability search, each zone obtained is convex, and can be stored as a single DBM. (A convex zone in DDDs can only have one path, this is also true for CRDs or CDDs.)

For single-path DDDs, the convex zone can be first transformed into a DBM on which the operation *Normalise* can be applied, and then transformed back into a DDD. The case of CRDs and CDDs can be handled in a similar way.

In the case of using DDDs to represent the generic zones including non-convex zones encountered during the non-probabilistic forward reachability search, the termination of

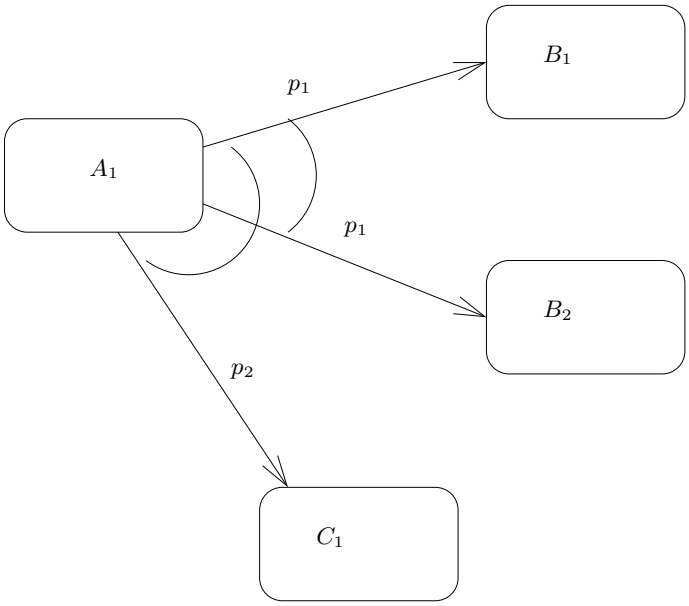

Figure 9.2: A example of sub zone graph with two non-deterministic choices corresponding to Figure 9.1

the forward algorithm can still be achieved because a DDD can be treated as a union of DBMs and each DBM corresponds to a path of the DDD. The transformation is done by first extracting all paths from a DDD and then applying operation *Normalise* to each path of the DDD, and finally combining all transformed single-path DDDs together by disjunction. Although the above approach is feasible, it is inefficient because of the need to perform transformation between DBMs and DDDs and to enumerate the paths of DDDs. It is desirable to find a method which could directly operate on DDDs to ensure termination.

9.2.3 Consideration of Merging of DDDs and MTBDDs

The symbolic method used in Chapter 8 for the forward reachability has a drawback, that is, the upper bound on the number of the zones encountered has to be estimated before-hand. One potential solution to this problem is to develop a new data structure which combines both functionalities of DDDs and MTBDDs.

We recall the issues that are mentioned in Chapter 8 concerning the requirements of such a data structure.

- How to represent and manipulate the sets of states which contain both a discrete part and a continuous part.
- How to represent the timed transition relation between states.
- How to represent the probabilistic transition relation between states.
- How to construct the dynamically generated information, states and probabilistic transitions between states.

Two possible solutions exist. One way is to extend MTBDDs and the other is to extend DDDs. Below we discuss how the questions can be answered one by one for each method.

For the former, [SB03] proposed a way to apply MTBDDs to real-time systems. The technique used in [SB03] can address questions in point 1 and 2 above because it uses a single data structure, MTBDDs, to represent both timing and discrete information, and it can represent the timed transition relation between states.

The second method is to extend DDDs. Similarly to the solution based on MTBDDs, this method can address questions in point 1 and 2.

In order to support probabilistic timed systems, what remains to be done is how to deal with the questions in point 3 and 4.

Although the method in [SB03] has been applied to timed systems, it has not been applied to the probabilistic timed systems. Because MTBDDs can naturally support real numbers, the answer to question in point 3 is easy to obtain: we only need to replace the terminal node with a real number representing the probabilistic value to be taken when the corresponding timed transition is chosen.

However, since DDDs have only two terminal nodes (true and false or 0 and 1), in order to support probability, first we must extend DDDs to enable them to have real values as terminal nodes besides terminal 0 and 1.

Now the only open question left is point 4. As methods based on either MTBDDs or DDDs will face similar problems, we address this by discuss them together. In the case of untimed probabilistic systems, a probabilistic transition between any two states is represented as one path of MTBDDs. In the case of probabilistic timed systems, this is

not the case because the states which are dynamically generated could not only be convex zones, which can be represented as a single-path MTBDD/DDD, but non-convex zones represented as a multi-path MTBDD/DDD as well. In other words, a state representation in MTBDDs/DDDs is a DAG (Directed Acyclic Graph) in general. This means that the probabilistic transitions between such states involve two DAGs. As a result, one has to consider two issues raised by DAGs appearing in a probabilistic transition. The first issue is the variable ordering among the unprimed and primed MTBDDs/DDDs variables. In the case of untimed probabilistic systems, there is a well-known heuristic on variable ordering which is to alternate the unprimed and primed variables. However, in the case of probabilistic timed systems, we would conjecture that it is better to put all unprimed variables before primed ones, or vice versa, to improve efficiency. The reason that this is more efficient is that performing of the existential quantification operations would cause fewer changes in MTBDDs/DDDs structure, and as is well known structure changes in MTBDDs/DDDs would result in bad performance in general.

The second issue is that traditional recursive algorithms cannot be applied in the case of two DAGs appearing along a probabilistic transition. This means that one needs to develop new traversal algorithms which can distinguish the DAGs instead of single path of MTBDDs/DDDs along the probabilistic transitions. Algorithms for matrix multiplication also have to be developed accordingly.

Appendix A

Model Checker for The Probabilistic Timed Automata

This appendix gives a brief description of the model checker developed and case studies used in this thesis. Section A.1 gives a brief overview of the model checker which is using techniques described in this thesis. A brief description of case studies used in this thesis is introduced in Section A.2.

A.1 Tool Overview

This section gives a brief overview of the model checker for probabilistic timed automata. The techniques described in this thesis have been implemented in a software tool. There are two versions of the tool. One is the explicit implementation and the other is the symbolic implementation. The tool accepts a model description and a reachability property. Two kinds of model description are supported. One is for the explicit version and the other is for the symbolic version. Depending on the version used, the corresponding input for models are parsed and approiate algorithms are used for model checking. Figure A.1 shows the structure of the model checker for probabilistic timed automata. Both versions are based on the command line interface. The explicit version supports both the forward and backward algorithms, while the symbolic version supports only the forward algorithm. The symbolic version is based on MTBDDs for constructing and storage of the

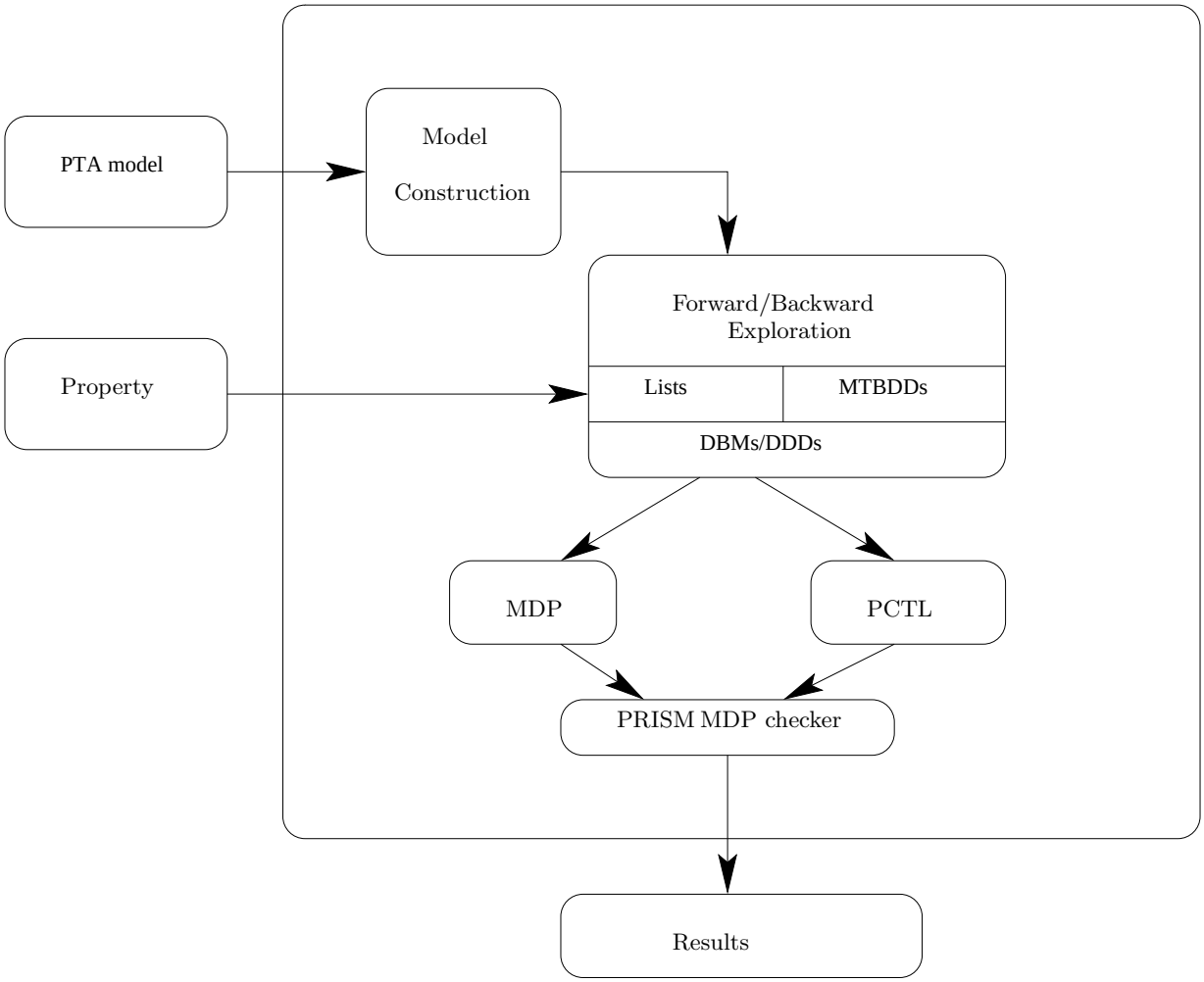

Figure A.1: The model checker for probabilistic timed automata

PTA syntax and generated MDP models. For the representation of timing information, the tool supports two kinds of representation: DBMs and DDDs.

A.2 Case Studies

A.2.1 CSMA/CD

CSMA/CD (Carrier Sense Multiple Access / Collision Detection) is the protocol used in Ethernet networks to make sure that only one station of the network is transmitting on the network wire at any time. If no transmission is taking place at the time, the

particular station can transmit. CSMA/CD is a type of contention protocol. When two stations attempt to transmit data simultaneously a collision occurs. If a collision is detected by all participating stations, after a random time interval, the stations that collided attempt to transmit again. If another collision occurs, the time intervals from which the random waiting time is selected are increased. The property we analysed is the maximum probability of both stations correctly delivering their packets by the deadline. The model we consider here is a probabilistic extension of the timed automata model given in [NSY92]. Models and case studies can be also found in [KNSW04, PRI]. Figure A.12 and A.13 is the network medium and station PTA for full model of CSMA/CD.

A.2.2 IEEE 1394 FireWire Root Contention

The 1394 High Performance serial bus is used to transport digital video and audio streams within a multimedia network. In essence, the tree identify process of IEEE 1394 is a leader election protocol which takes place after a bus reset in the network (i.e. when a node is added to, or removed from, the network). The protocol of IEEE 1394 is a randomised leader election algorithm designed for the establishment of a tree topology. If two nodes contend the leadership (root contention), the contenders resolve the situation by timing and choosing nondeterministically whether to wait for a long or short time. We have analysed the following probabilistic aspects of the protocol: the minimum probability that, from the initial state, a leader (root) is chosen before the deadline is reached. Both full and abstract models are based on probabilistic I/O automata models presented in [SV99], and models and case studies can be also found in [DKN02, KNS03a, PRI]. Figure A.9 is the PTA for abstract model $I_1^p$of FireWire. Figure A.10 and A.11 is the node and wire PTA for full model $impl^p$ of FireWire.

A.2.3 Milner's Scheduler with Only One Clock

Milner's scheduler [Mil89] consists of N cycles which are connected in a ring and cooperating on controlling N tasks. The version of Milner's scheduler with only one clock has a number of discrete states for each cycle. However, the number of state and transition of the systems grow exponentially in N. This case is used to test the scalability of our

symbolic method based on Kronecker, compared with the explicit method, to construct the product model of probabilistic timed automata. We have analysed the following probabilistic aspects of the protocol: the maximum probability of any two cycles being concurrent in the task mode. Our model is based on model presented in [MLAH99c] augmented with probability one for each transition. Figure A.8 is the PTA for model of Milner's scheduler.

A.3 Model Description Language for One Case Study

We include an example of the textual language description for one model of our case studies: the abstract model of FireWire with deadline 1000. Figure A.2 and A.3 give the description for symbolic version. The property to be checked is given in Figure A.4. The command to run this example under Linux is as follows:
$mytst\ PRISM\ *.nm\ *.pctl.$

A.4 Computations of the Minimum Probability for Example 5.3

Appendix A.5, A.6, A.7 show computations performed for $MaxV_{\geq 1}(2, (false), (close))$.

nondeterministic

module M_1

// local state

// 0 - startstart

// 1 - faststart

// 2 - slowstart

// 3 - startfast

// 4 - startslow

// 5 - fastfast

// 6 - fastslow

// 7 - slowfast

// 8 - slowslow

// 9 - seldone

// 10 - after

$loc : [0..10]$ **init** 0;

$guard : [0..7]$ **init** 0;

$reset : [0..3]$ **init** 0;

$[](loc = 0) \rightarrow 0.5 : (loc' = 1) \wedge (guard' = 0) \wedge (reset' = 0) + 0.5 : (loc' = 2) \wedge (guard' = 0) \wedge (reset' = 0);$

$[](loc = 0) \rightarrow 0.5 : (loc' = 3) \wedge (guard' = 0) \wedge (reset' = 0) + 0.5 : (loc' = 4) \wedge (guard' = 0) \wedge (reset' = 0);$

$[](loc = 1) \rightarrow 0.5 : (loc' = 5) \wedge (guard' = 0) \wedge (reset' = 1) + 0.5 : (loc' = 6) \wedge (guard' = 0) \wedge (reset' = 1);$

$[](loc = 2) \rightarrow 0.5 : (loc' = 7) \wedge (guard' = 0) \wedge (reset' = 1) + 0.5 : (loc' = 8) \wedge (guard' = 0) \wedge (reset' = 1);$

$[](loc = 3) \rightarrow 0.5 : (loc' = 5) \wedge (guard' = 0) \wedge (reset' = 1) + 0.5 : (loc' = 7) \wedge (guard' = 0) \wedge (reset' = 1);$

$[](loc = 4) \rightarrow 0.5 : (loc' = 6) \wedge (guard' = 0) \wedge (reset' = 1) + 0.5 : (loc' = 8) \wedge (guard' = 0) \wedge (reset' = 1);$

$[](loc = 5) \rightarrow 1 : (loc' = 0) \wedge (guard' = 1) \wedge (reset' = 1);$

$[](loc = 5) \rightarrow 1 : (loc' = 9) \wedge (guard' = 2) \wedge (reset' = 2);$

$[](loc = 6) \rightarrow 1 : (loc' = 9) \wedge (guard' = 3) \wedge (reset' = 2);$

$[](loc = 7) \rightarrow 1 : (loc' = 9) \wedge (guard' = 3) \wedge (reset' = 2);$

$[](loc = 8) \rightarrow 1 : (loc' = 0) \wedge (guard' = 4) \wedge (reset' = 1);$

$[](loc = 8) \rightarrow 1 : (loc' = 9) \wedge (guard' = 3) \wedge (reset' = 2);$

$[](loc = 9) \rightarrow 1 : (loc' = 10) \wedge (guard' = 5) \wedge (reset' = 0);$

$[](loc = 10) \rightarrow 1 : (loc' = 10) \wedge (guard' = 0) \wedge (reset' = 0);$

endmodule

Figure A.2: The textual description of the abstract model of FireWire

$< clocks >$

x

a

z

$< /clocks >$

$< module >$

$< invariants >$

$0 - x <= 36$

$1 - x <= 36$

$2 - x <= 36$

$3 - x <= 36$

$4 - x <= 36$

$5 - x <= 85$

$6 - x <= 167$

$7 - x <= 167$

$8 - x <= 167$

$9 - a <= 0$

$10 - true$

$< /invariants >$

$< guard >$

$0 - true$

$1 - x >= 76$

$2 - x >= 40$

$3 - x >= 123$

$4 - x >= 159$

$5 - z > 1000, a >= 0$

$< /guard >$

$< reset >$

$0 - null$

$1 - x = 0$

$2 - a = 0$

$< /reset >$

$< /module >$

Figure A.3: The textual description of the timing information for the abstract model of FireWire

$$Pmax = ?[true\ U\ (loc = 10)\ \{\text{“}init''\}]$$

Figure A.4: The property

$MaxV_{\geq 1}(2, \{false\}, \{close\})$
$Z := \{true\}$
$repeat$
$Y := Z$
$Z := \{(close)\} \wedge z.MaxU1(Y, (z > 2))$
$Z := \{(close)\} \wedge \{true\}$
$Z := \{(close)\}$
$Y := Z$
$Z := \{(close)\} \wedge z.MaxU1(Y, (z > 2))$
$Z := \{(close)\} \wedge z.(\{(close)\} \vee \{(open, x < 3 \wedge x - z < 1 \wedge z > 2)\})$
$Z := \{(close)\} \wedge \{\{(close)\} \vee \{(open)\}\}$
$Z := \{(close)\}$
$endrepeat$

Figure A.5: Calculation steps for the minimum probability of Example 5.3

$MaxU_{\geq 1}(\{true\}, (z > 2))$
$Z_0 := \{true\}$
$repeat$
$Y_0 := Z_0$
$Z_1 := \{(z > 2)\}$
$repeat$
$Y_1 := Z_1$
$Z_1 := \{(z > 2)\} \vee \{true\} \wedge pre1(Y_0, Y_1)$
$Z_1 := \{(z > 2)\} \vee \{true\} \wedge (\{(close)\} \vee \{(open, 2 < x < 3 \wedge x - z < 3)\})$
$Z_1 := Z_1 \vee tpre_{true}(Y_0 \wedge Y_1)$
$Z_1 := Z_1 \vee \{(close)\} \vee \{(open, x < 3 \wedge x - z < 1)\}$
$Z_1 := \{(close)\} \vee \{(open, 2 < x < 3 \wedge x - z < 3)\} \vee \{(open, x < 3 \wedge x - z < 1)\}$
$Y_1 := Z_1$
$Z_1 := \{(z > 2)\} \vee \{true\} \wedge pre1(Y_0, Y_1)$
$Z_1 := \{(z > 2)\} \vee \{true\} \wedge \{(close)\} \vee \{(open, 2 < x < 3 \wedge x - z < 3)\}$
$Z_1 := Z_1 \vee tpre_{true}(Y_0 \wedge Y_1)$
$Z_1 := Z_1 \vee \{(close)\} \vee \{(close)\} \vee \{(open, x < 3)\}$
$Z_1 := \{(close) \vee \{(open, x < 3)\}$
$Y_1 := Z_1$
$Z_1 := \{(z > 2)\} \vee \{true\} \wedge pre1(Y_0, Y_1)$
$Z_1 := \{(z > 2)\} \vee \{true\} \wedge \{(close)\} \vee \{(open, 2 < x < 3 \wedge x - z < 3)\}$
$Z_1 := Z_1 \vee tpre_{true}(Y_0 \wedge Y_1)$
$Z_1 := Z_1 \vee \{(close)\} \vee \{(close)\} \vee \{(open, x < 3)\}$
$Z_1 := \{(close)\} \vee \{(open, x < 3)\}$
$endrepeat$
$Z_0 := Z_1$
$Z_0 := \{(close)\} \vee \{(open, x < 3)\}$
$Z_0 := \{true\}$
$endrepeat$

Figure A.6: Calculation steps for the minimum probability of Example 5.3

$MaxU_{\geq 1}((close), (z > 2))$
$Z_0 := \{true\}$
$repeat$
$Y_0 := Z_0$
$Z_1 := \{(z > 2)\}$
$repeat$
$Y_1 := Z_1$
$Z_1 := \{(z > 2)\} \vee \{(close)\} \wedge pre1(Y_0, Y_1)$
$Z_1 := \{(z > 2)\} \vee \{(close)\} \wedge \{(close)\} \vee \{(open, 2 < x < 3 \wedge x - z < 3)\}$
$Z_1 := Z_1 \vee tpre_{\{(close)\} \vee \{(z>2)\}}(Y_0 \wedge Y_1)$
$Z_1 := Z_1 \vee \{(close)\} \vee \{(open, x < 3 \wedge x - z < 1, \wedge z > 2)\}$
$Z_1 := \{(close)\} \vee \{(open, x < 3 \wedge x - z < 1 \wedge z > 2)\}$
$Y_1 := Z_1$
$Z_1 := \{(z > 2)\} \vee \{(close)\} \wedge pre1(Y_0, Y_1)$
$Z_1 := \{(z > 2)\} \vee \{(close)\} \wedge \{(close, (z > 2)\} \vee \{(open, 2 < x < 3 \wedge x - z < 1 \wedge z > 2)\}$
$Z_1 := Z_1 \vee tpre_{\{(close)\} \vee \{(z>2)\}}(Y_0 \wedge Y_1)$
$Z_1 := Z_1 \vee \{close)\} \vee \{(close)\} \vee \{(open, x < 3 \wedge x - z < 1 \wedge z > 2)\}$
$Z_1 := \{(close)\} \vee \{(open, x < 3 \wedge x - z < 1 \wedge z > 2)\}$
$endrepeat$
$Z_0 := Z_1$
$Z_0 := \{(close)\} \vee \{(open, x < 3 \wedge x - z < 1 \wedge z > 2)\}$
$endrepeat$

Figure A.7: Calculation steps for the minimum probability of Example 5.3

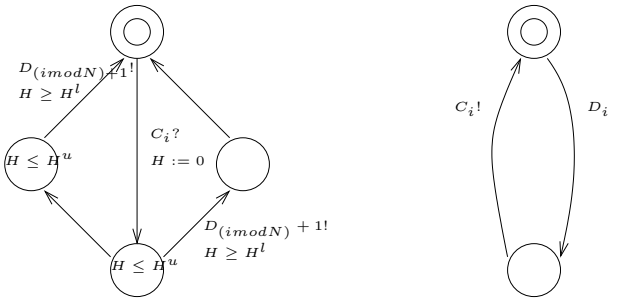

Figure A.8: The PTA for model of Milner's scheduler

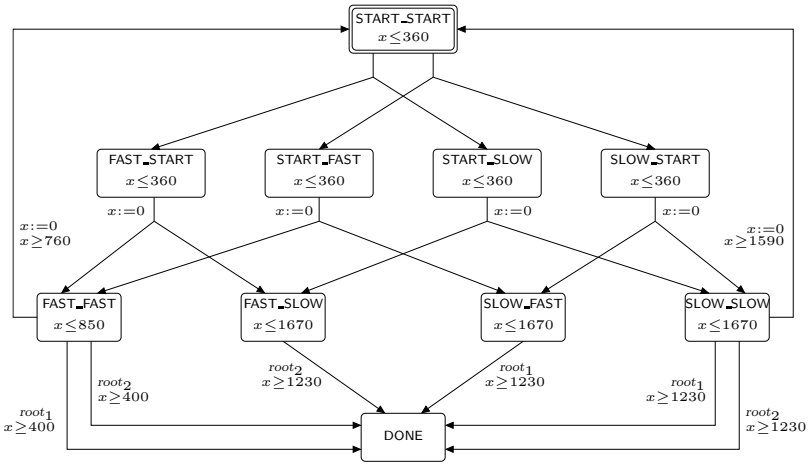

Figure A.9: The PTA for the abstract model of FireWire

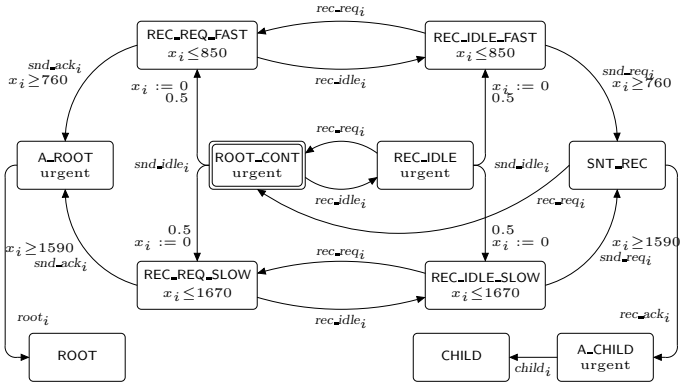

Figure A.10: The PTA node for the full model of FireWire

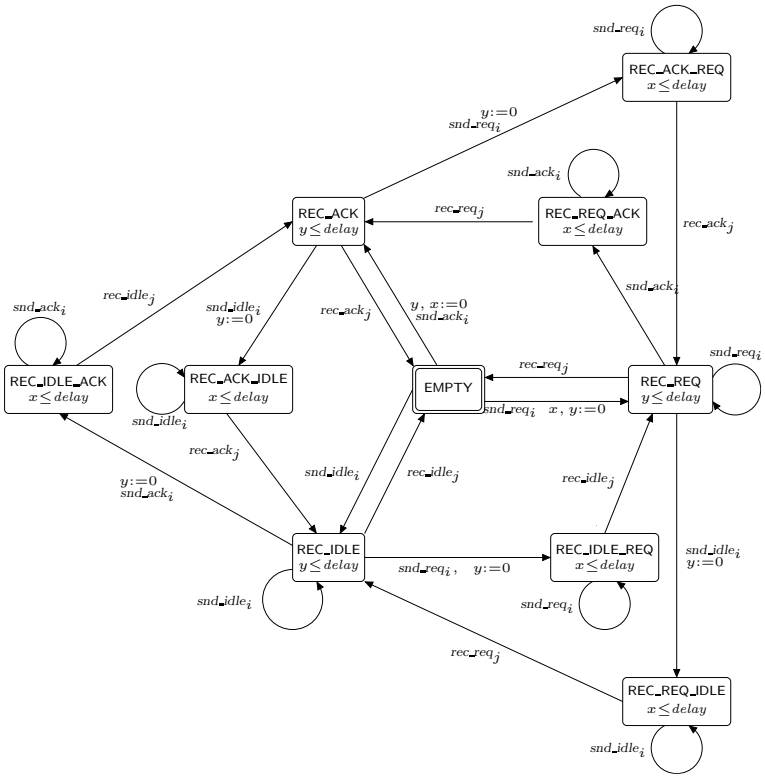

Figure A.11: The PTA wire for the full model of FireWire

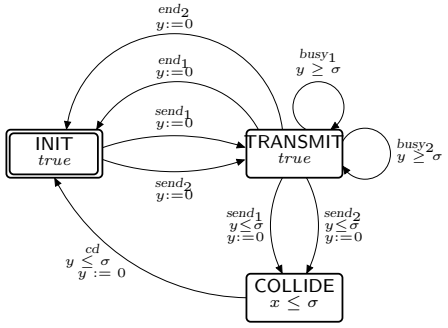

Figure A.12: The PTA medium for the model of CSMA/CD

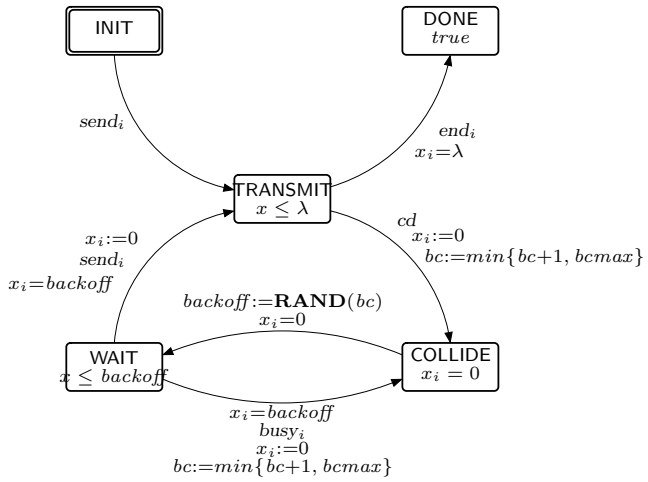

Figure A.13: The PTA sender for the model of CSMA/CD

Bibliography

[ACD90] R. Alur, C. Courcoubetis, and D. L. Dill. Model checking for real-time systems. In *Proc. Fifth Annual Symposium on Logic in Computer Science*, pages 414–425, 1990.

[ACD91a] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking for probabilistic real-time systems (extended abstract). In *Automata, Languages and Programming*, pages 115–126, 1991.

[ACD91b] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Verifying automata specifications of probabilistic real-time systems. In *REX Workshop*, pages 28–44, 1991.

[ACD93] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AH91] R. Alur and T.A. Henzinger. Logics and Models of Real-Time: A Survey. In *Real Time: Theory in Practice*, volume 600, pages 74–106. Springer-Verlag, 1991.

[Ake78] S. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, 1978.

[ASSB96] A. Aziz, K. Sanwal, V. Singhal, and R. K. Brayton. Verifying continuous-time markov chains. In Rajeev Alur and Thomas A. Henzinger, editors,

Eighth International Conference on Computer Aided Verification CAV, volume 1102, pages 269–276, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[Bai98] C. Baier. On algorithmic verification methods for probabilistic systems. Habilitation thesis, Fakultät für Mathematik & Informatik, Universität Mannheim, 1998.

[BBF+01] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and Software Verification- Model-checking Techniques and Tools*. Springer, 2001.

[BC95] Randal E. Bryant and Yirng-An Chen. Verification of arithmetic circuits with binary moment diagrams. In *Design Automation Conference*, pages 535–541, 1995.

[BCDK00] Peter Buchholz, Gianfranco Ciardo, Susanna Donatelli, and Peter Kemper. Complexity of memory-efficient kronecker operations with applications to the solution of markov models. *INFORMS J. on Computing*, 12(3):203–222, 2000.

[BCHG+97] C. Baier, E. Clarke, V. Hartonas-Garmhausen, M. Kwiatkowska, and M. Ryan. Symbolic model checking for probabilistic processes. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Proc. 24th International Colloquium on Automata, Languages and Programming (ICALP'97)*, volume 1256 of *LNCS*, pages 430–440. Springer, 1997.

[BCL91] J.R. Burch, E.M. Clarke, and D.E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P.B. Denyer, editors, *International Conference on Very Large Scale Integration*, pages 49–58, Edinburgh, Scotland, 1991. North-Holland.

[BCL+94] J.R. Burch, E.M. Clarke, D.E. Long, K.L. MacMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.

[BCM+90] J. Burch, E. Clarke, K. McMillan, D. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th Annual IEEE Symposium on Logic in Computer Science (LICS'90)*, pages 428–439. IEEE Computer Society Press, 1990.

[BdA95] Bianco and de Alfaro. Model checking of probabilistic and nondeterministic systems. *FSTTCS: Foundations of Software Technology and Theoretical Computer Science*, 15, 1995.

[BDM+98] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A model-checking tool for real-time systems. In A. J. Hu and M. Y. Vardi, editors, *Proc. 10th International Conference on Computer Aided Verification, Vancouver, Canada*, volume 1427, pages 546–550. Springer-Verlag, 1998.

[Bea03] Danièle Beauquier. On probabilistic timed automata. *Theor. Comput. Sci.*, 292(1):65–84, 2003.

[BK98] C. Baier and M. Kwiatkowska. Model checking for a probabilistic branching time logic with fairness. *Distributed Computing*, 11(3):125–155, 1998.

[BKH99a] Christel Baier, Joost-Pieter Katoen, and Holger Hermanns. Approximate symbolic model checking of continuous-time markov chains. In *International Conference on Concurrency Theory*, pages 146–161, 1999.

[BKH99b] Christel Baier, Joost-Pieter Katoen, and Holger Hermanns. Approximate symbolic model checking of continuous-time markov chains. In *International Conference on Concurrency Theory*, pages 146–161, 1999.

[BLL+98] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, W. Yi, and C. Weise. New generation of uppaal, 1998.

[BLP+99] Gerd Behrmann, Kim Guldstrand Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Efficient timed reachability analysis using clock difference diagrams. In *Computer Aided Verification*, pages 341–353, 1999.

[Bry86] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[BTY97a] A. Bouajjani, S. Tripakis, and S. Yovine. On-the-fly symbolic model checking for real-time systems. In *Proc. 18TH IEEE Real-Time Systems Symposium*, pages 25–34, Los Alamitos, 1997. IEEE CS Press.

[BTY97b] A. Bouajjani, S. Tripakis, and S. Yovine. On-the-fly symbolic model checking for real-time systems. In *RTSS '97: Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, page 25, Washington, DC, USA, 1997. IEEE Computer Society.

[BY04] J. Bengtsson and W Yi. Timed automata: Semantics, algorithms and tools. In *4th Advanced Course on Petri Nets*, volume 3098 of *LNCS*, pages 87–124. Springer, 2004.

[CCMM95] Sergio Vale Aguiar Campos, Edmund M. Clarke, Wilfredo R. Marrero, and Marius Minea. Verus: A tool for quantitative analysis of finite-state real-time systems. In *Workshop on Languages, Compilers and Tools for Real-Time Systems*, pages 70–78, 1995.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[CFM+93] E. Clarke, M. Fujita, P. McGeer, K. McMillan, J. Yang, and X. Zhao. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. In *Proc. IWLS'93*, pages 1–15, 1993. Also available in *Formal Methods in System Design*, 10(2/3):149–169, 1997.

[CGH+93] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the Futurebus+ Cache Coherence Protocol. In D. Agnew, L. Claesen, and R. Camposano, editors, *The Eleventh International Symposium on Computer Hardware Description Languages and their Applications*, pages 5–20, Ottawa, Canada, 1993. Elsevier Science Publishers B.V., Amsterdam, Netherland.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CGP99] Edmund M. Clarke, Jr. Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[Cha73] Chin-Liang Chang. *Symbolic logic and mechanical theorem proving*. Academic Press, 1973.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT press, 1990.

[CWA+96] Edmund M. Clarke, Jeannette M. Wing, Rajeev Alur, Rance Cleaveland, David Dill, Allen Emerson, Stephen Garland, Steven German, John Guttag, Anthony Hall, Thomas Henzinger, Gerard Holzmann, Cliff Jones, Robert Kurshan, Nancy Leveson, Kenneth McMillan, J. Moore, Doron Peled, Amir Pnueli, John Rushby, Natarajan Shankar, Joseph Sifakis, Prasad Sistla, Bernhard Steffen, Pierre Wolper, Jim Woodcock, and Pamela Zave. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[CY88] C. Courcoubetis and M. Yannakakis. Verifying temporal properties of finite state probabilistic programs. In *Proc. 29th Annual Symposium on Foundations of Computer Science (FOCS'88)*, pages 338–345. IEEE Computer Society Press, 1988.

[CY90] C. Courcoubetis and M. Yannakakis. Markov decision processes and regular events. In M. Paterson, editor, *Proc. 17th International Colloquium on Automata, Languages and Programming (ICALP'90)*, volume 443 of *LNCS*, pages 336–349. Springer, 1990.

[dAKN+00] L. de Alfaro, M. Kwiatkowska, G. Norman, D. Parker, and R. Segala. Symbolic model checking of concurrent probabilistic processes using MTBDDs and the Kronecker representation. In S. Graf and M. Schwartzbach, editors, *Proc. 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, volume 1785 of *LNCS*, pages 395–410. Springer, 2000.

[Dil89] D. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proc. Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 197–212. Springer, 1989.

[DKN02] C. Daws, M. Kwiatkowska, and G. Norman. Automatic verification of the IEEE 1394 root contention protocol with KRONOS and PRISM. In R. Cleaveland and H. Garavel, editors, *Proc. 7th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'02)*, volume 66.2 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[DKN04] 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 (STTT)*, 5(2–3):221–236, 2004.

[DOTY95] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III: Verification and Control*, volume 1066, pages 208–219, Rutgers University, New Brunswick, NJ, USA, 22–25 October 1995. Springer.

[DT98] Conrado Daws and Stavros Tripakis. Model checking of real-time reachability properties using abstractions. *Lecture Notes in Computer Science*, 1384:313–329, 1998.

[Duf91] David A Duffy. *Principles of automated theorem proving.* Wiley, 1991.

[DY95] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. In *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS'95),* Pisa, Italy, pages 66–75, 1995.

[GH02] H. Garavel and H. Hermanns. On combining functional verification and performance evaluation using cadp. In *In FME 2002: International Symposium of Formal Methods Europe*, volume 2391 of *LNCS*, pages 410–429. Springer, 2002.

[GKPP95] Rob Gerth, Ruurd Kuiper, Doron Peled, and Wojciech Penczek. A partial order approach to branching time logic model checking. In *Proceedings of the Third Israel Symposium on the Theory of Computing and Systems (ISTCS'95), Tel Aviv, Israel, January 4-6, 1995*, 1995.

[HGCC99] V. Hartonas-Garmhausen, S. Campos, and E. Clarke. ProbVerus: Probabilistic symbolic model checking. In J.-P. Katoen, editor, *Proc. 5th International AMAST Workshop on Real-Time and Probabilistic Systems (ARTS'99)*, volume 1601 of *LNCS*, pages 96–110. Springer, 1999.

[HJ94] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

[HKMKS00] Holger Hermanns, Joost-Pieter Katoen, Joachim Meyer-Kayser, and Markus Siegle. A markov chain model checker. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 347–362, 2000.

[HKN+03] H. Hermanns, M. Kwiatkowska, G. Norman, D. Parker, and M. Siegle. On the use of MTBDDs for performability analysis and verification of stochastic systems. *Journal of Logic and Algebraic Programming*, 56(1-2):23–67, 2003.

[HM05] Dang Van Hung and Zhang Miaomiao. On verification of probabilistic timed automata against probabilistic duration properties. *Technical Report 326*, UNU-IIST, P.O.Box 3058, Macau, 2005.

[HNSY92] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. In *7th. Symposium of Logics in Computer Science*, pages 394–406, Santa-Cruz, California, 1992. IEEE Computer Scienty Press.

[HR00] Michael Huth and Mark Ryan. *Logic in Computer Science: modelling and reasoning about systems*. Cambridge University Press, 2000.

[HYT] HYTECH. HyTech WebSite. http://www-cad.eecs.berkeley.edu/~tah/HyTech/.

[JWK+96] J. Bengtsson, W. O. D. Griffioen, K. J. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an audio protocol with bus collision using UPPAAL. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 244–256, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[KKZ05] J.-P. Katoen, M. Khattri, and I. S. Zapreev. A Markov reward model checker. In *Quantitative Evaluation of Systems (QEST)*, pages 243–244, 2005.

[KL97] P. Kemper and R. Lubeck. Model checking based on kronecker algebra, 1997.

[KNP00] M. Kwiatkowska, G. Norman, and D. Parker. Verifying randomized distributed algorithms with prism. In *Proc. Workshop on Advances in Verification (Wave'2000)*, July 2000.

[KNP01] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In P. Kemper, editor, *Proc. Tools Session of Aachen 2001 International Multiconference on Measurement, Modelling and Evaluation of Computer-Communication Systems*, pages 7–12, September 2001. Available as Technical Report 760/2001, University of Dortmund.

[KNP02] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. In J.-P. Katoen and P. Stevens, editors, *Proc. 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 52–66. Springer, 2002.

[KNPS06] M. Kwiatkowska, G. Norman, D. Parker, and J. Sproston. Performance analysis of probabilistic timed automata using digital clocks. *Formal Methods in System Design*, 29:33–78, 2006.

[KNS00] Marta Z Kwiatkowska, Gethin Norman, and Jeremy Sproston. Symbolic model checking of probabilistic timed automata using backwards reachability. Technical Report CSR-00-01, University of Birmingham, School of Computer Science, January 2000.

[KNS03a] M. Kwiatkowska, G. Norman, and J. Sproston. Probabilistic model checking of deadline properties in the IEEE 1394 FireWire root contention protocol. *Special Issue of Formal Aspects of Computing*, 14:295–318, 2003.

[KNS03b] M. Kwiatkowska, G. Norman, and J. Sproston. Symbolic model checking for probabilistic timed automata. Technical Report CSR-03-10, School of Computer Science, University of Birmingham, 2003.

[KNSS99] M. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Automatic verification of real-time systems with discrete probability distributions. In J-P. Katoen, editor, *Proc. Proc. Formal Methods for Real-Time and Probabilistic Systems (ARTS'99)*, volume 1601 of *Lecture Notes in Computer Science*, pages 75–95. Springer, 1999.

[KNSS02] M. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Automatic verification of real-time systems with discrete probability distributions. *Theoretical Computer Science*, 282:101–150, 2002.

[KNSW04] M. Kwiatkowska, G. Norman, J. Sproston, and F. Wang. Symbolic model checking for probabilistic timed automata. In *Joint Conference on FORMATS and FTRTFT*, volume 3253 of *LNCS*, pages 293–308. Springer, 2004.

[KRO] KRONOS. Kronos WebSite. http://www-verimag.imag.fr/TEMPORISE/kronos/.

[KSW04] M. Kuntz, M. Siegle, and E. Werner. Symbolic performance and dependability evaluation with the tool CASPA, 2004.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. In *IEEE Transactions on Software Engineering*, volume 3. IEEE, 1977.

[Lee59] C. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999, 1959.

[LLPY97] K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: Compact data structure and state space reduction. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 14–24, 1997.

[LPWY99] Kim G. Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Clock difference diagrams. *Nordic J. of Computing*, 6(3):271–298, 1999.

[LPY97a] Kim Guldstrand Larsen, Paul Petterson, and Wang Yi. UPPAAL: Status developments. In *Computer Aided Verification*, pages 456–459, 1997.

[LPY97b] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[LV93a] Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations – part I: untimed systems. In *135*, page 35. Centrum voor Wiskunde en Informatica (CWI), ISSN 0169-118X, 31 1993.

[LV93b] Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations – part II: timing-based systems. In *145*, page 36. Centrum voor Wiskunde en Informatica (CWI), ISSN 0169-118X, 31 1993.

[MCD00] Andrew S. Miner, Gianfranco Ciardo, and Susanna Donatelli. Using the exact state space of a markov model to compute approximate stationary measures. In *Measurement and Modeling of Computer Systems*, pages 207–216, 2000.

[McM93] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[MHA02] Jesper Møller, Henrik Hulgaard, and Henrik Reif Andersen. Symbolic model checking of timed guarded commands using difference decision diagrams, July–August 2002.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[ML98] J. Møller and Jakob Lichtenberg. Difference decision diagrams. Master's thesis, Department of Information Technology, Technical University of Denmark, Building 344, DK-2800 Lyngby, Denmark, August 1998.

[MLAH99a] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference decision diagrams. In *Proceedings 13th International Workshop on Computer Science Logic*, volume 1683 of *Lecture Notes in Computer Science*, pages 111–125, Madrid, Spain, September 1999.

[MLAH99b] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference decision diagrams. Technical Report IT-TR-1999-023, Department of Information Technology, Technical University of Denmark, Building 344, DK-2800 Lyngby, Denmark, February 1999.

[MLAH99c] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Fully symbolic model checking of timed systems using difference decision diagrams. In *Workshop on Symbolic Model Checking*, volume 23, The IT University of Copenhagen, Denmark, June 1999.

[MLAH99d] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. On the symbolic verification of timed systems. Technical Report IT-TR-1999-024, Department of Information Technology, Technical University of Denmark, Building 344, DK-2800 Lyngby, Denmark, February 1999.

[MP02] Arnaldo V. Moura and Guilherme A. Pinto. A note on the verification of automata specifications of probabilistic real-time systems. *Inf. Process. Lett.*, 82(5):223–228, 2002.

[NSY92] X. Nicollin, J. Sifakis, and S. Yovine. Compiling real-time specifications into extended automata. In *IEEE Transactions on Software Engineering*, volume 18(9) of *LNCS*, pages 794–804. Springer, 1992.

[Par02] D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, 2002.

[Pla85] B. Plateau. On the stochastic structure of parallelism and synchronisation models for distributed algorithms. In *Proc. 1985 ACM SIGMETRICS Conference on Measurement and Modeling of C omputer Systems*, volume 13(2) of *Performance Evaluation Review*, pages 147–153, 1985.

[Pnu77] A. Pnueli. The temporal logic of programs". In *Proc. 18th IEEE Symp. foundations of Computer Science*, pages 46–57, 1977.

[PRI] PRISM. PRISM WebSite. http://www.cs.bham.ac.uk/~dxp/prism/.

[PTA] PTAsource. PTA source code WebSite. http://www.cs.bham.ac.uk/~fxw.

[QS82] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proc. 5th Int. Symp. in Programming*. Springer, 1982.

[RDN90] F.W. Vaandrager R. De Nicola. Actions versus state based logics for transition systems. In *Proc. Ecole de Printemps on Semantics of Concurrency*, volume 469 of *LNCS*, pages 407–419. Springer, 1990.

[SB03] Sanjit A. Seshia and Randal E. Bryant. A boolean approach to unbounded, fully symbolic model checking of timed automata. Technical Report CMU-CS-03-117, Carnegie Mellon University, 2003.

[SV99] M. Stoelinga and F. Vaandrager. Root contention in ieee 1394. In *Proc. 5th AMAST Workshop on Real-Time and Probabilistic Systems (ARTS'99)*, volume 1601 of *LNCS*, pages 53–74. Springer, 1999.

[Tri98] Stavros Tripakis. *The analysis of timed systems in practice*. PhD thesis, University Joseph Fourier, Grenoble, France, December 1998.

[UPP] UPPAAL. UPPAAL WebSite. http://www.docs.uu.se/docs/rtmv/uppaal/.

[Var95] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Banff Higher Order Workshop*, pages 238–266, 1995.

[Wan03] Farn Wang. Efficient verification of timed automata with BDD-like data-structures. In *Verification, model checking, and abstract interpretation*, volume 2575 of *LNCS*, pages 189–205. Springer, 2003.

[WK05] F. Wang and M. Kwiatkowska. An MTBDD-based implementation of forward reachability for probabilistic timed automata. In D. Peled and Y.-K. Tsay, editors, *Proc. 3rd International Symposium on Automated Technology for Verification and Analysis (ATVA'05)*, volume 3707 of *LNCS*, pages 385–399. Springer, 2005.

[WVF95] Jeannette M. Wing and Mandana Vaziri-Farahani. Model Checking Software Systems: A Case Study. In *Proceedings of SIGSOFT'95 Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 128–139, 1995.

[WW96] Bernard Willems and Pierre Wolper. Partial-order methods for model checking: From linear time to branching time. In *Logic in Computer Science*, pages 294–303, 1996.

[YPD94] Wang Yi, Paul Pettersson, and Mats Daniels. Automatic verification of real-time communicating systems by constraint-solving. In *FORTE*, pages 243–258, 1994.