VERIFYING WORST-CASE EXECUTION TIME OF TIMED AUTOMATA MODELS WITH CYCLIC BEHAVIOUR

THIS THESIS IS PRESENTED TO THE
SCHOOL OF COMPUTER SCIENCE & SOFTWARE ENGINEERING AND
SCHOOL OF ELECTRICAL, ELECTRONIC AND COMPUTER ENGINEERING
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
OF
THE UNIVERSITY OF WESTERN AUSTRALIA

By
Omar Ibrahim Al-Bataineh
July 2015
VERIFYING WORST-CASE EXECUTION TIME OF TIMED AUTOMATA MODELS WITH CYCLIC BEHAVIOUR

Omar Ibrahim Al-Bataineh
Abstract

The thesis presents novel zone-based algorithms for computing worst case execution time (WCET) or maximum termination time of real-time systems using the timed automata (TA) model checking technology. The algorithms can work on any arbitrary diagonal-free TA \(^1\) and can handle more cases than previously existing algorithms for WCET computations, as it can handle cycles in TA and decide whether they lead to an infinite WCET. We show soundness of the proposed algorithms and study their complexity. The solutions provided here are conceptually a marked improvement over some earlier work on the problem, in which repeated guesses (guided by binary search) and multiple model checking queries were effectively but inelegantly and less efficiently used; here only one run of the zone construction is sufficient to yield the answers.

The thesis also proposes a set of acceleration techniques that improve the efficiency of WCET verification of TA with cyclic behaviour. We prove that the proposed accelerations are exact with respect to the WCET problem and demonstrate that model checking WCET with the proposed acceleration techniques can significantly speed-up the verification of WCET of real-time systems. We also compare our algorithm with the one implemented in the model checker UPPAAL which shows that the proposed algorithms can handle cases that UPPAAL fails to verify, where we show that in certain circumstances, when infinite cycles exist, UPPAAL’s algorithm may not terminate, and when largely repetitive finite cycles exist, UPPAAL’s algorithm suffers from the state space explosion, thus leading to a low efficiency or resource exhaustion.

The thesis presents also a set of new operations for improving the reachability analysis and the WCET analysis of Timed Automata (TA) using the Difference Bound

\(^1\)The diagonal-free TA is a class of TA in which the test of the form \(x - y \sim c\) is disallowed, where \(x, y\) are clocks, \(c\) is a constant, and \(\sim \in \{<, \leq, =, >, \geq\}\).
Matrices (DBMs), namely the partial canonicalization and the partial extrapolation of DBMs. The partial canonicalization allows one to fix the non-tightness introduced by extrapolation by updating only the clock constraints that have been changed during extrapolation, thus reducing the impact on the run time of the reachability algorithm of the canonicalization step. The proposed partial canonicalization algorithms are specializations of Floyd’s algorithm that have a time complexity of $O(c \times n)$ instead of $O(n^3)$ for the standard approach, where $n$ is the number of clocks in the automaton and $c$ is the number of variables that have been changed during extrapolation. We demonstrate that model checking TA with partial canonicalization can speed-up considerably the verification time of several interesting examples including the Philips audio control protocol and Fischer’s protocol. On the other hand, the partial extrapolation helps to perform a more precise analysis of the reachable states of the timed automaton and it is necessary for several applications of TA including the problem of computing minimum and maximum termination times.

The thesis also reports some lack of precision in previously published algorithmic of zones (sets of clock valuations) and difference bound matrices (a data structure to represent and handle zones). In fact, these algorithms are rarely discussed in details. In particular the extrapolation, canonicalization, and inclusion checking operations and their role in forward reachability algorithm with respect to certain problems such as the minimum termination time problem and the maximum termination time problem require extra care and non-trivial arguments for proving both correctness and termination. With this in mind, any generalization raises challenging questions and we believe that the partial canonicalization and the partial extrapolation of difference bound matrices proposed in this thesis are of great importance.
Acknowledgements

First and foremost, I thank and praise Allah (God) for the uncountable bounties He has bestowed upon me, among which is giving me the ability and the strength to continue my graduate studies and to complete this thesis. Second, a very special thanks due to Mark Reynolds (my principal supervisor) and Tim French (my co-supervisor). Mark has also covered me generously through his research grants during my trips to UK (TIME conference 2012 [BFW12]) and Italy (FORMATS conference 2014 [BRF14]). Special thanks due also to Terry Woodings, my co-supervisor, for his support, guidance and helpful feedback throughout my thesis. Throughout my PhD, we explored many aspects of timed automata; they have been supportive, and always available for discussions and advice. They encouraged me in pursuing my curiosities, and it has been a great experience working with them. I would also like to acknowledge the great financial support I received through ARC discovery project grant DP110104669.

This thesis is dedicated to my family for their endless love and support during my work on this thesis.

Omar Al-Bataineh, 2015
## Contents

1 Introduction

1.1 The Problem ................................................. 1
1.2 The Approach .............................................. 3
1.3 Original Contributions ................................. 5
1.4 Thesis Overview ..................................... 7

2 Background

2.1 Introduction ............................................. 10
2.2 The History of Real-time Model Checking Technology ....... 12
2.3 Preliminaries ............................................. 15
   2.3.1 The Need for Formal Methods ................ 15
   2.3.2 The Process of Model Checking .......... 17
   2.3.3 Basic Definitions, Terminology, and Concepts .... 21
2.4 Formal Analysis of Timed Systems .................. 26
   2.4.1 Semantic Model: Dense Time .......... 26
   2.4.2 System Modelling Language: Timed Automata Model .. 28
2.5 Specification Languages: Real-time Temporal Logic ...... 32
   2.5.1 Timed Computation Branching Temporal Logic .... 35
CONTENTS

2.6 State Space Abstraction Techniques ........................................ 38
  2.6.1 The Region Abstraction .............................................. 39
  2.6.2 The Zone Abstraction ................................................. 42
2.7 Representation of State Space ............................................. 44
  2.7.1 Difference Bound Matrices .......................................... 44
  2.7.2 Binary Decision Diagrams ........................................... 47
  2.7.3 Clock Restriction Diagrams ......................................... 48
  2.7.4 Clock Difference Diagrams .......................................... 48
2.8 Techniques for Constructing Reachability Graph of Systems ............ 49
  2.8.1 Backward Versus Forward Analysis .................................. 50
  2.8.2 On-the-Fly Approach ................................................ 50
  2.8.3 Compositional Approach ............................................. 51
2.9 Basic Operations on Zones .................................................. 52
2.10 Model Checking Tools for Real-time Systems ........................... 58
  2.10.1 HyTech ................................................................. 58
  2.10.2 Kronos ................................................................. 59
  2.10.3 UPPAAL ................................................................. 60
  2.10.4 RED .................................................................... 61
  2.10.5 Rabbit ................................................................. 62
  2.10.6 PRISM ................................................................. 64
2.11 Conclusion ................................................................. 64

3 The WCET Problem of Timed Automata .................................. 67
  3.1 Networks of Timed Automata ............................................ 67
  3.2 The WCET of Timed Automata ........................................... 69
### CONTENTS

3.2.1 Cycles in Timed Automata .......................... 69  
3.2.2 Some Challenges in Verifying WCET of TA with Cyclic Behaviour .......................... 75  
3.3 Related Work ............................................. 77  
3.3.1 Methods and Tools for the WCET Problem .......................... 77  
3.3.2 Model Checking and WCET Problem .......................... 78  
3.3.3 Model Checking and Acceleration Techniques .......................... 80  

4 WCET of Acyclic and Finite Cyclic Timed Automata 83  
4.1 Introduction ............................................. 84  
4.2 Approaches for Verifying WCET of TA .......................... 85  
4.2.1 The Binary Search Approach .......................... 85  
4.2.2 Some Limitations of The Binary Search Approach .......................... 87  
4.2.3 A modified Version of The Zone Approach .......................... 88  
4.3 Implementation ............................................. 94  
4.4 Case Studies ............................................. 95  
4.5 Conclusion ............................................. 99  

5 WCET of Arbitrary Timed Automata 100  
5.1 Introduction ............................................. 101  
5.2 Behrmann et al Minimum Cost Reachability Algorithm .......................... 102  
5.3 Computing WCET of Cyclic Real-time Systems .......................... 108  
5.3.1 Solving The Problem Using Partial Extrapolation .......................... 108  
5.4 A Zone-based Algorithm for Computing WCET of TA .......................... 125  
5.5 Implementation ............................................. 128
CONTENTS

5.6 Conclusion ......................................................... 129

6 Partial Canonicalization of Clock Zones ............... 130

   6.1 Introduction .................................................. 131

   6.2 Extrapolation Algorithms for Timed Automata .............. 134

      6.2.1 Classical Maximal Bounds. ................................. 135

      6.2.2 Lower and Upper Maximal Bounds. ......................... 136

   6.3 Partial Canonicalization of Difference Bound Matrices .... 141

   6.4 Checking Inclusion between Zones ........................... 151

   6.5 Implementation and Experiment ............................... 156

      6.5.1 The Fischer’s Protocol ................................. 156

      6.5.2 The Philips Audio Control Protocol ................... 157

   6.6 Conclusion ................................................... 158

7 Accelerating WCET Analysis of TA Models ........... 160

   7.1 Introduction .................................................. 161

   7.2 Abstractions for TA ........................................... 163

      7.2.1 The Extrapolation Abstraction ......................... 163

      7.2.2 The Activity Abstraction ............................... 164

      7.2.3 The Fixed Point Abstraction .......................... 165

   7.3 Accelerating WCET Calculations of Finite Cycles  ............ 170

      7.3.1 Finite Cycles with Simple Structure .................. 171

      7.3.2 Finite Cycles with Complex Structure ................. 183

   7.4 An Algorithm for Computing WCET with Accelerations ......... 188

   7.5 Implementation ............................................... 192
CONTENTS

7.6 Conclusions .......................................................... 193

8 Conclusions ................................................................ 196

8.1 Summary ................................................................. 196

8.2 Future Work ............................................................. 198

References ..................................................................... 200

Index ............................................................................. 217
List of Figures

1. $A_1$: an automaton with finite cycle ................................. 4
2. $A_2$: an automaton with infinite cycle ................................. 4
3. The Lamp template .......................................................... 32
4. The User template ........................................................... 32
5. An example of a timed automaton by which we demonstrate how one can compute the zone graph of an automaton. ......................... 57
6. A TA with a WCET = 1 ...................................................... 73
7. A TA with an infinite WCET ............................................... 73
8. The D-Robot template ....................................................... 96
9. The G-Robot template ....................................................... 96
10. The Box template ........................................................... 97
11. The processing station template ......................................... 97
12. An automaton with BCET = 20 and WCET = $\infty$ .................. 106
13. A simple infinite automaton with BCET = 1 and WCET = $\infty$ .. 106
14. An automaton with a BCET = 11 and WCET = $\infty$ ............... 106
15. An automaton where BCET = 0 and WCET = $\infty$ ............... 106
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>An automaton with three infinite cycles</td>
<td>141</td>
</tr>
<tr>
<td>17</td>
<td>An example by which we demonstrate the usefulness of the activity abstraction</td>
<td>165</td>
</tr>
<tr>
<td>18</td>
<td>Three finite dependent cycles that lead to infinite WCET</td>
<td>167</td>
</tr>
<tr>
<td>19</td>
<td>Three self-cycles that lead to WCET of 20</td>
<td>167</td>
</tr>
<tr>
<td>20</td>
<td>A largely repetitive finite cycle</td>
<td>171</td>
</tr>
<tr>
<td>21</td>
<td>An automaton containing a cycle where the lower delay of the cycle is zero and the upper delay is one</td>
<td>174</td>
</tr>
<tr>
<td>22</td>
<td>Timed automaton containing a finite cycle with constant delays</td>
<td>177</td>
</tr>
<tr>
<td>23</td>
<td>A TA with infinite realizable cycle</td>
<td>193</td>
</tr>
<tr>
<td>24</td>
<td>Three intersecting finite cycles that lead to an infinite WCET</td>
<td>193</td>
</tr>
<tr>
<td>25</td>
<td>A TA with largely repetitive finite cycle</td>
<td>194</td>
</tr>
<tr>
<td>26</td>
<td>A TA containing a composite cycle</td>
<td>194</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 The Problem

Real-time systems are systems that are designed to run applications and programs with very precise timing and a high degree of reliability. These systems are said to be failed if they can not guarantee response within strict time constraints. The success of a real-time system depends on whether all the scheduled tasks can be guaranteed to complete their executions before their deadlines. Usually, best-case execution time (BCET) and worst-case execution time (WCET) are used for schedulability analysis of real-time systems. In the last few years, there has been a considerable interest in using formal methods and, in particular (timed automata) model checking [CGP01] for computing BCET and WCET since it gives precise answers to these questions in an automatic way [Met04, BLR05a].

Typically, the infinite state-space of a timed transition system is converted into an equivalent finite state-space of a symbolic transition system called a zone graph [Dil90, CGP01]. In a zone graph, zones (i.e. sets of valuations of the timed automaton clocks) are used to denote symbolic states. The zone graph has been successfully used for the verification of safety and liveness properties of timed automata. Although the zone graph is precise enough to preserve the reachability properties in
TA, it is too abstract to infer continuous time progress. At each step of the successor computation, the generated zones are extrapolated (abstracted) using a set of extrapolation operators and then canonicalized (tightened) in order to obtain a unique representation of the resulting zones. A test for inclusion of zones is then applied to check whether the new generated zone at a particular control location in the graph is already covered by some previously generated zones associated with that location. This helps to ensure termination of the analysis of TA even when infinite cycles exist.

The thesis considers the problem of computing the “worst case execution time” (WCET) in timed automata. *Given a timed automaton $A$ with a start location $l_s$ and a final location $l_f$, this problem asks to compute an upper bound on the time needed to reach the final location $l_f$ from the start location $l_s$. The worst-case execution-time of $A$ denoted as $WCET(A)$ is the supremum on the accumulated delays of all runs $R$ of $A$. In general, WCET analysis is undecidable: it is undecidable to determine whether or not an execution of a program will eventually halt. However, for TA models one can use model-checking techniques to analyse the system and compute the WCET.*

The WCET problem is easy to solve in the case of acyclic TA (i.e. TA contains no cycles), but cycles might introduce an unbounded WCET, that needs to be detected on-the-fly during the analysis. Also, verifying WCET of systems with cyclic behaviour can cause the state explosion problem if we traverse all the cycle’s iterations during the analysis in particular when there is a largely repetitive finite cycle in the behaviour of the analysed system. Although symbolic methods (e.g., using BDDs or CRDs) can be applied to analyse programs and systems with a huge number of states, they may fail to compute the exact WCET when cycles exist. Furthermore, as we know, the efficiency of TA model checking technology relies mainly on the application of a combination of abstraction techniques such as the inclusion abstraction [DT98], the activity abstraction [DT98], and the extrapolation abstraction [DT98], which help to accelerate the reachability analysis.
1.2. **THE APPROACH**

of TA. However, these abstractions can harm the WCET analysis (i.e. they might introduce coarse over-approximation or coarse under-approximation on the value of WCET) if not applied carefully. For example, the inclusion abstraction can lead to stopping the exploration when some zone is “included” inside an already explored zone. This way of cutting exploration of state space based on inclusion is correct for reachability properties, but not when properties critically involve detection of cycles in the timed automaton. On the other hand, the extrapolation abstraction replaces set of states by larger ones that are equivalent w.r.t reachability properties. Such replacement is correct for reachability properties but not for WCET analysis. To demonstrate the problem, we give in Figures 1 and 2 two automata where both generate identical zone graphs when applying the standard zone approach for reachability analysis. The automaton $\mathcal{A}_1$ represents an automaton with finite cycle where $WCET(\mathcal{A}_1) = 12$. For this automaton, the standard zone approach computes correctly the WCET without involving any extra check. On the other hand, the automaton $\mathcal{A}_2$ represents an automaton with an infinite cycle where $WCET(\mathcal{A}_2) = \infty$. For this automaton, the standard zone approach fails to give the correct answer for WCET since it returns 12 instead of $\infty$. Note that if we disable extrapolation during the construction of the zone graph, the search may not stop and we may not be able to obtain an answer.

### 1.2 The Approach

The solutions proposed here consist in modifying the classical zone-based forward approach for reachability in TAs. Obviously, adding an extra clock $\delta$ that is never reset and never occurs in the guards and invariants of the TA, and computing the zone graph (with special modified extrapolation and canonicalization procedures) of the TA allows to obtain the WCET, by inspecting the elements of the zone graph of the form $(l_f, Z)$, where $l_f$ is a final location in the automaton, and by computing an upper bound on the clock valuations that satisfy all those zones $Z$. Unfortunately,
without the extrapolation operator, the algorithms might not terminate in general because the zone graph can be infinite. Also, using extrapolation might introduce coarse over-approximations on the valuations of $\delta$, in case it exceeds the maximal extrapolation constant, for instance.

The thesis proposes two approaches to solve the WCET problem which are mainly a combination of (i) new extrapolation and canonicalization procedures that keep the valuations of the extra clock $\delta$ precise, and (ii) an on-the-fly analysis of the possible cycles in the TA. The proposed algorithms guarantee termination and compute precisely the WCET and can detect the cases where the WCET is infinity. Thus, the proposed algorithms can be a significant break-through in computing WCET.

The first approach follows a new paradigm where zones are not abstracted, hence preserving the value of the extra clock. Instead, abstraction is used to detect when a zone has already been explored, that is, when two non-abstracted zones have the same abstraction. The second approach uses what we call partial extrapolation
that keeps the value of the extra clock precise while abstracting the values of the automaton clocks. The algorithm avoids loss of precision in WCET calculations that may happen due to extrapolating zones inside cycles since it checks for exact fixed points rather than inclusion between zones and it uses partial extrapolation rather than full extrapolation. The algorithms use also a set of acceleration techniques that improve significantly the efficiency of WCET verification of cyclic real-time systems in particular when infinite cycles and largely repetitive cycles exist. We prove that the proposed accelerations are exact with respect to the WCET problem.

We compare also our algorithms with this implemented in UPPAAL which shows that our algorithm can outperform UPPAAL’s algorithm by many orders of magnitude and it can handle largely repetitive cycles efficiently. On the other hand, when verifying TA with infinite cycles or largely repetitive cycles in UPPAAL, we find that UPPAAL’s algorithm suffers from the state space explosion, thus leading to a low efficiency or resource exhaustion.

1.3 Original Contributions

- The thesis presents novel algorithmic solutions to the problems of computing the shortest and the longest time taken by a run of a timed automaton from an initial state to a final state. The solutions are conceptually a marked improvement over some earlier work on the problems [Met04, DOT+10], in which repeated guesses (guided by binary search) and multiple model checking queries were effectively but inelegantly and less efficiently used; while in our algorithms only one run of the zone construction is sufficient to yield the answers. We first limit applicability of our approach to timed automata without infinite runs and then extend the algorithms to work on any arbitrary diagonal-free TA.

- The thesis presents a set of new operations for improving the reachability analysis of Timed Automata (TA) using the Difference Bound Matrices (DBMs),
CHAPTER 1. INTRODUCTION

namely the partial canonicalization and the partial extrapolation of DBMs. The partial canonicalization allows one to fix the non-tightness introduced by extrapolation by updating only the clock constraints that have been changed during extrapolation, thus reducing the impact on the run time of the reachability algorithm of the canonicalization step. The proposed partial canonicalization algorithms are specializations of Floyd’s algorithm that have a time complexity of $O(c \times n)$ instead of $O(n^3)$ for the standard approach, where $n$ is the number of clocks in the automaton and $c$ is the number of variables that have been changed during extrapolation. We demonstrate that model checking TA with partial canonicalization can speed-up considerably the verification time of several interesting examples including the Philips audio control protocol and Fischer’s protocol. The partial extrapolation helps to perform a more precise analysis of the reachable states of the timed automaton and it is necessary for several application of TA including the problem of computing minimum and maximum termination times.

- We discuss also some lack of precision in previously published algorithms for timed automata, where surprisingly we discover that reducing an extrapolated matrix to canonical form can give a non-extrapolated matrix. This may cause problems when checking inclusion between zones. According to [Pet99, BBFL03, BY04, Bou04, BLR05b, BBLR06] the test for inclusion of zones should be checked syntactically on the canonical form of DBMs and hence the extrapolated zones need to be canonicalized before checking inclusion between zones. However, we show that checking inclusion between zones using the non-canonical extrapolated zones is a safe operation since termination using this form of test is guaranteed, while on the other hand, checking inclusion between zones using the canonical extrapolated zones is unsafe operation since termination may not be guaranteed. This may look surprising to the reader. This is mainly due to the fact that canonizing extrapolated zones can give non-extrapolated zones which can cause problems during inclusion.
checking and hence it may affect adversely termination of the analysis.

- The thesis proposes also a set of acceleration techniques that improve the efficiency of WCET verification of real-time systems that contain largely repetitive cycles (loops). We prove that the proposed acceleration techniques are exact with respect to the WCET problem. Moreover, we demonstrate that model checking WCET with these techniques can significantly speed-up the verification of WCET of several interesting applications.

## 1.4 Thesis Overview

The thesis contents are organised as follows:

- Chapter 2 reviews some basic concepts, definitions, and terms that are commonly used in the literature of (real-time) model checking. It gives an overview of temporal logic and some of its well-known real-time extensions, in particular we focus on TCTL, TPTL, and MTL logics. It then discusses some commonly accepted models (formalisms) for real-time systems in particular the timed automata model, and some well-known state space construction methods, reduction techniques, and some practical algorithms and techniques for model checking real-time systems and finally it discusses some state-of-the-art model checking tools.

- Chapter 3 gives a formal definition of the WCET problem in TA and a formal definition of cycles (loops) in TA. It then discusses the different types of cycles in TA and the cases where the WCET of an automaton can be infinity. It also discusses related work in model checking Worst-case Execution Time (WCET), some acceleration techniques, and other approaches and algorithms for model checking TA.

- Chapter 4 proposes a model checking solution to the problem of computing the minimum and maximum termination time of TA using a new paradigm
where zones are not abstracted. Instead, abstraction is used to detect when a zone has already been explored, that is, when two non-abstracted zones have the same abstraction. The applicability of the proposed solution in this chapter is limited to TA without infinite runs.

- Chapter 5 reviews the existing extrapolation procedures of TA and discusses their role in forward reachability algorithms. Then it discusses some interesting issues about the minimum cost reachability algorithm proposed by Behrmann et al [BFH+01] and its implementation in UPPAAL. Then it introduces what we call partial extrapolation procedure of zones and prove its correctness. It discusses what we call fixed point abstraction which can be used to detect (on-the-fly) infinite cycles. A description of a model checking algorithm for computing WCET of a general class of diagonal-free TA is presented. An implementation of the algorithm using the model checker opaal and a description of the associated verification results on a set of examples are then discussed.

- Chapter 6 introduces the partial canonicalization of zones that allows one to fix the non-tightness introduced by extrapolation by canonizing only the constraints that have been changed during extrapolation. It then describes a specialisation of Floyd’s algorithm for the $M$-extrapolation and the $LU$-extrapolation procedures and another specialisation of Floyd’s algorithm for the $M^+$-extrapolation and the $LU^+$-extrapolation procedures. A description of a new form of inclusion checking between zones is introduced and its correctness is proved. An implementation of the reachability algorithm using the new canonicalization procedures and the associated verification results on a set of examples are presented. Finally, we draw some conclusions and discuss future directions.

- Chapter 7 proposes a set of acceleration techniques that improve the efficiency of WCET verification and proves that the proposed acceleration techniques
are exact with respect to the WCET problem. It demonstrates that model checking WCET with these techniques can significantly speed-up the verification of WCET of several interesting applications.

- Chapter 8 concludes and discusses some directions for future research.
Chapter 2

Background

This chapter provides the reader with necessary background information. We review some basic concepts, definitions, and terms that are commonly used in the literature of (real-time) model checking. We give an overview of temporal logic and some of its well-known real-time extensions, in particular we focus on real-time of the branching temporal logic (TCTL). We then discuss some commonly accepted models (formalisms) for real-time systems in particular the timed automata model, then we discuss some well-known state space construction methods, reduction techniques, and some practical algorithms and techniques for model checking real-time systems and finally we discuss some state-of-the-art model checking tools.

2.1 Introduction

Real-time systems are systems that are designed to run applications and programs with very precise timing and a high degree of reliability. These systems can be said to be failed if they can not guarantee response within strict time constraints. Real time systems have many safety critical applications such as computer-controlled medical devices, air traffic control systems, airbag system in cars, and real-time database systems. Failure of these systems can
have catastrophic consequences. Consider for example designing an airbag system for a car. In this case, a small error in designing timing constraints (causing the airbag to deploy too early or too late) could be catastrophic and cause injury. Real-time systems are amongst the most challenging systems to analyse. This is mainly because the correctness of real-time systems depends on the actual times at which events occur. Moreover, these systems involve interactions of a number of concurrent components that have a high level of complexity. Such interactions might lead to many subtle or undesirable situations if they are not considered carefully. Hence, real-time systems need to be rigorously modeled and verified in order to have confidence in their correctness with respect to the desired properties. One of the well-known formal verification techniques that can be used to verify correctness of systems is the model checking technique [CGP01].

In model checking, we verify whether a system given in a formal mathematical model satisfies a certain specification written as a logical formula. Model checking has been successfully used to find non-trivial errors in hardware designs, distributed systems, and security protocols. Model checking uses a variety of sophisticated heuristics and symbolic implementation techniques to check that a logical formula holds in the given system design. Due to the state space explosion problem [CGP01] only restricted forms of models and formulas can be shown to be decidable.

In the last two decades, researchers have proposed several different timed extensions of temporal logics that can be used for specifying and verifying real-time systems. Examples include real-time of the branching temporal logic (TCTL) [ACD90], the Metric temporal logic (MTL) [Koy90], the timed propositional temporal logic (TPTL) [ACD90], and their fragments. Many achievements in the formal verification of real-time systems have been reported, from various solid theory foundations to complex data structures and implementation techniques. Several tools have been also developed for the
purpose of model-checking real-time systems such as Hytech [HHWt97], UP-PAAL [BDL04], KRONOS [DOTY96], RED [Wan04c], and Rabbit [BLN03]. In this chapter, we give a review of these many achievements so that readers can use the chapter as an index to the literature.

2.2 The History of Real-time Model Checking Technology

Several formalisms have been proposed for modeling and describing real-time systems including timed automata model [AD94], timed I/O model [KLSV03], timed process algebras [Wan02], and timed Perti networks [AN01]. However, the timed automata model of Alur and Dill has become the standard. Several timed extensions of temporal logic have been proposed for specifying real-time systems. Examples include TCTL, MTL, TPTL, and RTL.

The first proposal for specifying real time requirements is found in [BH81], in which Bernstein and Harter extend Propositional Temporal Logic (PTL) [CGP01] via introducing operators such as $\phi \leq_n \psi$, which means that “every $\phi$-state is followed by $\psi$-state within $n$ time units”. Later Koymans [KVdR83, Koy90] proposes metric temporal logic (MTL) as a specification language for real-time systems. In [Lew90], Lewis considers a real time branching time logic. The syntax is an extension of CTL [CES86] with interval subscripts on temporal operators. In [AH94], Alur and Henzinger show that the dense time semantics leads to undecidability of the satisfiability problem in the presence of operators such as the future operator $F_{=3}$. A large body of work exists on studying decidability and expressiveness questions of several timed extensions of temporal logics [AH90, AH92, AH94]. In particular, MTL has been studied extensively for the purposes of verification. In [AFH96] Alur et al show that the model checking problem of MTL is undecidable when considering the
interval-based semantics. However, in [AH90] they propose a decidable subset of the logic MITL which disallows singular interval subscripts. Recently, it has been shown that MTL model checking and satisfiability are decidable over finite words under the pointwise semantics [OW05]. Several decidable fragments of MTL were identified by syntactically or semantically restricting the MTL language [AFH96, HRS98, HR04, BMOW08, OW06], while keeping logic expressive enough to specify timing properties. Some interesting fragments of MTL are Safety-MTL and CoFlat-MTL, that have been shown to be decidable with a reasonable model checking complexity. In the literature, there is a debate of which view of temporal logic (i.e. the linear or the branching view) is more appropriate for expressing timed properties. Among other researchers, Wolper [Wol81], Pnueli [Pnu85], Rosner and Pnueli [RP86], and Vardi [Var01] have recognised that linear-time requirements languages are preferred over branching-time ones, but they show that both of them can not express certain properties. However, each of them has some certain limitations that hamper the representations of some interesting properties. We agree with Tripakis in [Tri98] that both views are necessary and they complement each other.

The literature of timed automata theory is a rich literature since it was introduced in 1990. In [ACD90] Alur et al showed that the model checking problem for TCTL over TA is PSPACE-complete and gave a model checking algorithm of TCTL. In [HNSY92] Henzinger et al proposed another algorithm for TCTL based on symbolic techniques and works sometimes in practice. Alur and Madhusudan [AM04] present a full survey of known results for decidability problems in timed automata theory. Ober et al [OGO04] proposed a timed unified modeling language (UML) for real-time systems and showed how to translate timed UML into timed automata that can be used for formal analysis. Tripakis [Tri98] gives algorithms and techniques to verify timed
systems using TCTL logic and Timed Buchi Automata, which have been implemented in KRONOS model checking tool. KRONOS is a full DBM-based model checker that supports both forward and backward TCTL model checking. In [BTY97] Bouajjani et al gives an on-the-fly model checking algorithm for TECTL* which is shown to be more expressive than TCTL. They present a number of experiments which show that their on-the-fly algorithm outperforms the fixpoint-based algorithm implemented in Kronos. Lamport [Lam05] claims that most real-time properties can be verified using existing languages and tools. He suggests to specify timing properties using TLA+ (Temporal logic of Actions) which is a high level mathematical language. He proposes to present time as an ordinary variable (now) which is incremented using an action (Tick), and then specify timing properties using a special timing variable.

Because the state space of timed automata is infinite, a straightforward model checking of timed automata models is impossible. Researchers therefore proposed several abstraction approaches by which an infinite timed transition system can be converted into an equivalent finite symbolic transition. The region [AD94] and zones [Dil90, HNSY92] approaches are examples of such abstraction approaches. However, in practice, the region approach has not been implemented in any of the available tools since it suffers from a combinatorics explosion. In fact, an on-the-fly forward reachability algorithm using zones has been implemented in tools like Kronos and UPPAAL.

To represent zones efficiently, researcher have proposed several data structures such as Difference Bound Matrices (DBMs) [Dil90], Binary Decision Diagrams (BDDs) [Bry86, BCM+90], and Clock Difference Diagrams (CDDs) [LPWY99]. However, the most commonly used one is DBMs which has been implemented in various tools such as UPPAAL and Kronos. However, in the last few years, BDD-like data structures have been used in the verification of timed systems. The model checkers Rabbit and RED have been developed
based on BDD-like technology. The efficiency and performance of these data structures have been compared by several researchers. Empirical results given in [Wan04a] and [BLN03] have shown that RED and Rabbit outperformed UPPAAL in some particular examples such as Fisher mutual exclusion and FDDI Token Ring protocol. However, the empirical results presented in these works were reported using an old version of UPPAAL (v3.2.4), which lack many of the optimisations that are used in the current version of the tool (v4.1.19).

In [BN03] Beyer shows that the size of the BDD and the CRD representation of the reachability set depends on two properties of the models: the number of automata and the magnitude of the clock values. In [Wan04a] Wang shows that CRDs outperform DBMs when verifying specifications that contain large number of clocks. However, he pointed out that CRDs consume much space (memory) in handling intermediate data structures, and therefore require intensive use of garbage collection.

2.3 Preliminaries

This section provides the reader with necessary background information. We introduce and discuss the concepts that are necessary for proper understanding of the thesis.

2.3.1 The Need for Formal Methods

Modern computer systems are required to operate in complex dynamic environments correctly handling safety-critical or sensitive functions for government and industry. Typical applications include air-traffic control systems, power supply networks, and finance systems. To ensure robust, computationally predictable behaviour of these systems, their designs and implementations
must have a formal basis. This is an important but difficult challenge. Today, complex systems often involve interaction between many components, including hardware, analogue devices and human users. If we look at a hardware design project, for example at a chip design project, we find that designers are often spending around 75% of their effort on just verifying the correctness of the design by doing intensive testing and simulations of the chip in order to ensure that the design is free from any bugs. As we know, when we fail to catch the errors early in the system development process, the cost of fixing the bugs after that can be very large. In general, the earlier an error is discovered, the cheaper it is to fix. Also, we have seen that in software projects, a very small error in the code could result in a security vulnerability that can have a very large economic cost. However, for real-time systems, where the correctness of the system depends on the actual time at which events occur, the problem becomes much more complicated. Failure of real-time systems can have catastrophic consequences since these systems are used in many safety critical applications such as medical device controllers, air traffic controllers, and real-time database transactions. So there is a big motivation to do formal verification.

Developers very often use informal techniques such as testing and simulation techniques to check whether the given system meets its intended properties. Although these techniques are provably effective in the very early stages of debugging, their effectiveness drops dramatically as the complexity of the system design increases; since one can not cover all possible behaviours of the system using these techniques. So it turns out that when we buy software or a system, we get no guarantee that the system is indeed free from any error. So naturally we might imagine that the way to attack the problem would be to apply logical proofs in order to help us to design correct systems. However, constructing proofs can be an overwhelming task. The proof can be even substantially more complex than the system itself. So it has been
argued that we need automation to verify systems. That brings to us formal automated verification techniques such as model checking technique which is an automated technique that helps us to do proofs about systems.

2.3.2 The Process of Model Checking

Model checking is an automated method for verifying concurrent systems. The technique was first developed to verify systems with finite state space. In the last few years, however, the technique has been used to handle systems with infinite state space due to the development of finite abstractions that preserve properties of interest. It uses an intelligent exhaustive search algorithm to check that a logical formula holds in the system design. Typically, formulas are expressed using logics such as temporal logic, belief logic, or epistemic logic, where as the system will be expressed as a set of states (finite number of states) and a set of transitions. When the system fails to meet a desired property, the model checker produces a counterexample that helps to identify the source of the error in the system design.

Applying model checking to prove the correctness of a system design consists of several formal independent steps, which are:

1. **Modeling**: The first step in this process is to convert the system under analysis into formalism accepted by the model checking tool. Due to limitations on memory and time, we usually model a simplified (abstract) version of the system (i.e. we eliminate the details that are irrelevant to the properties that we aim to verify).

2. **Specification**: After we state the properties that we intend to check, it is necessary to present them using some logical formalism. For real-time systems, it is common to use dense-time temporal logic to express desired properties such as bounded liveness properties.
3. **Verification**: At this stage, the model checker tool will determine whether the model of the analysed system satisfies the given specifications. The time that this process takes before showing verification results depends on the size of the model state space, in addition to the number and length of the formulas being verified.

We can formalise the model checking problem as follows: given a finite model $\mathcal{M}$ for the system being analysed, with an initial state $s_0$, and a desired property, expressed as a logical formula $\phi$, check whether the model $\mathcal{M}$ satisfies the property $\phi$ written as $\mathcal{M}, s_0 \models \phi$. The process is completely automatic, and in case that $\mathcal{M}$ does not satisfy $\phi$, a counterexample will be produced to help verification engineers trace the source of the error. When we deal with the problem of model checking with respect to a certain logical language $\mathcal{L}$, we are often interested in two parameters that strongly influence the complexity of the model checking problem: structures; and formulas. Let us explain how each parameter influences the complexity of the model checking problem. We mean by structures the state transition graphs that are used to capture or model the behaviour of systems. Examples of such structures (models) are Kripke structure, Petri net, I/O automata, finite automata, and timed automata. We describe here briefly the Kripke structure as an example on these structures since it is the most popular structure used in model checking.

**Definition 2.3.1.** Let $AP$ be a set of atomic propositions. A Kripke structure $\mathcal{M}$ over $AP$ is a four tuple $\mathcal{M} = (S, S_0, R, L)$ where

- $S$ is a finite set of states.
- $S_0 \subseteq S$ is the set of initial states.
- $R \subseteq S \times S$ is a transition relation between states.
- $\mu : S \to 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.
The model checking problem is to decide whether a Kripke structure $\mathcal{M}$ satisfies a given property $\phi \in \mathcal{L}$. The model checking algorithm first constructs a tableau $\mathcal{M}_{\neg \phi}$ which generates precisely the models of $\neg \phi$ and then uses the product construction to check if the intersection of $L(\mathcal{M})$ and $L(\mathcal{M}_{\neg \phi})$ is empty ($L(\mathcal{M}) \cap L(\mathcal{M}_{\neg \phi}) = \emptyset$) [LP85, VW86]. The complexity of the model checking algorithm depends mainly on the size of the formula (i.e. the size of the parse trees of the given formula) and the size of the Kripke structure (i.e. the size of the state transition graph). In the literature, three standard ways have been proposed to measure complexity when dealing with model checking depending on which of the two input parameters are fixed. The first one is called *combined complexity* when both structures and formulas can be changed. The second one is *data complexity* which considers formulas to be fixed and only structures to be part of the input. This measure is considered to be the most useful measure in the context of verification since for most systems or programs the size of the structures can be extremely large, while the size of formulas is in practice small. The third one is *expression complexity* which considers only formulas to be part of the input, while the structures are fixed.

The model checking technique has several major advantages over traditional verification techniques. First, it is an automatic and exhaustive technique, since it checks all possible behaviours of the system without requiring any user supervision. Second, it helps us to avoid constructing possibly long complex proofs that we may need to construct by hand. Third, the logic used for specifications can directly express many of the properties that are needed to verify the correctness of the system.

So if we have a communication protocol $R$ and $P$ is the condition of receiving a request and $Q$ is the condition of sending a reply then this specification can be, when a request arrives eventually in the future we receive a reply (i.e. $R \models (P \Rightarrow future(Q)))$. This represents a temporal specification that we
can typically make in model checking where the model of the system together with the specification will be given to a model checker and we can get one of two answers. Either yes, which means that every possible behaviour of the system satisfies our logical specification, or no and the model checker in this case will give a behavioural counterexample, which is an execution trace of the system that would show in this example, a request being received ($P$) without a reply being sent ($\neg Q$). It has been argued that the ability of model checking to produce a counterexample is a major advantage because it helps us to debug the system under analysis.

However, the technique suffers from serious scalability issues, since it can not handle large systems in an efficient way, due to the state space explosion problem. The state space explosion problem can occur if the system being analysed has many components that can make transitions in parallel. Note that the size of the model may grow exponentially with the number of concurrent components or processes in the system under analysis, which makes the model checking problem infeasible or even impossible. The problem becomes much harder when we consider real-time systems where the size of the model depends not only on the number of the concurrent components but also on the number of clocks in the system. In order to convert model checking technology into a useful tool for engineers, we need to find ways to overcome the state space explosion problem. We will discuss some of these techniques in Section 7.

It should be noted that verifying correctness using model checking technique is a relative concept. That is, when one successfully verifies the correctness of a given system design using model checking, he can only prove its correctness for the one specified model. (Rather than all possible models). So that one can not give a general correctness proof of systems using model checking technique. However, a combination of model checking technique and other techniques such as theorem proving can allow us to tackle systems with intractably large
or even infinite state spaces. We do not give too many details here about theorem proving technique and we refer the reader to [LKLM05] for more details since we are only concerned with model checking technique. Another obstacle of using model checking for verifying systems correctness is that the effective and proper use of model checking requires great experience on the part of the user. To which level we can trust the outcomes or the results of the model checking process depends actually on two factors: (a) how faithfully the model (e.g., a network of timed automata) represents the real system, and (b) to what extent the properties represent the system requirements.

### 2.3.3 Basic Definitions, Terminology, and Concepts

In this section we review some basic terms, definitions, and concepts that are commonly used in the literature of model checking and we feel they are necessary for proper understanding of the remainder of the thesis. In particular, we discuss terms like decidability, decision procedure, complexity of a verification problem, satisfiability problem, validity, tableau-based decision procedures, formal logic, expressive power of a logical language, safety property, liveness property, and liveness bounded property.

There are two main verification problems: model-checking (given a formula and a model, does the formula holds for the model?) and satisfiability (given a formula, does there exist a model satisfying the formula?). If a problem is undecidable, there is no algorithm for it. But in some cases, you can have semi-algorithms that terminate with an answer (yes or no or maybe) or do not terminate.

When the model checking problem with respect to a logical language is proven to be undecidable, very often researchers then study various fragments (sub-languages) of that language in order to understand precisely what makes
model checking problem undecidable. The analysis of the reasons of the undecidability of a language might lead the researchers to propose a new fragment of that language that can be shown to be decidable. Such fragments are usually obtained via applying some certain restrictions on the syntax and the structure of the language. However, if the problem is shown to be decidable, the next interesting question is then to find an effective decision procedure for solving the model checking problem of the given logical language. A decision procedure is an algorithm which checks whether a formula is valid in a given decidable theory. It always terminates with either a positive or a negative answer. Once the model checking problem has been proven to be decidable and a model checking algorithm has been developed, the next basic question in the theory of model checking is measuring its complexity. Complexity of a model checking problem means the growth of resources that are required to solve the problem with respect to the size of the input formula given in a bit counts. The resources can be CPU time, memory space, message counts, etc. Very often when studying complexity of model checking problems, we are mainly concerned with CPU time and memory space. In the field of computational complexity theory, we say that a decision problem is difficult to solve if the computational resources needed by the most efficient algorithm that can be found for that problem is relatively large. Some of the complexity classes include: \textit{PTIME}, \textit{PSPACE}, \textit{EXPTIME}, \textit{EXPSPACE}, and undecidability. For example, we say that the solution has an EXPTIME complexity when the problem consumes CPU time exponential to the size of the input formula. We say that the solution has EXPSPACE complexity when it consumes at most memory space exponential to the size of the formula given in bit counts.

We now turn to discuss briefly some basic definitions of formal logic and the various types of properties that are usually used when verifying the correctness of systems design. \textit{Formal logic} is a set of rules (grammars) which allow us to represent systems in a high level way, and hence it enables us to verify
systems in an abstract way. There are several types of logic that can be used to express systems properties. Examples of these logics include: epistemic, belief, time, uncertainty, and ignorance. In general, the selection of the logic depends on the type of the property we intend to analyse. For example, for security protocols, where we are concerned with the way information flows among the agents in the environment, it is therefore natural to use epistemic logic as an analysis logic when analysing security protocols.

When introducing a new logical language, one should give the syntax of the language and the semantic of the modalities that are used in that language. The logical syntax of a language is concerned with the rules used for constructing a logical formula without regard to any interpretation or meaning of the formula. While the logical semantics of a language are concerned with the meaning or the interpretation of the modalities and operators used in that language. However, when studying a logical language we usually consider three problems: the model checking problem which asks whether a given model \( M \) satisfies the property \( \varphi \), the satisfiability problem which asks whether there exists a model \( M \) such that \( M \) satisfies \( \varphi \), and the validity problem which asks whether every \( M \) satisfies \( \varphi \). We say that a logical formula \( \varphi \) is satisfiable if it is possible to find a model that makes \( \varphi \) true. The problem of whether a given formula is satisfiable is a decision problem. On the other hand, we say that a formula \( \varphi \) is valid if it is true under all circumstances. The satisfiability and validity problems are elementary concepts in the theory of model checking. Because of the close mathematical relationship between the two concepts, one can express the question of the validity of a formula to one involving satisfiability. We say that \( \varphi \) is true (valid) if and only if \( \neg \varphi \) is unsatisfiable, which is to say it is not true that \( \neg \varphi \) is satisfiable. Very often researchers use the so-called tableau-based decision techniques for satisfiability checking. Traditionally, tableau-based techniques work by decomposing the formula (whose satisfiability is under test) using a set of formal decomposition rules in order
to “semantically” simplify the formula. The main idea is to systematically build a model for the input formula. If all the representation models that are constructed by the method contain a contradiction (i.e. reach an inconsistency state), then we conclude that the formula is unsatisfiable. However, if at least one branch of the constructed tree is free from any inconsistent node, then the search has succeeded and the formula is satisfiable [BS00].

When studying a logical language we often use the term expressive power to evaluate the usefulness of that language. The expressive power of a formalism is measured by the set of properties that can be specified using it. The notion of expressive power is usually used when comparing logical languages that can express or describe the same kind of formulas. One can for example compare the expressive power of TCTL and MTL languages which have been designed for the purpose of expressing timing properties of real-time systems. It has been shown that the TCTL language lacks the ability to express properties that contain multiple timing constraints, which can be expressed in MTL language. However, the restricted syntax of the TCTL language allows more efficient reasoning than MTL. In general, there is a trade-off between expressive power of a language and the complexity of the reasoning in that language. The more expressive power the language has, the harder to reason about in that language.

Temporal logics are often classified into linear and branching time logics. In a system with a linear sequence of computation, where we have only one possible future, it is suitable to use linear temporal logic to express its properties, while in systems with branching sequence of computation, where we have many possible futures we need to use branching temporal logic in order to express their properties. There are five basic operators in the linear time logic, which are: \(X\) (“next time”), \(G\) (“always or globally”), \(F\) (“eventually or in the future”), \(U\) (“until operator”), \(R\) (“release operator”), and two path quantifiers which are used to describe the branching time logic: \(A\) (“for all
computation paths”) and E (“for some computation paths”) [CGP01].

When verifying the correctness of systems design, we are often interested in verifying two types of properties: safety properties and liveness properties, which are probably the most important properties in verification. A safety property asserts that for every possible execution of the system nothing bad or undesirable happens. For example, in the mutual exclusion problem [Lam86], a bad situation that we want to avoid is the situation where two or more processes are in their critical sections at the same time. We can express this property in temporal logic as follows:

\[ \bigwedge_{i \neq j} (AG \neg(i.\text{critical-section} \land j.\text{critical-section})) \]

On the other hand, a liveness property asserts that something good eventually happens. The question: does every process eventually get an opportunity to access its critical section, is an example of liveness properties, which can be expressed as follows:

\[ \bigwedge_{i=1..n} (AF (i.\text{critical-section})) \]

However, in the context of real-time systems, we are also interested in verifying properties like bounded liveness properties which means that something good should happen within fixed determined amount of time. It should be noticed that bounded liveness properties are also safety properties and thus they benefit from nice algorithmic properties. As an example, in the context of transaction processing systems, we might have a property like “every transaction request should be processed within 10 time units”, which can be expressed as follows:

\[ AG (\text{request} \Rightarrow AF_{\leq 10} \text{processed}) \]
2.4 Formal Analysis of Timed Systems

As Manna and Pnueli identified in [MP95] a formal analysis framework should contain the following elements:

– A semantic model that is able to capture the behaviour of systems.
– A system-modelling language to describe systems.
– A property-specification language to express the properties that a system should satisfy.
– Algorithms and techniques to analyse systems with respect to their properties.

Of course, without a formal analysis framework, rigorous and mechanical verification of systems would be impossible. In general, when designing a formal analysis framework, we want the modelling language to be sufficiently powerful so that it can model a wide range of systems. We want also the specification language to be sufficiently expressive so that it can capture the most commonly occurring properties. Finally, we want the algorithms to be as efficient as possible so that they can analyse systems in a practical way. Various frameworks have been proposed for analysing real-time systems. Each framework has its unique advantages and may incur some limitations. In the following subsections we discuss a formal framework for timed systems that has been considered to be the most powerful framework for describing and analysing timed systems.

2.4.1 Semantic Model: Dense Time

For Metric temporal logics (MTLs) there are two different semantic models that can be used for describing systems, the discrete-time semantics [AH94, MP95] and the dense-time semantics [Dil90, AD90]. In the discrete-time semantics for MTLs, all time readings are integer, all clocks increment their
values at the same time, and events can only occur at integer time values. Discrete time models are suitable for synchronous systems where all concurrent processes are synchronised by a single global clock. This model has been successfully used to analyse a large class of systems including synchronous hardware designs and distributed systems [CGP01, Chap. 16].

On the other hand, in the dense-time semantics for MTLs the time readings can be rational or reals and the values of clocks increment at a uniform rate. Today, for applications involving real-time and hybrid systems, and with a system-wide view of overall required behaviour, it is more likely that an underlying dense, or specifically real-numbered, model of time is used and a formal logical language is employed with the ability to express metric requirements as well as the relative order and overlap of propositional states and events. There are many applications involving dense time from multimedia [SLFC96], databases [TCG+93], artificial planning [AKPT91], and natural language processing [JF00] but motivational force for this move comes from the need to consider composition, refinement, and accurate durations in the specification, design and verification of complex systems. However, the state space of dense-time models is infinite (uncountable) and therefore can not be directly model checked. Nevertheless, there are several abstraction techniques that can be used to obtain a finite representation of the infinite state space of timed systems include the region [ACD90] and zone [Alu98] methods. We discuss these techniques in depth at Section 2.6.

As we will show later the choice of the syntax of time in the language affects strongly the verification complexity and the satisfiability problems. As you might expect, the discrete time models lead to lower complexity of the model checking problem than the dense-time model, since there are many fewer states in discrete models. For example, the model checking complexity of CTL logic has been shown to be decidable with a linear time [CE82], while it is PSPACE-complete for its dense-time version (TCTL) [ACD90].
2.4.2 System Modelling Language: Timed Automata Model

Developing formal methods for verifying real-time systems has been an active area of research. Several researchers have proposed different modeling formalisms for describing real-time systems such as timed transition systems [LLPY97], timed I/O automata [KLSV03], timed automata model [AD94], and modelcharts [JM87]. However, the timed automata model introduced by Alur and Dill has been shown to be a powerful model for describing real-time systems where it allows us to easily express timing constraints (delays) between events in real-time systems. The literature on timed automata is very rich, and many variations of the model have been proposed (see, e.g., [Tri98, HNSY92, BBD+02]) and adopted by model-checkers. In this section we will discuss the properties of the original timed automata model and some major techniques and algorithms that have been developed for verifying them.

We will restrict our attention to the reachability problem of timed automata. Timed automata can model several interesting features of real-time systems including qualitative features such as liveness, fairness, and nondeterminism and quantitative features such as bounded response and timing delays.

Timed automata are an extension of the classical finite state automata with clock variables to model timing aspects [AD94]. Let $X$ be a set of clock variables, the clock valuation $v$ for the set $X$ is a mapping from $X$ to $\mathbb{R}^+$ where $\mathbb{R}^+$ denotes the set of non-negative real numbers. Let $C(X)$ be the set of clock constraints defined as follows:

- All inequalities of the form $x \sim c$ or $c \sim x$ are in $C(X)$, where $\sim$ is either $<$ or $\leq$ and $c$ is a non-negative real number.
- If $\varphi_1$ and $\varphi_2$ are in $C(X)$ then $\varphi_1 \land \varphi_2$ is in $C(X)$.

**Definition 2.4.1.** A timed automaton $A$ is a tuple $(\Sigma, L, L_0, L_F, X, I, E)$, where
2.4. FORMAL ANALYSIS OF TIMED SYSTEMS

- $\Sigma$ is a finite set of actions.
- $L$ is a finite set of locations.
- $L_0 \subseteq L$ is a finite set of initial or starting locations.
- $L_F \subseteq L$ is a finite set of final locations.
- $X$ is a finite set of clocks.
- $I : L \rightarrow C(X)$ is a mapping from locations to clock constraints, called the location invariant.
- $E \subseteq L \times L \times \Sigma \times 2^X \times C(X)$ is a finite set of transitions. An edge $(l, l', a, \lambda, \phi)$ represents a transition from location $l$ to location $l'$ after performing action $a$. The set $\lambda \subseteq X$ gives the clocks to be reset with this transition, and $\phi$ is a clock constraint over $X$.

**Definition 2.4.2.** The states in a timed automaton $A$ are tuples $(l, v)$, where $l$ is the current location of the automaton, and $v$ is a function that maps each of the clocks of the automaton to a non-negative real number. The initial states are of the form $(l_0, v_0)$ where $l_0 \in L_0$ and the valuation $v_0(x) = 0$ for all $x \in X$.

Note that the time domain of TA can be either dense domain ($\mathbb{R}^+$) or discrete domain ($\mathbb{N}$). Both choices are possible and lead to different complexity classes. However, the most classical choice is $\mathbb{R}^+$.

**Definition 2.4.3.** Transitions of an automaton may include clock resets and guards which give conditions on the interval in which a transition can be executed. We define two types of transitions in timed automata:

1. **delay transitions** that model the elapse of time while staying at some location: for a state $(l, v)$ and a real-valued time increment $\delta \geq 0$, $(l, v) \xrightarrow{\delta} (l, v + \delta)$ if for all $v'$ with $v \leq v' \leq v + \delta$, the invariant $I(l)$ holds.

2. **action transitions** that execute an edge of the automata: for a state $(l, v)$ and a transition $(l, l', a, \lambda, \phi)$ such that $v \models \phi$, $(l, v) \xrightarrow{a} (l', v[\lambda := 0])$. 

So for an automaton to move from a location to another a delay transition followed by an action transition must be performed. We write this as $d_i \rightarrow a_i$.

**Definition 2.4.4.** A model for a timed automaton $A$ is an infinite state transition graph $I(A) = (\Sigma, Q, Q_0, R)$. Each state in $Q$ is a pair $(l, v)$ where $l \in L$ is a location and $v : X \rightarrow \mathbb{R}^+$ is a clock assignment, mapping each clock to a non-negative real value. The set of initial states $Q_0$ is given by \{(l, v) \mid l \in L_0 \wedge \forall x \in X [v(x) = 0]\}. The transition relation $R$ of $I(A)$ is obtained by combining the delay and action transitions. We write $(l, v) d \rightarrow (l', v') a \rightarrow (l'', v'')$ if there exists $l'''$ such that $(l, v) d \rightarrow (l''', v''') a \rightarrow (l', v')$ for some $d \in \mathbb{R}^+$.

**Definition 2.4.5.** A run of a timed automaton $A = (\Sigma, L, L_0, L_F, X, I, E)$ with an initial state $(l_0, v_0)$ over a timed trace $\zeta = (t_1, a_1), (t_2, a_2), ...$ is a sequence of transitions of the form

\[
\langle l_0, v_0 \rangle \xrightarrow{d_1} a_1 \xrightarrow{a_1} \langle l_1, v_1 \rangle \xrightarrow{d_2} a_2 \xrightarrow{a_2} \langle l_2, v_2 \rangle, ...
\]

satisfying the condition $t_i = t_{i-1} + d_i$ for all $i \geq 1$ and that $l_0 \in L_0$.

Note that we do not require a run in a TA to be a time-divergent run and we allow both zeno and non-zeno runs [BG06, Góm06] in our model. However, since the locations of an automaton are decorated with a delay-quantity and that transitions between locations are instantaneous, the delay of a run is simply the sum of the delays spent in the visited locations. Recall that the amount of time that can be spent in a certain location is described by means of invariants on a number of clock variables.

**Definition 2.4.6.** Let $r = \langle l_0, v_0 \rangle \xrightarrow{d_1} a_1 \xrightarrow{a_1} \langle l_1, v_1 \rangle \xrightarrow{d_2} a_2 \xrightarrow{a_2} \langle l_2, v_2 \rangle \xrightarrow{d_3} a_3 \xrightarrow{a_3} \langle l_3, v_3 \rangle ...$ be a timed run in the set of runs $\mathcal{R}$. The delay of $r$, $delay(r)$, is the sum $\sum_{i=1}^{n} d_i$, where $n$ can be infinity.

The transition system in timed automata is infinite even if the clocks in TA are integer valued, therefore timed automata models can not be directly model
checked. However, there exist methods to reduce the infinite state space of timed systems to finite space while preserving properties of interest. Examples of abstraction methods include the region [ACD90] and zone [Alu98] methods.

Perhaps, the most interesting question to ask about a timed automaton is the reachability of the final state of a given system. Reachability is a fundamental problem in verification. For timed automata, it is stated as follows: given a timed automaton $A$ with an initial location $l_0 \in L_0$ and a set of final locations $L_f$, does there exist a run leading to some state $(l, v)$ with $l \in L_f$? This problem has been proved decidable and shown to be PSPACE-complete by Alur and Dill [AH94]. As we discussed before, the key reason behind that is the construction of the clock regions [ACD90] and clock zones [Alu98]. This is done in such a way that checking a reachability property in a timed automaton is reduced to checking the property in an untimed finite automaton. For example, for the region automaton construction for timed automata [AD94], the precise values of the clocks are not really relevant, and the things which are important in a configuration are the integral parts of the clocks and the relative order of the fractional parts of the clocks. In fact, one of the main reasons for the great advances in verification of timed systems is the use of symbolic techniques [Dil90, HNSY92, YJ94], that are developed in connection with verification tools. These approaches adopt the idea from symbolic model checking for untimed systems. In general, symbolic techniques use boolean formulas to represent sets of states and operations on formulas to represent sets of state transitions.

We now give an example of how one can use timed automata model to describe real-time systems. We consider here a very simple example from the literature, the lamp system [BDL04]. The system is modelled as a set of timed automata, one for each component. Figure 3 shows a timed automaton modelling of a simple lamp system. Each automaton has a set of states (locations) that capture all information about the current state of the system,
CHAPTER 2. BACKGROUND

and a set of edges that capture the transitions between states. The lamp has three locations: off, low, and bright. The user (see Figure 4) and the lamp synchronise with each other using the synchronisation label press. When the user presses the button the lamp is turned on. If the user presses the button again, the lamp is turned off. However, if the user is fast and rapidly presses the button twice, the lamp is turned on and becomes bright. The clock $x$ of the lamp is used to detect if the user was fast ($x < 5$) or slow ($x \geq 5$).

2.5 Specification Languages: Real-time Temporal Logic

The idea of verifying computer systems by means of formal reasoning within proof systems for temporal logic was first proposed by Pnueli in 1977 [Pnu77].
Temporal logic is now a widely ranging and extremely active area of theoretical computer science with heavy duty applications in industry and developments reported in many international conferences. The original logic was propositional linear time temporal logic (PLTL), which is based on a discrete, step-by-step natural numbers model of time and the successes of formal verification via temporal logic have been built on this foundation. Nevertheless, the temporal logic research community is large and interested in a wide range of potential applications and so there have also grown up alongside PLTL a variety of other temporal logics appropriate for a variety of other reasoning tasks. In fact, it has long been acknowledged that dense or specifically real-numbers time models may be better for many applications, ranging from philosophical, natural language and AI modelling of human reasoning to computing and engineering applications of concurrency, refinement, open systems, analogue devices and metric information.

As indicated by Koymans [Koy90] there are several issues that need to be considered when incorporating time. For example, one should address the following issues:

- How should time elements be represented (explicitly or implicitly)?
- What is the notion of time reference (absolute or relative)?
- Which semantical time domain to choose (discrete or continuous)?
- How is time measure presented (additive or metric)?

Following the great success of classical (untimed) temporal logic in verifying discrete systems, timed temporal logics have been proposed, which extend the classical temporal logic via augmenting its modalities with timing constraints. Several timed extensions of CTL [CES86] and LTL [Pnu77] have been proposed, where the TCTL logic, the real-time extension of the branching temporal logic CTL, has been first proven suitable (from a decidability point-of-view) for model-checking purposes. On the contrary, model checking
the timed versions of LTL, like MTL [AH90] and TPTL [ACD90], is either
undecidable or very hard [AH90, Hen96, OW05]. However, the TCTL logic
has expressiveness limitations that hamper the natural representation of re-
quirements due mainly to the limited syntax of the TCTL logic and to its
branching nature. For instance, one can not express properties with multi-
ple timing constraints using TCTL logic. On the other hand, the MTL and
TPTL logics have richer expressiveness than TCTL but the model checking
complexity of these logics is either undecidable or infeasible.

In the literature, two ways have been proposed to extend the classical temporal
logic into timed temporal logic so that it can express real-time properties. The
first way is via decorating the basic modalities such as until and eventually
with an interval. So now one can write the subformula $F_{[a,b]} \varphi$, where $a$ and $b$
are integers, which means within $a$ and $b$ time units, the subformula $\varphi$ must
hold. The MTL and the TCTL languages use this way of modeling to capture
timed properties. The other extensions use explicit formula clocks in their
logical languages, that are assumed to grow at the same rate as time, and it
can be reset before evaluating a subformula. The TPTL language uses this
way of modeling to express timed properties. For example, one can write the
subformula $x.F(q \land x \leq 4)$ in TPTL which means that $q$ will hold within 4
time units from now.

In the following subsections we discuss the syntax and the semantics of the
three logics and give a comprehensive overview of decidability results of their
model checking problem. We discuss also the set of fragments of MTL that
have been proposed in the literature with their decidability and complexity
results.
2.5. SPECIFICATION LANGUAGES: REAL-TIME TEMPORAL LOGIC

2.5.1 Timed Computation Branching Temporal Logic

In the branching time formalism, the system is viewed as a tree, where the branches in the tree correspond to the possible executions or runs of the systems. This is a useful formalism for verifying systems that have many different paths in the future, where any one of which might be an actual path that is realised. The CTL logic has been introduced in 1982 by Emerson and Clarke [CE82] as a specification language for verifying finite-state systems.

Let us briefly review the syntax and the semantics of the CTL. Let $AP$ be a set of atomic propositions. The formulas of CTL can be defined inductively as follows:

$$
\text{CTL} \ni \phi ::= p \mid \neg \phi \mid \phi_1 \lor \phi_2 \mid \text{EX} \phi \mid \text{E} \phi_1 U \phi_2 \mid \text{A} \phi_1 U \phi_2
$$

where $p \in AP$, and $\phi_1$ and $\phi_2$ are CTL formulas. The sub-formula $\text{EX} \phi$ means there is an immediate successor state, reachable by executing one step, in which $\phi$ holds. The modality $\text{A}$ means along all paths from the current state, and the modality $\text{E}$ means along at least one path from the current state.

The semantics of CTL is defined with respect to a Kripke structure. Recall that a Kripke structure $M$ is a triple $(S, S_0, R, L)$, where $S$ is the set of states, $S_0 \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is the transition relation, and $L : S \rightarrow 2^{AP}$ is a function that labels each state with a set of atomic propositions true in that state. A path or a run is an infinite sequence of states $(s_0, s_1, \ldots) \in S$ such that $(s_i, s_{i+1}) \in E$ where $R$ represents a binary relation over $S$ giving the possible transitions. Given a formula $\phi$ and a state $s \in S$, the satisfaction relation $M, s \models \phi$ is defined inductively on the syntax of $\phi$ as follows.

- $M, s \models p$ iff $p \in L(s)$
- $M, s \not\models \neg \phi$ iff $M, s \not\models \phi$
– \( M,s \models \varphi \lor \psi \) iff \( M,s \models \varphi \) or \( M,s \models \psi \)
– \( M,s \models E\varphi U\psi \) iff there is a run \( \zeta \) in \( M \) from \( s \) such that \( \zeta \models E\varphi U\psi \).
– \( M,s \models A\varphi U\psi \) iff for any run \( \zeta \) in \( M \) from \( s \) such that \( \zeta \models E\varphi U\psi \).
– \( \zeta \models \varphi U\psi \) iff there exists a position \( \pi > 0 \) along \( \zeta \) such that \( \zeta[\pi] \models \psi \), for every position \( 0 < \pi' < \pi \), \( \zeta[\pi'] \models \varphi \), and duration \((\zeta_{\leq \pi}) \in I\)

The satisfiability problem for CTL is EXPTIME-complete. However, the model checking problem is in PTIME, as stated in the following theorem.

**Theorem 2.5.1.** [CES86] Given a CTL formulas \( \varphi \), and a finite Kripke structure \( \mathcal{M} = (S,S_0,R,L) \), there is an algorithm for deciding whether or not \( \mathcal{M} \) satisfies \( \varphi \) with time complexity \( O(|\varphi|(|S| + |E|)) \).

An easy way to allow the verification of dense-time properties is to add bounds in the CTL temporal operators. The extended logic is called TCTL. The expressive power of TCTL is similar to the CTL but it incorporates time-constrained modalities in order to specify timing properties. We now give the syntax and the semantics of the TCTL logic.

\[
\text{TCTL} \ni \varphi ::= p \mid \neg \varphi \mid \varphi_1 \lor \varphi_2 \mid E\varphi_1 U_I \varphi_2 \mid A\varphi_1 U_I \varphi_2
\]

where \( I \) is an interval of \( \mathbb{R}^+ \) that can be bounded, unbounded, singular, non-singular, closed or open, or semi closed. However, the bounds of \( I \) have to be in \( \mathbb{N} \cup \{\infty\} \) since this is crucial for all decidability results.

The basic TCTL modality in the above definition is the *until* modality \( U \)-modality which can be used to define the time interval in which the property should be true. As we notice, TCTL has no next operator \( X \), because the time is dense, then by definition, there is no unique next time [Alu92]. There are two different semantics of TCTL, the continuous semantic and the discrete (pointwise) semantic. Both of them share the same modalities, but the interpretation of the modalities in the two semantics is different. While the
formulas in the continuous semantic are interpreted over timed state sequence, they are interpreted over a discrete state sequence in the pointwise semantic. So they differ mainly in the way they interpret the term position. In the continuous semantics, a position $\pi$ in a run $\zeta$ can be any state along $\zeta$. For example, if there is a transition $\langle l, v \rangle \xrightarrow{a,\tau} \langle l', v' \rangle$ in $\zeta$, then any state $(l, v + t)$ with $0 \leq t \leq \tau$ is a position of $\zeta$. One advantage of continuous semantic over pointwise semantic is that it gives us the ability to assert formulas at arbitrary time points (positions). However, as we will see latter, the model checking problem under the continuous setting is usually much harder than it is under the pointwise setting.

Given a formula $\varphi$ and a configuration $(l, v)$ of a timed automata $A$, the satisfaction relation $(l, v) \models \varphi$ is defined inductively on the syntax of $\varphi$ as follows.

- $(l, v) \models p$ iff $p \in L(l)$
- $(l, v) \models \neg \varphi$ iff $(l, v) \not\models \varphi$
- $(l, v) \models \varphi \lor \psi$ iff $(l, v) \models \varphi$ or $(l, v) \models \psi$
- $(l, v) \models E \varphi U I \psi$ iff there is a run $\zeta$ in $A$ from $(l, v)$ such that $\zeta \models \varphi U I \psi$.
- $(l, v) \models A \varphi U I \psi$ iff for any run $\zeta$ in $A$ from $(l, v)$ such that $\zeta \models \varphi U I \psi$.
- $\zeta \models \varphi U I \psi$ iff there exists a position $\pi > 0$ along $\zeta$ such that $\zeta[\pi] \models \psi$, for every position $0 < \pi' < \pi$, $\zeta[\pi'] \models \varphi$, and duration $(\zeta_{\leq \pi}) \in I$

**Example 2.5.2.** In TCTL, one can express a wide variety of properties such as safety properties and bounded liveness properties. We give here a simple example of bounded liveness properties.

$$AG(fault \Rightarrow AF_{[0,2]}repaired)$$

The property states that every time a fault occurs then along all possible runs, later it will be repaired within no more than 2 time units.
In [ACD90] Alur et al have shown that the satisfiability problem of TCTL is undecidable, while the model checking problem with respect to TCTL is decidable with a PSPACE-Complete complexity. Once the model checking problem has been proven to be decidable, the next question is to find an (optimal) algorithm solving the problem. In fact, several algorithms have been proposed for model checking TCTL logic [ACD90, HNSY92, Tri98]. The proposed algorithms differ mainly in how they explore the state space of the system under analysis. We do not give more details for the model checking of TCTL formulas, as these results are well-known.

**Theorem 2.5.3.** [ACD90] The satisfiability problem for TCTL for a timed automaton $\mathcal{A}$ is $\Sigma_1^1$-hard

**Theorem 2.5.4.** [ACD93] The model checking problem for TCTL for a timed automaton $\mathcal{A}$ is PSPACE-Complete.

Several other real-time temporal logics have been proposed in the literature such as the Metric Temporal Logic (MTL) [Koy90] and the Timed Propositional Temporal logic (TPTL) [ACD90]. The MTL temporal logic is a real-time extension of temporal logic that is obtained by augmenting the modalities of linear temporal logic (LTL) with timing constraints. Other real-time extension of temporal logic is the TPTL temporal logic which is a real-time temporal logic that is obtained by extending the linear time propositional temporal logic with a reset quantification allowing one to introduce clocks that are used in timing constraints. This logic with external clock variables has been first proposed in [AH94]. However, it is beyond the scope of this thesis to discuss the syntax and the semantics of these logics.

### 2.6 State Space Abstraction Techniques

Clearly, techniques used for verifying finite automata can not be used for timed automata. Specific symbolic abstraction techniques have to be developed, that
take into account the unique characteristics of timed automata, in particular the fact that clocks in timed automata evolve densely and synchronously with global time, which lead to an infinite number of time transitions. In this section, we review the most basic methods that can be used for analysing timed automata. In the original work of Alur and Dill [AD94], they proposed an abstraction technique by which an infinite timed transition system (i.e. timed automata) can be converted into an equivalent finite state-space of a symbolic transition system called a region graph where reachability is decidable. In fact, this was the first work that gave us the hope of being able to model check real-time systems. However, it has been shown that the region automaton is highly inefficient to be used for implementing practical tools. Instead, most real-time model checking tools like UPPAAL, Kronos and RED apply abstractions based on so-called zones, which is much more practical and efficient for model checking real-time systems.

2.6.1 The Region Abstraction

In the definition of timed automata, we allow clock constraints in the invariant of a location and the guards of the transitions to contain arbitrary non-negative real numbers, which makes the model checking problem of timed automata seem intractable since the number of states is infinite.

In order to obtain a finite representation for the infinite state space of a timed automaton, Alur and Dill [AD94] proposed the so-called clocks regions approach, which represents sets of clock assignments. Several clever observations on the transition system of timed automata have been discovered by Alur and Dill that led them to propose this interesting abstraction technique. The region graph technique allows us to obtain the decidability of many verification problems in the timed automata model. A finite model can be obtained by defining a proper equivalence relation on the state space of the systems. If two states, which correspond to the same location of the timed automata $\mathcal{A}$, agree
on the integral part of all clock values and on the ordering of the fractional parts of all the clocks, then the states have a similar future behaviour. The integral parts of the clock values determines whether a clock constraint in the invariant of a location or in the guard of a transition is satisfied or not, while the fractional parts of the clock values determine which clock will change its integral part first. This is because clock constraints in timed automata can be only compared to integers.

The value of a clock can become arbitrary large. However, if the clock is never compared to a constant greater than $c$, then the value of the clock will have no impact on the computation of $A$ once it exceeds $c$. Suppose for example, the clock $x$ in a timed automaton $A$ is never compared to a constant greater than 50 in any of the invariants or guards in $A$, then it is impossible for an observer to distinguish between $x$ having the value 100 and $x$ having the value 150.

Alur et al [ACD90, Alu92] show how to use the region graph for automatic verification of real-time systems. They formalise the problem as follows.

**Definition 2.6.1.** [Alu92] For any $x \in \mathbb{R}^+$ let $\text{frac}(x)$ be the fractional part of $x$ and let $\lfloor x \rfloor$ be the integral part of $x$. That is, $x = \lfloor x \rfloor + \text{frac}(x)$. Let also $c_x$ be the largest integer $c$ that $x$ is compared with in the invariant of any location or in the guard in any transition. The equivalence relation $\cong$ is defined over the set of all clock interpretations for $X$; $v \cong v'$ if all the following conditions hold:

1. For all $x \in X$ either $\lfloor v(x) \rfloor$ and $\lfloor v'(x) \rfloor$ are the same or both $v(x)$ and $v'(x)$ are greater than $c_x$.
2. For all $x_1, x_2 \in X$ with $v(x_1) \leq c_{x_1}$ and $v(x_2) \leq c_{x_2}$, $\text{frac}(v(x_1)) \leq \text{frac}(v(x_2))$ iff $\text{frac}(v'(x_1)) \leq \text{frac}(v'(x_2))$.
3. For all $x \in X$ with $v(x) \leq c_x$, $\text{frac}(v(x)) = 0$ iff $\text{frac}(v'(x)) = 0$.

We define the equivalence relation $\cong$ on the set of possible clock assignments, we will use $\lfloor x \rfloor$ to denote the clock region to which $x$ belongs. Each region
can be uniquely characterized by a finite set of clock constraints it satisfies. Configurations in the same region have the property that they are indistinguishable to an observer in the sense they have the same sequence of future transitions. Alur and Dill show that $\cong$ has finite index and consequently that the number of regions is bounded.

Lemma 2.6.2. [Alu92] The number of clock regions is bounded by $(|X|! \times 2^{|X|} \times \prod_{x \in X}(2.c_x + 2))$.

As a result of the above lemma, we can construct a finite transition graph (region graph) $R(A)$ that is bisimulation equivalent to the original infinite transition graph $I(A)$ [ACD90, Alu92]. A region is a pair $(l, [v])$. The states of the region graph are the regions of $A$. Every state in $I(A)$ has a corresponding region $(l, [v])$ which will be a state of $R(A)$. The initial states of the region graph have the form $(l_0, [v_0])$ where $l_0$ is the initial state of $A$ and $v_0$ is a clock interpretation that assigns 0 to every clock in $A$. Finally, the transition relation in $R(A)$ is defined in such a way it preserves the bisimulation equivalence relation $\cong$. We summarise the construction of the region graph (automaton) $R(A)$ in definition 2.6.3. We refer the reader to [Alu92] for more details and examples about how one can construct a region graph for a given timed automaton, and for algorithms that can be used for model checking timed systems based on region graph.

Definition 2.6.3. Let $A = (\Sigma, L, L_0, L_F, X, E)$ be a timed automaton. Then,

- The states of $R(A)$ have the form $(l, [v])$ where $l \in L$ and $[v]$ is a clock region.
- The initial states have the form $(l_0, [v])$ where $l_0 \in L_0$ and $v(x) = 0$ for all $x \in X$.
- $R(A)$ has a transition $((l, [v]), a, (l', [v']))$ iff $(s, \omega) \xrightarrow{a} (t', \omega')$ for some $\omega \in [v]$ and some $\omega' \in [v']$. 

Theorem 2.6.4. [CGP01, ch. 17, p. 280]. The state transition graph $\mathcal{I}(A)$ and the region graph $\mathcal{R}(A)$ are bisimilar as transition systems.

Several algorithms for minimizing the region automaton have been proposed [ACH+92, TY96]. However, there is no tool adopting any of these algorithms in its implementation.

2.6.2 The Zone Abstraction

As we discussed in the previous section, the size of the region graph grows exponentially not only with the number of components in a system but also with the largest time constant and the number of clocks that are used to specify timing constraints. Thus the need for an alternative, symbolic approach to model checking, which avoids the explicit construction of the region graph, is highly desirable. Researchers proposed a more efficient representation of the state space of timed automata which is based on zone and zone-graphs [Dil90, HNSY92, YL97]. In a zone graph, zones are used to denote symbolic states. This in practice gives a more compact representation of the state-space.

Definition 2.6.5. (Clock zone). A clock zone is a conjunction of inequalities that compare either a clock value or the difference between two clock values to an integer. We allow inequalities of the following types:

$$x \sim c, c \sim x, x - y \sim c$$

where $x, y \in X$ and $c$ is an integer and $\sim \in \{<, \leq\}$. By introducing a special clock $x_0$ that is always zero, it is possible to obtain a more uniform notation for clock zones. Thus, the general form of a clock zone is

$$x_0 = 0 \land \bigwedge_{0 \leq i \neq j \leq n} x_i - x_j \sim c_{i,j}.$$
Consider a timed automaton $\mathcal{A} = (\Sigma, L, L_0, L_F, X, E)$ with a transition $e = (l, a, \psi, \lambda, l')$ we can construct an abstract zone graph $Z(\mathcal{A})$ such that states of $Z(\mathcal{A})$ are zones of $\mathcal{A}$. A zone is a pair $(l, \varphi)$, where $l$ is a location of $\mathcal{A}$ and $\varphi$ is a clock zone. The clock zone $\text{succ}(\varphi, e)$ will denote the set of clock valuations $v'$ such that for some $v \in \varphi$ the state $(l', v')$ can be reached from the state $(l, v)$ by letting time elapse and by executing the transition $e$. The pair $(l', \text{succ}(\varphi, e))$ will represent the set of successors of $(l, \varphi)$ under the transition $e$. Note that the assignment of the values of the clocks in the initial location of a timed automaton $\mathcal{A}$ is easily expressed as a clock zone since $v(x) = 0$ for every clock $x \in X$. Note that every constraint used in the invariant of an automaton location or in the guard of a transition is a clock zone. Therefore, clock zones can be used for various state reachability analysis algorithms for timed automata. Usually these algorithms use three basic operations on clock zones [Alu98]: Intersection operation $(\varphi \land \psi)$ where $\varphi$ and $\psi$ are two clock zones, Clock Reset, and Elapsing of Time.

We now give a brief description of how one can construct an abstracted zone graph of a given timed automaton $\mathcal{A}$. However, in Section 2.9, we give a formal detailed description of how one can construct a zone graph for a given timed automaton and a full description of the set of the operations that may be needed when constructing a reachability graph of TA using the zone abstraction.

**Definition 2.6.6.** Given a timed automaton $\mathcal{A}$, we can construct an abstract transition graph called zone graph $Z(\mathcal{A})$ following the approach described above. The states of $Z(\mathcal{A})$ are the zones of $\mathcal{A}$. The zone $(l_0, [X = 0])$ is the initial state of $Z(\mathcal{A})$ where $l_0 \in L_0$ and $X = 0$ means the value of any clock in $X$ is 0. There will be a transition from the zone $(s, \varphi)$ in $Z(\mathcal{A})$ to the zone $l', \text{succ}(\varphi, e)$ in $Z(\mathcal{A})$ labelled with the action $a$ for each transition of the form $e = (l, a, \psi, \lambda, l')$ of the timed automaton $\mathcal{A}$. Note that because each step in the construction of the zone graph is effective, this gives an algorithm
for determining state reachability in the state transition graph \( I(A) \).

## 2.7 Representation of State Space

In this section we discuss some common data structures that are used for representing set of states of timed automata. We discuss the following three data structures: Difference Bound Matrices (DBMs), Binary Decision Diagrams (BDDs), and Clock Difference Diagrams (CDDs). However, we focus more on the DBM data structure since it is the most commonly used data structure for representing timing constraints in the TA model.

### 2.7.1 Difference Bound Matrices

Difference bound matrices [Dil90] are the most common data structure for the representation of state spaces of timed automata. A DBM is a two-dimensional matrix that records the difference upper bounds between clock pairs up to a certain constant. Recall that a clock constraint over the set of clocks \( X \) is a conjunction of atomic constraints of the form \( x \sim m \) and \( x - y \sim n \) where \( x, y \in X, \sim \in \{\leq, <, =, \geq, >\} \), and \( m, n \) are integers. In order to have a unified form for clock constraints in a DBM we introduce a reference clock \( x_0 \in X \) with the constant value 0 that is not used in any guards or invariants. The matrix is indexed by the clocks in \( X \) together with the special clock \( x_0 \). The element \( D_{i,j} \) in matrix \( D \) is of the form \((n, \prec)\) where \( x_i, x_j \in X \), \( n \) represents the difference between them, and \( \sim \in \{\leq, <\} \). Each row in the matrix represents the bound difference between the value of the clock \( x_i \) and all the other clocks in the zone, thus a zone can be represented by at most \( |X|^2 \) atomic constraints. This implies that each pair of the variables \((x_i, x_j) \ (i \neq j)\) will be represented by two atomic constraints \((d_{i,j}, \prec)\) and \((d_{j,i}, \prec)\). Since the variable \( x_0 \) is always 0, it can be used to express constraints that only involve a single variable. We call these constraints as one-variable
constraints. For example, $D_{i,0} = (d_{i,0}, \prec)$, means that we have the constraint $x_i \prec d_i$.

Consider the following example by which we explain how one can construct a DBM matrix for a zone $Z$ given by the following formula.

$$Z = x_1 - x_2 < 2 \land 0 < x_2 \leq 2 \land 1 \leq x_1$$

To compute the DBM representation for the zone $Z$ we start by numbering all clocks in $X$ from 0 to $n$ where the index for the reference clock $x_0$ is 0. As described above, we can store zones using $|X| \times |X|$ matrices where each element is on the form $(n, \prec)$ which represents an atomic constraint. Let each clock be denoted by one row in the matrix. The row is used for storing lower bounds on the difference between the clock and all other clocks. The corresponding column is used for storing upper bounds. Note that we use the special value $\infty$ when no bound is present in the given zone. Intuitively, the difference between a clock and itself is always 0. This explains why the entries along the major diagonal will have the form $(0, \leq)$. Note that the first row and column in the matrix is labelled with the reference clock $x_0$. From the given representation of DBMs we note that the first column in the matrix encodes the upper bounds of the clocks since the constraints in that column are of the form $x_i - x_0 < c$. Recall that the reference clock $x_0$ is always 0. On the other hand, the first row in the matrix encodes the lower bounds of the clocks where the constraints on that row are of the form $x_0 - x_i \prec c$. We know from [AD94, CGP01] that every clock zone can be represented by a DBM and every DBM represents a clock zone. However, the representation of a zone by a DBM is not unique in the sense that the same zone can be represented by more than one DBM. This is because some of the bounds in the matrix may not be tight enough. The first straightforward DBM representation for zone $Z$ will be as follows.
\[ D_1(Z) = \begin{pmatrix}
  x_0 & x_1 & x_2 \\
  x_0 & (0, \leq) & (-1, \leq) & (0, <) \\
  x_1 & (\infty, <) & (0, \leq) & (2, <) \\
  x_2 & (2, \leq) & (\infty, <) & (0, \leq)
\end{pmatrix} \]

Note that there are some implicit constraints that are not reflected in the matrix \( D_1(Z) \). For example, since \( x_1 - x_2 < 2 \) and that \( x_2 - x_0 < 2 \), then it must be the case that \( x_1 - x_0 < 4 \) since \( x_0 = 0 \). This observation can be used to tighten the difference bound matrix \( D_1(Z) \). We therefore obtain a new matrix \( D_2(Z) \) for the same zone \( Z \).

\[ D_2(Z) = \begin{pmatrix}
  x_0 & x_1 & x_2 \\
  x_0 & (0, \leq) & (-1, \leq) & (0, <) \\
  x_1 & (4, <) & (0, \leq) & (2, <) \\
  x_2 & (2, \leq) & (\infty, <) & (0, \leq)
\end{pmatrix} \]

Clearly, the matrix \( D_2(Z) \) represents the same set of clock valuations as the original matrix \( D_1(Z) \). Note that since entries of DBM represent bound differences between the values of the clocks in the model, it is possible sometimes to derive some constraints using the other constraints. These constraints can be derived by benefiting from the implicit relations between the entries in the matrix. For example, the sum of the upper bounds on the difference \( x_i - x_j \) and \( x_j - x_k \) is an upper bound on the difference \( x_i - x_k \). That is, \( d_{i,k} = d_{i,j} + d_{j,k} \).

We can use this observation to tighten the difference bound matrix. This operation is called a tightening operation. We can repeat this operation until further application of the operation does not change the matrix. At that time we obtain what we call a canonical form of the matrix which is a matrix that contains the most precise constraints that can be derived for the zone under consideration. The most commonly used algorithm for computing canonical form of DBMs is the Floyd-Warshall algorithm [Flo62] which can be described algorithmically as shown in Figure 1.
Algorithm 1: Floyd-Warshall algorithm for computing shortest path

Using the Floyd-Warshall algorithm presented in Figure 1 the canonical form of the difference bound matrix for the above zone $Z$ is as follows.

$$
\begin{bmatrix}
\ x_0 & x_1 & x_2 \\
\ x_0 & (0, \leq) & (-1, \leq) & (0, <) \\
\ x_1 & (4, <) & (0, \leq) & (2, <) \\
\ x_2 & (2, \leq) & (1, <) & (0, \leq) \\
\end{bmatrix}
$$

In fact, canonical forms simplify some operations over DBMs like the test for equality or inclusion. For example, when comparing whether two zones are equivalent we need to verify whether the corresponding canonical DBMs of these zones are identical. In order to do so we need to represent zones using matrices that have standard form (canonical form) which allows us to compare zones correctly.

### 2.7.2 Binary Decision Diagrams

A Binary Decision Diagram (BDD) \cite{Bry86, BCM90} is a propositional directed acyclic graph. The graph consists of a set of decision nodes and has two terminal nodes TRUE-terminal and FALSE-terminal. Each decision node is labeled by a Boolean variable and has two child nodes called low child and high child. A path from the root node to the TRUE-terminal represents a variable assignment for which the represented Boolean function is true. As the path descends to a low child (high child) from a node, then that node’s variable is assigned to TRUE (FALSE). For untimed system verification, BDD
has shown great success. But for timed system verification, so far, all BDD-like structures have not performed as well as the popular DBM. The BDD data structure is used in the tool Rabbit (see Section 2.10.5).

### 2.7.3 Clock Restriction Diagrams

Clock Restriction Diagrams (CRDs) [Wan01] is a BDD-like data structure for representation of sets of zones, with related set-oriented operations for fully symbolic verification of real-time systems. It has similar structure as BDD without FALSE terminal. Unlike BDD, CRD is not a decision diagram for state space membership. It acts like a database for zones and is appropriate for manipulation of sets of clock difference constraints. It has been claimed that CRDs provide more efficient space representation of timed automata than DBMs data structure. The CRD technology is used in the current version of the tool RED. More details and examples of the CRD data structure can be found in [Wan01, Wan04a].

### 2.7.4 Clock Difference Diagrams

A clock difference diagram [LPWY99] or short as CDD is a BDD-like data-structure for federations (i.e. finite unions of zones), where inner nodes are associated with a given pair of clocks and outgoing arcs state bounds on their difference. This data-structure contains DBM’s as a special case and offers simple boolean set-operations and easy inclusion and emptiness-checking. A CDD is a compact representation of a decision diagram for federations, so we take a valuation, and follow the unique path along which the constraints given by type and interval are fulfilled by the valuation. If the path ends at a true node, the valuation belongs to the federation represented by this CDD, otherwise not. A version of the real-time verification tool UPPAAL using
CDD’s data structure has been implemented and tested on 8 industrial examples [PWY+98]. The experimental results reported demonstrate significant space-savings with a moderate increase in run-time, compared to the original implementation of the UPPAAL tool based on DBM data structure.

2.8 Techniques for Constructing Reachability Graph of Systems

One of the interesting issues that affects the performance of model checking problems is the size of the reachability graph of the system under consideration, which is also known as the state space. The larger the size of the graph, the longer the verification process. In this section we review some basic techniques that are used in the construction of state space representations of real-time systems. In fact, several techniques and algorithms have been proposed for representing, characterizing and generating state space of real-time systems. However, we discuss here only the basic approaches like the forward and backward approach, on-the-fly approach, and compositional approach. As we mentioned before, the state explosion problem represents one of the major limitations of the model checking technique. However, the problem becomes more serious when considering real-time systems since the state space of real-time systems is infinite. The question of which technique to choose when constructing the state space of a given system model is an important question since the size of search space affects the computational cost of the model checking problem. The idea is to find an error trace (if any) of the given finite state machine rapidly. So given a timed automaton $A$ and a property $\phi$, we construct a reachability graph from $A$ and then search for a trace in which $\phi$ fails. In general, two main approaches have been discussed in the literature, the fixed-point approach and on-the-fly approach. In the
fixed-point approach, an exhaustive search is used: all states need to be represented in memory at the same time. In the on-the-fly approach only part of the reachability graph is generated, and the property is checked while the graph is constructed. However, it should be noted that the approaches described below can be used not only for constructing the state space of real-time systems but also for untimed or discrete systems.

2.8.1 Backward Versus Forward Analysis

In the forward analysis method the model checker attempts to construct a characterization of all states that can be reached from a set of initial states $I$ with respect to the declared behaviour structure $M$. In the backward reachability analysis the model checker attempts to construct a characterization of all states that can reach specific goal states (unsafe states) with respect to the declared behaviour structure. The majority of the existing tools support both forward and backward analysis techniques, where both of them can perform well for checking safety properties. In general, backward model checking may be better than forward model checking for some examples, and vice versa. However, the backward analysis technique is necessary for model checking certain modalities such as “until” $U$ and “eventually” $F$ modalities, since these modalities can be only handled with backward fixpoint computations [Wan04b]. Kronos and RED are examples of tools that support both forward and backward model checking of TCTL logic.

2.8.2 On-the-Fly Approach

In the on-the-fly approach [BTY97], the state space of the system is generated dynamically and only the minimal amount of information is stored in memory. The property is checked while the graph is being generated. On-the-fly model checking is a powerful verification technique, especially efficient
2.8. TECHNIQUES FOR CONSTRUCTING REACHABILITY GRAPH OF SYSTEMS

when the specification is false and significantly efficient when the computation needed to get to a failing state is short. One big advantage of on-the-fly verification is that it needs only proceed until an error or a fault is found, in this case a counterexample is generated to assist the designer with error correction. Very often, errors are discovered very early during the search, thus avoiding the exploration of the entire state space. On the other hand, if the system is correct with respect to the desired specification, the search needs to cover the entire state. The technique is therefore particularly appropriate for discovering errors in early stages of design. On-the-fly based techniques have been used to check safety properties, deadlock, and timelock detection where all of these properties can be reduced to reachability properties [Tri98]. Model checking algorithms based on on-the-fly approach can sometime consume a lower memory and provide an answer to the given model checking problem faster than the fixed-point methods. However, there are examples where on-the-fly approach performs better than fixed-point approach and vice versa. UPPAAL is an example of model checker that supports on-the-fly approach model checking.

2.8.3 Compositional Approach

The state space explosion problem arises in systems or programs composed of many parallel or concurrent components. For such systems, the model checking technique might be inefficient or impossible at all. A possible solution to this problem is to decompose complex systems into components or processes and then to verify them individually [BCC98]. If the verification results of all the components are positive then we conclude the system is correct. The main difficulty in this process is to know when some property of a process remains correct in a parallel composition involving that process. Clearly, one can easily find examples of parallel systems where this approach can not be applied where processes need to interact with each other in order
to achieve their intended goals, and therefore one can not verify processes individually. Formally, given a network of timed automata \(A_1, \ldots, A_n\), and a formula \(\phi\) the compositional model checking problem is to prove that \(A_1 \parallel \ldots \parallel A_n \models \phi\). More precisely, we would like to verify that the parallel composition of these automata satisfies the formula \(\phi\) without having to construct the complete state space of \(A_1, \ldots, A_n\). There are some evidences in the literature that compositional model checking can be an efficient technique for real-time systems [LLL95, JXXG98]. However, more experiments need to be performed to prove that this technique is indeed a useful technique for mitigating the effects of the state explosion problem of real-time systems.

2.9 Basic Operations on Zones

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.) In another words, how 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. We review here the elementary operations that are used to construct the zone graph of a given automaton using DMB, which are the intersection operation, the reset operation, the delay operation or the up operation, the and operation, and the inclusion checking operation.

To ensure efficiency of the implementation of the zone approach, efficient manipulation of convex zones and canonicity are important. A zone \(Z\) is called convex if for all \(v_1, v_2 \in Z\), for any \(0 \leq \delta \leq 1\), \(\delta v_1 + (1 - \delta)v_2 \in Z\). This means the line joining any two points \(v_1, v_2 \in Z\) lies inside \(Z\). Note that a set containing non-convex zones that are apart from each other cannot be 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. It is therefore necessary that the most frequently used operations preserve convexity of zones. We give now the set of operations that are used in constructing the zone graph of a given automaton.

**Definition 2.9.1.** (The intersection operation). The intersection of two DBMs is computed entry-wise. Given two canonical DBMs $D_1$ and $D_2$, and $D' = D_1 \land D_2$, then

$$D'_{i,j} = \min\{D^1_{i,j}, D^2_{i,j}\}$$

**Definition 2.9.2.** ($\text{UP}(D)$). Elapsing time means that the upper bounds of the clocks are set to infinity. That is, after that operation $\forall x \in X : x - x_0 < \infty$ holds. Let $D' = D \uparrow$, then:

$$D'_{i,j} = \begin{cases} (\infty, <) & \text{for any } i \neq 0 \text{ and } j = 0, \\ (D_{i,j}) & \text{if } i = 0 \text{ or } j \neq 0 \end{cases}$$

The property that all the clocks advance with the same amount of time is ensured by the fact that the constraints on the differences between clocks are not altered by the operation. 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.

**Definition 2.9.3.** ($\text{reset}(D, \lambda)$). With the reset operation, the values of clocks can be set to zero. Let $\lambda$ be the set of clocks that should be reset. We can define $D' = D[\lambda := 0]$ as follows.

$$D'_{j,k} = \begin{cases} (0, \leq) & \text{if } x_j \in \lambda \text{ and } x_k \in \lambda, \\ D_{0,k} & \text{if } x_j \in \lambda \text{ and } x_k \not\in \lambda, \\ D_{j,0} & \text{if } x_j \not\in \lambda \text{ and } x_k \in \lambda, \\ D_{j,k} & \text{if } x_j \not\in \lambda \text{ and } x_k \not\in \lambda \end{cases}$$
Definition 2.9.4. (inclusion $(D_1, D_2)$). Another key operation in state space exploration is inclusion checking between zones. For any two DBMs $D_1, D_2$ in canonical form, the inclusion operation denoted as $D_1 \subseteq D_2$ returns true if $D_{1,i,j} \leq D_{2,i,j}$ for any $i, j$, otherwise, it returns false.

Definition 2.9.5. (free($D, x$)). The free operation removes all constraints on a given clock. In state space exploration, this operation is used in combination with conjunction to implement reset operation on clocks. The operation can be implemented by changing elements in the matrix $D_1$ as follows:

- $D_{1,i,j} = (\infty, <)$ if $i \neq j$,
- $D_{1,j,i} = (\infty, <)$ if $i \neq j$,
- $D_{1,i,0} = (0, \leq)$,
- $D_{1,0,i} = (\infty, <)$,

Definition 2.9.6. (consistent($D$)). For a DBM to be inconsistent there must be at least one pair of clocks where the upper bound on their difference is smaller than the lower bound. Checking for the consistency of a DBM is done by computing the canonical form and then checking the diagonal for negative entries.

Definition 2.9.7. copy($D, x := y$). This operation can be used in forward state space exploration. It simply copies the value of one clock to another. Recall that each clock in the zone has a lower and upper bound. Hence, the copy operation can be implemented in multiple steps: assigning $D_{x,y} = (0, \leq)$, $D_{y,x} = (0, \leq)$, removing all other bounds on $x$, and then recanonicalize the zone.

Definition 2.9.8. (The extrapolation operation). Let $Z$ be a zone represented by a DBM in a canonical form $D = (m_{i,j}, \prec_{i,j})_{i,j=0,..n}$ and $M$ be a clock ceiling. We can compute the extrapolation function $\text{Extra}_M(D')$ of the
2.9. BASIC OPERATIONS ON ZONES

Zone \( D' = (m'_{i,j}, \prec'_{i,j})_{i,j=0,\ldots,n} \) as follows:

\[
(m'_{i,j}, \prec'_{i,j}) = \begin{cases} 
(\infty, <) & \text{if } m_{i,j} > M, \\
(-M, <) & \text{if } m_{i,j} < -M, \\
(m_{i,j}, \prec_{i,j}) & \text{otherwise.}
\end{cases}
\]

The above operation is known in the literature as the \( M \)-extrapolation operation [DT98] where the zone is extrapolated (abstracted) with respect to the maximum constant each clock is compared to in the automaton. That is, if the clock is never compared to a constant greater than \( M \) in a guard or invariant, then the value of the clock will have no impact on the computation of the automaton once it exceeds \( M \). The procedure to obtain the \( M \)-extrapolation of a given zone is to remove all upper bounds higher than the maximum constant and lowering all lower bounds higher than the maximum constant down to the maximum constant. This operation is necessary in order to guarantee the finiteness of the generated zone graph.

Note that all the above operations preserve the convexity of the zone in the sense the result of applying any of these operations on a zone is also a zone. It is well-known that intersection, addition, comparison preserve convexity of zones while subtraction and union may break the convexity of zones.

We now discuss how the zone or DBM based successor computation can be performed and hence how the zone graph of a given automaton can be constructed. Let \( D \) be a DBM in canonical form. We want to compute the successor of \( D \) w.r.t a transition \( e = (l, a, \psi, \lambda, l') \). The clock zone \( \text{succ}(D, e) \) can be obtained using a number of elementary DBM operations which can be described as follows [Alu98, BY04]:

1. Let an arbitrary amount of time elapse on all clocks in \( D \). In a DBM this means all elements \( D_{i,0} \) are set to \( \infty \). We will use the operator \( \uparrow \) to denote the time elapse operation.
2. Take the intersection with the invariant of location \( l \) to find the set of possible clock assignments that still satisfy the invariant.

3. Take the intersection with the guard \( \phi \) to find the clock assignments that are accepted by the transition.

4. Canonicalize the resulting DBM and check the consistency of the matrix.

5. Set all the clocks in \( \lambda \) that are reset by the transition to 0.

6. Take the intersection with the location invariant of the target location \( l' \).

7. Canonicalize the resulting DBM.

8. Extrapolate and canonicalize the resulting zone at the target location \( l' \) and check the consistency of the matrix.

Combining all of the above steps into one formula, we obtain

\[
D' = (\text{Canon}(\text{Extra}(\text{Canon}(\text{Canon}((\text{Canon}((D^\oplus \land I(l)) \land \phi)[\lambda := 0]) \land I(l'))))))
\]

where \text{Extra} represents an extrapolation function that takes as input a DBM and returns the \( M \)-form of the matrix, while \text{Canon} represents a canonicalization function that takes as input a DBM and returns a canonicalized matrix in the sense that each atomic constraint in the matrix is in the tightest form, \( I(l) \) is the invariant at location \( l \), and \( \uparrow \) denotes the elapse of time operation. Note that intersection does not preserve canonical form [BY04], so we should canonicalize \(((D^\oplus \land I(l)) \land \phi)\) before resetting any clock (if any). Since after executing the transition \( e \) all the clocks in the automaton have to advance at the same rate. After applying the guard, the matrix must be checked for consistency. Checking the consistency of a DBM is done by computing the canonical form and then checking the diagonal for negative entries. The resulting zone at step 5 needs to be intersected with the clock invariant at the target location \( l' \) and extrapolating/canonicalization afterwards. This is necessary in order to ensure that the guard \( \phi \) and the reset operation \(([\lambda := 0])\)
implies the invariant at the target location, as the transition to $l'$ would be
disabled otherwise, and one could erroneously reach a final state with such a
transition, resulting in a wrong answer for the reachability query.

We give now an example by which we demonstrate how one can construct a
finite zone graph of an automaton following the above described approach. Let
us see what happens when we compute the zones of the automaton in Figure 5
using the $M$-extrapolation algorithm. Firstly, note that the extrapolation
constant $M$ is 20. The automaton has two clocks $y$ and $z$. We give the
sequence of zones obtained below. Note that for convenience only the full
canonical zone is written. First at location start we have the zone $(x = 0 \land y = 0 \land z = 0)$. During the forward traversal of the TA the location
loop is reached with the clock zone $(x \leq 10 \land y \leq 10 \land z \leq 10)$. Clearly,
extrapolation is not necessary here since none of the constraints exceeds the
extrapolation constant $M$. After taking the transition loop $\rightarrow$ loop a state
$(\text{loop}, z_2)$ with $z_2 = (x \leq 20 \land y \leq 10 \land z \leq 20)$ will be added. Again
extrapolation is not necessary here. A second loop will add $(\text{loop}, z_3)$ with
$z_3 = (x \leq 30 \land y \leq 10 \land z \leq 30)$. Before proceeding further, note that the
zone $z_3$ needs to be extrapolated since there are some constraints exceed the
value of the extrapolation constant $M$. One can check this would give the
zone $z'_3 = (x \leq 30 \land y \leq 10 \land x - z = 0 \land z = \infty)$.

Note that after executing the loop three times and extrapolate the zone $z_3$
with respect to $M$ and then compute the successor of the state $(\text{loop}, z_3)$ we

Figure 5: An example of a timed automaton by which we demonstrate how one can
compute the zone graph of an automaton.
will find that the new state \((\text{loop}, z_4)\) has the same clock zone as the previous state \((\text{loop}, z_3)\) (i.e. \(z_3 = z_4 = (x \geq 20 \land y \leq 10 \land z \geq 20)\)) and the test \((z_4 \subseteq z_3)\) holds and hence we reach a fixed point and we can safely exit the loop.

2.10 Model Checking Tools for Real-time Systems

In this section, we discuss some well-known tools that have been developed for the purposes of model checking real-time systems. In general, the tools share the same theoretical foundation in the sense they adopted the timed automata model of Alur and Dill or an extension of that model as a description language, and they use the TCTL logic as a specification language. However, they differ from each other in the type of data structure they implement to represent the state space of the system, the techniques they use to construct reachability graph of timed systems, the abstraction techniques they apply, and whether they support a full or a fragment of the TCTL language. The model checkers vary also in how easy, or difficult, it is to formalise systems and their properties in the language of each model checker. We give some hints when and why one should use one timed automata model checker over another.

2.10.1 HyTech

Hytech \cite{HHWt97} is a model checking tool designed for the analysis of hybrid systems. A hybrid system is a system consists of a collection of digital programs that interact with each other and with an analog environment. Examples of hybrid systems include manufacturing controllers, flight controllers, medical equipment, and robots. In Hytech, systems are described as collections of automata with discrete and continuous components, and temporal
requirements are verified by symbolic model checking techniques. A HyTech input file consists of two parts. The first part contains the textual description of the system as hybrid automata, which are automatically composed for the analysis. The second part contains a sequence of analysis commands. The programming language of Hytech provides some primitive data types such as “state assertion” with a variety of operations, including Post, boolean operators, and existential quantification. One can also describe synchronisation between processes, the urgency of transitions, and generate a counterexample if the specification fails. HyTech has been used successfully to analyse a number of case studies including control-based applications such as distributed robot controller [HhH95], a robot system in manufacturing [HhH95], and the Philips audio control protocol [HWt95].

2.10.2 Kronos

Kronos [DOTY96] is a TCTL model checker based on timed automata model. Kronos supports both forward and backward TCTL model checking, and counterexample generation. One of the limitations of KRONOS is that its input language supports a very restricted data types that allow only the declaration of clock variables. Several successors of Kronos have been developed such as open-Kronos which provides a more convenient modelling language than Kronos (e.g., it supports discrete variables). Kronos has been used successfully to verify real-time communication protocols such as CSMA/CD, FDDI protocols, Philips audio control protocol, timed asynchronous circuits, and hybrid systems. It is freely available at http://www-verimag.imag.fr/DISTTOOLS/TEMPO/kronos.
2.10.3 UPPAAL

UPPAAL [BDL04] is a model checker for real-time systems developed in conjunction by Uppsala University, Sweden, and Aalborg University, Denmark. It extends the basic timed automata with features for concurrency, communication, data variables, and priority. The current version of the tool is 4.1.19 and it is freely available at http://www.uppaal.com. UPPAAL uses a dense-time model to describe systems, where each clock variable evaluates to a real number. A UPPAAL model is a parallel composition of all of its timed automata. All automata start at its initial state (location) and run independently of each other unless synchronization with other automata is required. A transition is enabled when all enabling conditions are evaluated to true and all the synchronization statements are executed. If more than one transitions are enabled, one of them is chosen non-deterministically. In addition to binary synchronisation, UPPAAL supports also urgent and broadcast synchronisation. Synchronisation on urgent channels should occur as soon as both components are enabled. Note that in transitions with urgent channels, guards can not have clock constraints. A broadcast channel allows a process (component) to synchronise with more than one component at the same time. Note that broadcast channels in UPPAAL are non-blocking in the sense that if a process has a transition with $a!$, where $a$ is declared as a broadcast channel, then $a!$ can be performed even when no $a?$ action is enabled on any of the processes of the system.

UPPAAL uses a client-server architecture which splits the tool into a graphical user interface (client) and a model checking engine (server). The user interface consists of three main sections: system editor, simulator, and verifier. The editor allows the user to model the system as a network of timed automata. The simulator gives the user the capability to interactively run the system to check if there are some trivial errors in the system design. The verifier allows the user to enter the properties to be verified in a restricted language of TCTL.
UPPAAL can verify safety, bounded liveness, and reachability properties.

For timed systems that have complex modeling details that require a very rich modeling language to capture them, we recommend using the UPPAAL tool since it has richer expressiveness in modeling systems than all other available tools (i.e. it supports a large class of data variables including arrays.)

2.10.4 RED

RED [Wan04c] stands for (Region-Encoding Diagrams) is a TCTL model checker for real-time systems. This model checker has been designed and developed by Farn Wang and colleagues at the National Tawian University. The current version of the tool is 5.0. The RED tool is available for free at http://cc.ee.ntu.edu.tw/ farn. An interesting feature of the RED model checker is that it is totally based on symbolic technology with BDD-like diagrams. RED now supports varieties of data-structures, which can be described as follows.

- Timed automata with CRD (Clock-Restriction Diagram) technology,
- linear hybrid automata with HRD (Hybrid-Restriction Diagram) technology,
- MDD-based diagrams which stands for multi-value decision diagrams and is very much like BDD except that variables are now decimal and arcs are labeled with value intervals.

In RED, systems are described as parametrized communicating timed automata (CTA), where processes can be model processes, specification processes, or environment processes. In a system with n processes, the user invokes the RED model checker via telling it which processes are for the model, and which for the specification. The remaining processes will be for the environment. Since the automata in RED are parametrised automata (i.e. we can
pass parameters to them) then we can declare many process automata with the same automaton template and identify each process automaton with a process index. RED supports both forward and backward analyses, deadlock detection, and counter-example generation. In RED, users can declare global and local variables of type boolean, discrete, clock-restriction variable, and hybrid-restriction variable. RED uses an extended version of TCTL and it supports strong and weak fairness assumptions of CTA.

For real-time systems that have complex timing properties, we recommend using the RED tool since it supports a full TCTL language which allows to express a wide variety of timed properties.

2.10.5 Rabbit

Rabbit [BLN03] is a model checking tool for real-time systems. The theoretical foundation of the tool is mainly based on timed automata extended with concepts for modular modeling. This tool has been developed by Dirk Beyer and his research group at the University of Passau, Germany. The current version of the tool is 2.1. The tool is freely available at http://www.sosy-lab.org/ dbeyer/Rabbit. The tool deals with timed systems using BDD data structure and uses some heuristic techniques to compute good variable orderings of the model and an estimate of the BDD size. We give an informal description of the formalism of Cottbus Timed Automata (CTA), which is used in the modelling language of Rabbit.

A CTA system consists of a set of modules that can be defined in a hierarchical way. One of them (the one at the top of the hierarchy) is used to model the whole system. The other modules are used as templates. Each module in the system model should have the following components:

- An identifier. Identifiers are used to name the modules within the system description. Using identifiers we can create several instances of the
modules associated with these identifiers.

- **An Interface.** The interface of a module contains the declarations of the variables that are used in that module. In a CTA module, we can declare clock variables, discrete variables, and synchronisation labels.

  - *Synchronisation labels.* Sometimes called signals which are used to synchronise timed automata that exist in different modules in the system. The concept of synchronisation labels in modules is very similar to the concept of events in CSP.

  - *Variables.* Rabbit allows us to declare both continuous (clock) variables and discrete variables. The values of these variables can be updated using assignment statements in the transition rules of the automaton.

- **A timed automaton.** Each module contains a timed automaton. The automaton consists of a finite set of states, a finite set of transitions, and a set of synchronisation labels.

- **Initial condition.** This is a formula over the module variables and the states of the module’s automaton, which specifies the initial state of the module.

- **Instances.** In CTA model, a module can contain instances of the other defined modules in the model. This is useful when we model systems containing subsystems, and that these subsystems occur several times in a system.

Since Rabbit supports modular modelling that allows us to represent systems components in a hierarchy way we recommend using it when we have a system where its components have different levels of hierarchy.
2.10.6 PRISM

PRISM [KNP11] is a probabilistic model checker that has been developed for the purpose of modelling and analysis of systems that exhibit random or probabilistic behaviour. The tool has been designed and developed in the University of Birmingham, UK. It supports both discrete-time and dense-time analysis. PRISM has been used to analyse systems from many different application domains, including communication and multimedia protocols, distributed algorithms, security protocols, and biological systems.

PRISM can build and analyse several types of probabilistic models:

- discrete-time Markov chains (DTMCs)
- continuous-time Markov chains (CTMCs)
- Markov decision processes (MDPs)
- probabilistic automata (PAs)
- probabilistic timed automata (PTAs)

PRISM allows for automated analysis of a wide range of quantitative properties of systems, e.g. “what is the probability of a failure causing the system to shut down within 4 hours?”. PRISM incorporates state-of-the art symbolic data structures and algorithms, based on BDDs and MTBDDs (Multi-Terminal Binary Decision Diagrams) [KNP04]. Since PRISM supports probabilistic model checking of timed systems we recommend using it when the given system has some interesting probabilistic behaviour that need to be considered while trying to prove the correctness of the system.

2.11 Conclusion

The rich and powerful theoretical foundation of the timed automata model makes it a popular choice for the formal verification of real-time systems.
The Alur-Dill timed automata model has not been used only for correctness verification but also for several other interesting and useful applications such as scheduling, code synthesis, optimal resource analyses, parametric analysis, quality assurance, Worst-Case Execution Time analysis, and much more. Several tools have been developed based on that model such as UPPAAL, Kronos, RED, Rabbit, and VerICS. However, only a few number of them such as UPPAAL and RED have been actively improved and evolved for a long period of time.

Real-time temporal logics have been proposed as a formalism for describing systems in which timing information plays a crucial factor for their correctness. Two extensions of temporal logic have been proposed for the purposes of specifying real-time properties, the timed linear temporal logic such as MTL and TPTL, and the timed branching temporal logic such as TCTL. Even though MTL and TCTL are expressively incomparable from a theoretical point of view, from a practical point of view MTL is considered to be more expressive than TCTL. Model-checkers, whose requirements languages are based on real-time branching-time logics, cannot express a variety of interesting timed properties including those which contain multiple timing constraints. On the other hand, timed linear temporal logic such as MTL can straightforwardly express multiple timing constraints on runs, which are sometimes interesting when analysing the correctness of real-time systems. There are many interesting computer applications where their correctness depend on satisfying multiple timing constraints. Real-time control systems are typical examples of systems that have multiple timing constraints such as traffic light control systems, and nuclear control systems. These systems can not be analysed correctly using any of the available real-time model checking tools such UPPAAL, Kronos, and RED. This is in fact a solid justification why we are looking to have a model checking tool based on real-time linear temporal logic such as MTL logic.
The difference in the complexity of timed linear and timed branching model checking has been viewed as an argument in favor of the timed branching paradigm. In particular, the computational advantage of TCTL model checking over MTL and TPTL model checking makes TCTL a popular choice, leading to efficient model-checking tools for this logic [BY04, DOTY96, Wan04c]. Through the 1990s, the dominant real-time temporal specification language in industrial use has been TCTL logic.

However, a large body of work has been done on studying Metric Temporal Logic, but there are currently no model checking tools implementing MTL logic. There is still a lack of efficient decision procedures for MTL logic. However, we believe that several theoretical works need to be done first before we can have a practical model checking tool based on MTL logic. No real data structures do exist for this type of logic. A new efficient decision procedure, abstraction techniques, and model checking algorithm for MTL logic should also be investigated. We suspect that it could be possible and feasible to have a tool for verifying real-time systems based on some expressive fragments of MTL logic that have been shown to be decidable with a reasonable complexity.
Chapter 3

The WCET Problem of Timed Automata

In this chapter we show first how real-time systems can be modeled as parallel compositions of timed automata [Alu98]. We then give a definition of cycles (loops) in TA and discuss the cases where WCET can be infinity. We discuss also some of the challenges that may arise when analysing WCET of TA with cyclic behaviour using model checking. Finally, we summarise the previous methods and approaches and the existing tools for verifying WCET of systems.

3.1 Networks of Timed Automata

To model concurrent systems, timed automata is often extended with parallel composition, giving networks of timed automata. The product automaton, i.e. the automaton describing the combined system, can be defined as follows.

Definition 3.1.1. Let $A_1 = (\Sigma_1, L_1, L_0^1, L_F^1, X_1, I_1, E_1)$ and $A_2 = (\Sigma_2, L_2, L_0^2, L_F^2, X_2, I_2, E_2)$ be two timed automata. Assume that the clock sets $X_1$ and $X_2$ are disjoint. Then the parallel composition of the two automata, denoted by $A_1 || A_2$ is the timed automaton $(\Sigma_1 \cup \Sigma_2, L_1 \times L_2, L_0^1 \times L_0^2, L_F^1 \times L_F^2, X_1 \cup X_2, I, E)$,
where \( I(l_1, l_2) = I(l_1) \land I(l_2) \) and the transition relation \( E \) is given by the following rules:

1. For \( a \in \Sigma_1 \cap \Sigma_1 \), if \((l_1, a, \varphi_1, \lambda_1, l'_1) \in E_1 \) and \((l_2, a, \varphi_2, \lambda_2, l'_2) \in E_2 \), then \( E \) will contain the transition \((l_1, l_2, a, \varphi_1 \land \varphi_2, \lambda_1 \cup \lambda_2, (l'_1, l'_2))\).

2. For \( a \in \Sigma_1 - \Sigma_2 \), if \((l, a, \varphi, \lambda, l') \in E_1 \) and \( t \in L_2 \), then \( E \) will contain the transition \((l, t, a, \varphi, \lambda, (l', t))\).

3. For \( a \in \Sigma_2 - \Sigma_1 \), if \((l, a, \varphi, \lambda, l') \in E_2 \) and \( t \in L_1 \), then \( E \) will contain the transition \((t, l, a, \varphi, \lambda, (t, l'))\).

A state of the parallel composition is a pair \( \langle l, v \rangle \), where \( l \) denotes a vector of current locations of the network (i.e. one for each process in the network), and \( v \) is a clock assignment that remembering the current values of the clocks in the system. The invariant of the compound location is the conjunction of the invariants of the automaton locations. There will be a transition in the parallel composition for each pair of transitions from the individual timed automata with the same action. The source location of the transition will be the composite location obtained from the source locations of the individual transitions. The target location will be the composite location obtained from the target locations of the individual transitions. The guard will be the conjunction of the guards for the individual transitions, and the set of clocks that are reset will be the union of the sets for the individual transitions. If the action of a transition is only an action of one of the two processes (automata), then there will be a transition in the parallel composition for each location of the other timed automaton.

**Definition 3.1.2.** A run of a timed automaton \( \mathcal{A} = (\Sigma, L, L_0, L_F, X, I, E) \) with an initial state \((l_0, v_0)\) over a timed trace \( \zeta = (t_1, a_1), (a_2, t_2), ... \) is a sequence of transitions of the form

\[
\langle l_0, v_0 \rangle \xrightarrow{d_1} a_1 \xrightarrow{l_1} v_1 \xrightarrow{d_2} a_2 \xrightarrow{l_2} v_2, ...
\]

satisfying the condition \( t_i = t_{i-1} + d_i \) for all \( i \geq 1 \) and that \( l_0 \in L_0 \).
3.2 The WCET of Timed Automata

Worst-case execution time (WCET) analysis is a major technique in verification of real-time systems, i.e. ensuring that the system will perform a given task not exceeding a given time limit. WCET of a system’s tasks is also required for schedulability analysis of the system. For timed automata the problem can be defined as follows. Given a timed automaton \( A \) with a start location \( l_s \in L_0 \) and a final location \( l_f \in L_F \), the WCET problem asks to compute an upper bound on the time needed to reach the final location \( l_f \) from the start location \( l_s \). Since the locations of an automaton are decorated with a delay-quantity and that transitions between locations are instantaneous, the delay of a timed execution is simply the sum of the delays spent in the visited locations.

**Definition 3.2.1.** (Delay of a run.) Let \( r = \langle l_0, v_0 \rangle \xrightarrow{d_1} a_1 \langle l_1, v_1 \rangle \xrightarrow{d_2} a_2 \langle l_2, v_2 \rangle, \ldots \) be a run in the set of runs \( \mathcal{R} \). The delay of \( r \), \( \text{delay}(r) \), is the sum \( \sum_{i=1}^{n} d_i \), where \( n \) can be infinity. Hence, the problem of computing the BCET and WCET of \( A \) can be formalized as follows

\[
\begin{align*}
BCET(A) &= \inf_{\forall r \in \mathcal{R}^f} \text{delay}(r) \\
WCET(A) &= \sup_{\forall r \in \mathcal{R}^f} \text{delay}(r)
\end{align*}
\]

where \( \mathcal{R}^f \) is a subset of \( \mathcal{R} \) which reach a final location. Of course, a valid WCET bound is \([0, \infty]\), and WCET can be infinity if there is an infinite non-zeno run (an infinite run in which time can diverge) [BG06, Góm06]. This can happen if there is a reachable cycle or loop that can be repeated infinitely often and that time can elapse between iterations.

3.2.1 Cycles in Timed Automata

The WCET problem is easy to solve in the case of acyclic TA (i.e. TA contains no cycles), but cycles might introduce unbounded WCET, that needs to be
detected on-the-fly during the analysis. We give now a formal definition of cycles in TA and discuss the different types of cycles in TA and the cases where the WCET of an automaton can be infinity.

**Definition 3.2.2. (Cycles in TA).** A cycle in a timed automaton is a finite sequence of edges where the source location of the first edge in the sequence is the target location of the last edge in the sequence. Let $A = (\Sigma, L, L_0, L_F, X, I, E)$ be a timed automaton and let $m$ be a natural number such that $m \geq 1$. We say that a sequence $(e_0, e_1, ..., e_{m-1}) \in E^m$ is a cycle if $\text{trg}(e_i) = \text{src}(e_{i+1})$ for all $0 \leq i < m - 1$ and $\text{trg}(e_{m-1}) = \text{src}(e_0)$.

**Definition 3.2.3. (Types of Cycles).** Let $E_\pi = (e_0, ..., e_{m-1})$ be the sequence of edges of a cycle $\pi$. Let $\text{delay}(\pi, j) = \sum_{i=0}^{m-1} (d_i^j) = \triangleright a, b \sqsubset$ be a function that computes the delay interval of the symbolic cycle $\pi$ at a given iteration $j$ (note that any two consecutive visits to the start location of the cycle represents a full iteration of the cycle), and $d_i^j$ is the delay interval that can be spent at location $\text{src}(e_i)$ at iteration $j$, and $\triangleright \in \{[, ], (,)\}$ and $\triangleleft \in \{,), ][\}$. Recall that in TA model checking we deal with symbolic states (i.e. states of the form $(l, D)$ where $l$ is a control location and $D$ is the clock zone at that location (see Definition 2.9.5)) and hence the delay interval of a cycle can take any of the following forms $[a, b], (a, b], [a, b)$ or $(a, b)$, where $0 \leq a, b \leq \infty$. However, we write $\triangleright a, b \sqsubset$ to indicate that the interval can be any of the above four forms. We can then classify cycles based on their topologies as follows.

- **(Self-cycles).** We say that the set of edges $E_\pi = \{e_0\}$ is a self cycle if $\text{trg}(e_0) = \text{src}(e_0) = l$.

- **(Simple cycles).** We say that the sequence $(e_0, e_1, ..., e_{m-1})$ is a simple cycle if $\text{trg}(e_i) = \text{src}(e_{i+1})$ for all $0 \leq i < m - 1$ and $\text{trg}(e_{m-1}) = \text{src}(e_0)$ and when $i \neq j \Rightarrow e_i \neq e_j$ for all $0 \leq i, j < m$.

---

1A cycle $\pi$ is a symbolic cycle if the delay that can be spent at each location of the cycle is described as a clock constraint of the form $\varphi := x \sim c | \varphi \wedge \varphi$, where $\sim \in \{<, \leq\}$ and $c$ is a constant, and hence the delays at locations will be represented as intervals rather than concrete values.
3.2. **THE WCET OF TIMED AUTOMATA**

- **(Composite cycles).** We say that the sequence \((e_0, e_1, ..., e_{m-1})\) is a composite cycle if \(\text{trg}(e_i) = \text{src}(e_{i+1})\) for all \(0 \leq i < m - 1\) and \(\text{trg}(e_{m-1}) = \text{src}(e_0)\) and there is some \(i, j\) such as \(e_i = e_j\) for some \(0 \leq i, j < m\).

Cycles can be also classified based on their time behaviour as follows.

- **(Realizable cycles).** We say that \(\pi\) is a realizable cycle if time can elapse at each iteration of the cycle. Formally, we say that \(\pi\) is a realizable cycle if \(\text{delay}(\pi, j) = \triangleright a, b\triangleleft\) and \(a, b > 0\) for all \(j \geq 2\). We call runs containing such class of cycles as strong non-zeno runs.

- **(Non-realizable cycles).** We say that \(\pi\) is non-realizable cycle if time can not elapse at some iteration of the cycle. Formally, we say that \(\pi\) is a non-realizable cycle if \(\text{delay}(\pi, j) = \triangleright 0, 0\triangleleft\) for any \(j \geq 2\).

Note that since the clocks in TA models are real-valued variables whose values are represented as symbolic constraints using zones, a particular symbolic run of a TA model can represent an infinite number of concrete runs. It is possible then to have a symbolic cycle \(\pi\) where \(\text{delay}(\pi, j) = \triangleright 0, b\triangleleft\) and \(b > 0\). Such cycles can not be considered neither as realizable nor as non-realizable. We call the runs containing such class of cycles as weak non-zeno runs. However, infinite weak non-zeno runs lead to an infinite WCET. Cycles can be classified also into cycles with constant delays and cycles with non-constant delays.

- **(Cycles with constant delays).** We say that \(\pi\) is a cycle with constant delay if the delay (duration) of the cycle at each iteration is constant. Formally, we say that \(\pi\) has constant delays if for any two distinct iterations \(j, k\) such that \(j, k \geq 2\) we have \(\text{delay}(\pi, j) = \triangleright a_j, b_j\triangleleft\) and \(\text{delay}(\pi, k) = \triangleright a_k, b_k\triangleleft\)′ and that \(a_j = a_k\) and \(b_j = b_k\) and \(\triangleright = \triangleright\)′ and \(\triangleleft = \triangleleft\)′.
(Cycles with non-constant delays). We say that $\pi$ is a cycle with non-constant delay if the delay of the cycle can vary between iterations. Formally, we say that $\pi$ has non-constant delays if for some distinct iterations $j,k$ such that $j,k \geq 2$ we have $\text{delay}(\pi,j) = \triangleright a_j, b_j \triangleleft$ and $\text{delay}(\pi,k) = \triangleright' a_k, b_k \triangleleft'$ and that $a_j \neq a_k$ or $b_j \neq b_k$ or $\triangleright \neq \triangleright'$ or $\triangleleft \neq \triangleleft'$.

Cycles can be also classified based on the maximum number of times they can be iterated (let us denote it as $N$) into finite cycles (i.e. cycles that can be iterated a finite number of times where $N < \infty$) and infinite cycles (i.e. cycles that can be iterated infinitely often where $N = \infty$).

(Finite cycles in TA). We say that the cycle $\pi$ with the sequence of edges $E_\pi = (e_0, e_1, \ldots, e_{m-1})$ is a finite cycle if at some iteration $1 \leq j \leq N$ and at some $0 \leq k \leq m - 1$ the check $\text{consistent}(\text{succ}(Z_j^k, e_k))$ fails, where $\text{consistent}(Z)$ is a boolean operation that checks whether the zone $Z$ that results from executing a particular edge is consistent (is not empty) as described in Definition 2.9.6.

(Infinitely cycles in TA). We say that the cycle $\pi$ with the sequence of edges $E_\pi = (e_0, e_1, \ldots, e_{m-1})$ is an infinite cycle if there is no iteration $j$ where the check $\text{consistent}(\text{succ}(Z_j^k, e_k))$ fails for any $0 \leq j \leq m - 1$.

As we will see later, distinguishing between different types of cycles is important for the efficiency of the WCET analysis since identifying the type of the cycle under analysis may help in accelerating its WCET computation. For example, if the cycle under analysis is a non-realizable cycle then the search can skip that cycle since time can not elapse during the execution of the cycle. On the other hand, if the cycle under analysis is a realizable cycle that can be iterated infinitely often then the search can stop immediately since the WCET will be infinity. Note that we skip iteration one when defining realizable and non-realizable cycles and when defining cycles with constant delays.
and cycles with non-constant delays since the generated zones at iteration one can be arbitrary zones. This is because the starting zone at iteration one is not obtained from the computations of the cycle itself and hence using the zones that result from the first iteration of the cycle during the analysis may yield to wrong conclusions about the behaviour of the cycle in the sense that one may conclude that the cycle is non-realizable while it is realizable or vice versa.

It is interesting to note also that not every infinite cycle (i.e. infinitely repeating cycle) can lead to an infinite WCET. For instance, the automata in figures 6 and 7 represent automata with infinite cycles, where the automaton in figure 6 has a WCET of 1, while the automaton in figure 7 has an infinite WCET. However, recognising infinite cycles and determining whether they lead to an infinite WCET is extremely important for the efficiency of the WCET analysis since the search can stop immediately if it detects that the cycle under analysis can lead to an infinite WCET. We discuss in the following chapters how one can detect on-the-fly infinite cycles and determine whether they lead to an infinite WCET. We now turn to discuss the cases where the WCET of an automaton can be infinity.
As we discuss in the following lemma the existence of a single reachable unconstrained location in $A$ (i.e. a location that is not guarded with an invariant) is sufficient to conclude that the least upper bound for termination of $A$ is infinity since in this case the automaton can stay at that location an arbitrary amount of time. Similarly, the existence of an infinite strong or weak non-zeno run in $A$ is sufficient to conclude that the least upper bound for termination of $A$ is infinity.

**Lemma 3.2.4.** Let $A = (\Sigma, L, L_0, L_F, X, I, E)$ be a timed automaton. Then $\text{WCET}(A) = \infty$ if one of the following conditions holds:

1. There is an unconstrained location in $A$ (that is not a final location).
2. There is an infinite strong or weak non-zeno run in $A$.

**Proof.** – (Case 1). Suppose that $L$ represents the set of reachable locations of $A$. Suppose further that we have an unconstrained location $l \in L$ such that $l \not\in L_F$, where $L_F$ is the set of final locations of $A$. From the assumption that $l$ is an unconstrained location we know that all the lower bounds of the clocks in the DBM representing the clock invariant at location $l$ have the form $D_{0,j} = (0, <)$, and all the upper bounds of the clocks have the form $D_{j,0} = (\infty, <)$, for all $j = 1..n$, where $n$ represents the number of clocks in $A$. From the definition of the zone approach and following steps (1-8) at Section 2.10 then the diagonal constraints involving the clock $\delta$ will be of the form $D_{i,j} = (\infty, <)$ regardless of the form of the guards at the outgoing edges of location $l$. Hence, $D_{i,0}$ will be of the form $(\infty, <)$ due to the canonicalization steps, where $i$ is the index of the clock $\delta$. From the definition of WCET (see Definition 3.2.1) we know then that WCET of the automaton in this case will be infinity.

– (Case 2). To prove this case we need to consider two sub-cases. The first sub-case is when there is an infinite strong non-zeno run in $A$ and the second sub-case is when there is an infinite weak non-zeno run in $A$. 
3.2. THE WCET OF TIMED AUTOMATA

* Suppose that \( r \in \mathcal{R} \) represents an infinite strong non-zeno run in \( A \). From the assumption that \( r \) is an infinite strong non-zeno run we know that \( r \) contains an infinite realizable cycle. From the definition of realizable cycles we know that the delay of the cycle between iterations has the form \( \triangleright a, b \triangleleft \) and \( a, b > 0 \) for all \( j \geq 2 \). From the assumption that the cycle is infinite we know that \( N = \infty \), where \( N \) is the number of iterations that the cycle can be repeated. From the definition of the WCET (see Definition 3.2.1) and given that \( N \) is infinity and \( b > 0 \) we know then that WCET of the automaton in this sub-case will be infinity.

* Suppose that \( r \in \mathcal{R} \) represents an infinite weak non-zeno run in \( A \). From the assumption that \( r \) is an infinite weak non-zeno run we know that \( r \) contains an infinite cycle. From the definition of weak non-zeno run we know that the delay of the cycle between iterations has the form \( \triangleright 0, b \triangleleft \) and \( b > 0 \) for all \( j \geq 2 \). From the assumption that the cycle is infinite we know that \( N = \infty \). From the definition of WCET (see Definition 3.2.1) and given that \( N \) is infinity and \( b > 0 \) we know then that WCET is this sub-case will be infinity.

\[ \square \]

3.2.2 Some Challenges in Verifying WCET of TA with Cyclic Behaviour

In general, WCET analysis is undecidable (equivalent to halting problem), which states that it is undecidable to determine whether or not an execution of a program will eventually halt. However, for TA models one can use model-checking techniques to analyse the system and compute the WCET. We discuss here some of the challenges that may arise when analysing WCET of TA with cyclic behaviour.
Verifying WCET of systems with cyclic behaviour can cause the state explosion problem if we traverse all the cycles iterations during the analysis in particular when there is a largely repetitive finite cycle in the behaviour of the analysed system.

The efficiency of TA model checking technology relies mainly on the application of a combination of abstraction techniques such as the inclusion abstraction [DT98], the activity abstraction [DT98], and the extrapolation abstraction [DT98], which help to accelerate the reachability analysis of TA. However, these abstractions can harm the WCET analysis (i.e. they might introduce coarse over-approximation or coarse under-approximation on the value of WCET) if not applied carefully. For example, the inclusion abstraction can lead to stop the exploration when some zone is “included” inside an already explored zone. This way of cutting exploration of state space based on inclusion is correct for reachability properties, but not when properties critically involve detection of cycles in the timed automaton. The question is then how one can benefit from these abstractions without adversely affecting the WCET computations.

Identifying how many times the cycle can iterate is a complicated function of the active clocks of that cycle (i.e. the set of clocks that appear in the guards and in the invariants of the cycle). In certain cases, the worst-case execution time of the cycle is easy to see by inspection. However, there are cases where the maximum number of iterations of a cycle is not obvious and complex checks may be needed to identify precisely the maximum number of iterations the cycle can be repeated.
3.3 Related Work

In this section we summarise the previous methods and approaches and the existing tools for verifying WCET of systems. We then describe the previous model checking algorithms for verifying WCET. Finally, we discuss some of the acceleration techniques that have been developed to improve the performance of model checking systems with cyclic behaviour.

3.3.1 Methods and Tools for the WCET Problem

The reader is referred to [WEE+08] for an exhaustive presentation of WCET computation techniques and tools. In general, there are two main classes of methods for computing WCET.

- Testing-based methods. These methods are based on experiments i.e., running the program on some data, using a simulator of the hardware or the real platform. The execution time of an experiment is measured and, on a large set of experiments, maximal and minimal bounds can be obtained. A maximal bound computed in this way is unsafe as not all the possible paths have been explored. These methods might not be suitable for safety critical embedded systems but they are versatile and rather easy to implement. Mtime [RPW08] is a measurement tool that implements this technique.

- Verification-based methods. These methods often rely on the computation of an abstract graph, the control flow graph (CFG), and an abstract model of the hardware. Together with a static analysis tool they can be combined to compute WCET. The CFG should produce a super-set of the set of all feasible paths. Thus the largest execution time on the abstract program is an upper bound of the WCET. Such methods produce safe WCET, but are difficult to implement. Moreover, the abstract
program can be extremely large and beyond the scope of any analysis. In this case, a solution is to take an even more abstract program which results in drifting further away from the exact WCET. Although difficult to implement, there are quite a lot of tools implementing this scheme: Bound-T [30], OTAWA [BCRS10], TuBound [PSK08], Chronos [LLMR07], SWEET [EES+03] and aiT [FHW04] are static analysis-based tools for computing WCET.

The verification-based tools mentioned above rely on the construction of a control flow graph, and the determination of loop bounds. This can be achieved using user annotations (in the source code) or sometimes inferred automatically. The CFG is also annotated with some timing information about the cache misses/hits and pipeline stalls, and paths analysis is carried out on this model e.g., by Integer Linear Programming (ILP).

3.3.2 Model Checking and WCET Problem

It is claimed in [Wil04] that model checking is inadequate for WCET analysis. However, in [Met04] Metzner has shown that model checking can be used efficiently for WCET analysis. He used model checking to improve WCET analyses for hardware with caching.

In [BLR05a] Behrmann et al provide zone-based algorithms for parameter synthesis for two strict forms of TCTL properties: (1) $\text{AF}_{\leq p} \phi$ and (2) $\text{AG}(\psi \Rightarrow \text{AF}_{\leq p} \phi)$. The first form can be used to calculate the WCET of the given TA model. However, it is not clear to us how this approach can be used to handle TA with infinite cycles and whether it can detect the cases where the WCET is infinity. Moreover, the approach presented in [BLR05a] can not handle efficiently TA containing cycles, in particular largely repetitive cycles.

The work in [BFH*01] uses a variant of timed automata called “priced timed
3.3. RELATED WORK

automata" and the DBM data structure to compute the minimum cost of reaching a goal state in the model. A priced timed automata can associate costs with locations, where the costs are multiplied by the amount of time spent in a location. An automata may be designed so that the total cost corresponds to the execution time, and thus this approach may be used to calculate the best case execution time problem. However, the WCET problem is different than the BCET problem and needs special treatment during the analysis in particular when there are cycles in the behaviour of TA.

There are very few model checking tools that support WCET computation. For example, the model checker UPPAAL [BDL04] supports operators called $\inf$ and $\sup$, which can be used respectively to compute the lower and upper bound for termination of TA. However, we show that in certain circumstances, when infinite cycles exist, the operators may not terminate, and we provide examples which UPPAAL fails to verify, and when largely repetitive finite cycles exist, the operators suffer from the state space explosion, thus leading to a low efficiency or resource exhaustion. In fact, the $\sup$ operator can not handle efficiently TA with (infinite) cycles. On the other hand, our algorithms can work properly on any arbitrary diagonal-free TA and can handle efficiently infinite cycles and some interesting forms of largely repetitive finite cycles.

The use of timed automata (TA) and the model-checker UPPAAL for computing WCET on pipelined processors with caches was reported in [DOT+10] where the METAMOC method is described. METAMOC consists in: 1) computing the CFG of a program, 2) composing this CFG with a (network of timed automata) model of the processor and the caches. Computing the WCET is then reduced to computing the longest path (timewise) in the network of TA.

In [HS09], B. Huber and M. Schoeberl consider Java programs and compare ILP-based techniques with model-checking techniques using the model-checker UPPAAL. Model-checking techniques seem slower but easily amenable to changes (in the hardware model). The recommendation is to use ILP tools
for large programs and model-checking tools for code fragments.

Symbolic evaluation of the longest duration of time spent in a dense set of executions was addressed in [Vic98] using Time Petri Nets, which are different in the formulation of the TA model but result in equivalent problems for zone-based solution. Several tools have been developed based on the Time Petri Nets using the zone construction such as Tina [BB04], Romeo [GLMhR05], and Oris [BCRV10].

### 3.3.3 Model Checking and Acceleration Techniques

We summarise now some of the works that have been done to accelerate the verification process of systems with cyclic behaviour using model checking. The work in [BW94] proposes acceleration techniques of symbolic verification of systems that are modeled by a discrete control graph with unbounded integer variables. Static analysis is used to detect interesting cycles, of which the result of iterated execution can be computed by one single meta transition. These meta transitions are then added to the system and favored by the state space exploration of the state space.

Symbolic techniques using *queue-content decision diagrams*, for the analysis of communication protocols that are modeled by finite-state machines that communicate through unbounded FIFO-queues, also use meta transitions to accelerate the exploration of the state space [BG99, BGWW97]. Special cycles in the control graph, i.e. the repeated receiving of messages from a channel, are associated with meta transitions that compute all states that are reachable by the iterative execution of the cycle.

The most closely-related work to our proposed acceleration techniques in Chapter 7 is the work on loop acceleration for UPPAAL done by Hendriks and Larsen [HL02]. However, their work is quite different from ours in technique and objectives. In particular, their technique is particularly tailored
to verifying correct controllers. Moreover, we are interested in accelerating WCET analysis of finite and infinite cycles in TA. Their work was developed in response to a problem verifying the correctness of a controller that conducted a busy-waiting loop during the course of a long-running process in the controller’s environment. It was very expensive to recognize that the system would eventually reach the desired state – verification of an “eventually reachable” TCTL goal. Their solution involves adding loop acceleration edges to the model, allowing UPPAAL to detect the reachability more quickly, using breadth-first search, than with the original un-augmented model. Our technique, by contrast, works by collapsing together parts of the state space that are equivalent with respect to the WCET analysis, making exhaustive search faster.

Hendriks and Larsen’s technique also works only in the case where a loop is concerned only with a single clock. That is, where all the guards and invariants involve only a single clock, and where the value of that single clock increases monotonically. By contrast, in our loops there are typically multiple clocks racing against each other, and clocks are reset since the loop involves the controller repeatedly servicing some process in the environment.

Acceleration techniques improving the efficiency in the construction of the zone graph was addressed in the context of Time Petri Nets [HB11, BGR09]. Reduction of the complexity for the derivation of the canonical form of a DBM through the Floyd-Warshall algorithm, so as to go under the limit of $O(n^3)$, has been reported in various work [RC94, Vic98, ZLZ05]. However, the reduction to $O(c \times n)$ proposed in this thesis is new. As we shall show in Chapter 6 the new canonicalization algorithm is not specific to the WCET problem, and has a more general interest in the reachability analysis of TA models.

Other researchers have recently presented approaches to acceleration in different, but related contexts. For example, Fietzke et al [FKW12] have shown
how to accelerate loops involving a single clock in a variation of timed automata they call “Extended Timed Automata”. Bozga et al. [BIK10] present theoretical work on identifying periodic relations in model checking and an implementation in their FLATA toolset. However, they are interested in infinitely periodic loops, and are working with integer programs, rather than timed automata. Bardin et al have developed a general theory of acceleration in model-checking [BFLS05], that includes loop acceleration as a special case of what they call “flat acceleration”. They have applied it to integer programs in the FAST tool [BLP06]. The work by Ben Salah on partial order optimization for timed automata [Sal07]. However, the scope and the goal of their optimization is different since their optimization intends to accelerate the reachability analysis of TA and not the WCET analysis.
Chapter 4

WCET of Acyclic and Finite Cyclic Timed Automata

The chapter provides a solution to the fundamental problem of computing the shortest and the longest time taken by a run of a timed automaton from an initial state to a final state. It does so using the difference-bound matrix data structure to represent zones, which is a state-of-the-art heuristic to improve performance over the classical (and somewhat brute-force) region graph abstraction. The solution provided here is conceptually a marked improvement over some earlier work on the problems [Met04, DOT+10], in which repeated guesses (guided by binary search) and multiple model checking queries were effectively but inelegantly and less efficiently used; here only one run of the zone construction is sufficient to yield the answers. However, the applicability of the proposed solution in this chapter is limited to TA without infinite runs. The chapter then reports on a prototype implementation of the algorithms using Difference Bound Matrices (DBMs), and presents the results of its application on a realistic automatic manufacturing plant.
4.1 Introduction

As a first contribution of this chapter, we give an efficient zone-based algorithm for computing BCET and WCET using Difference Bound Matrices (DBMs). Similar to [BBD03, TLR08, HPRV12] the solution adds an extra global clock that acts as an observer and then computing the zone graph of the automaton, by means of a standard forward analysis using DBMs. One of the outcomes of the analysis is that the correctness of the computation of BCET and WCET depends mainly on the way we extrapolate (abstract) the zones in the resulting zone graph. It is well-known that extrapolation is a necessary step in order to guarantee termination in timed automata [Rok93, BY04]. However, the standard approaches for analysing a timed automaton that depend on computing the zone graph of the automaton while extrapolating zones at each step of the successor computation [BY04], will give abstract zones and hence result in abstract values of the execution times. Therefore, the direct application of these approaches is inconvenient for the analysis of WCET.

To get precise and accurate values of execution times we choose to work with search trees whose nodes are non-extrapolated or real zones. The algorithms follow a new paradigm where zones are not abstracted, hence preserving the value of extra clock. Instead, abstraction is used to detect when a zone has already been explored, that is, when two non-abstracted zones have the same abstraction. When a final state is reached, the constraints on the extra clock in the zone yield expected BCET and WCET values. The proposed algorithms can successfully handle acyclic TA and TA containing cycles given that the automaton under analysis has no run of infinite length. Note that even on a TA without cycles, a WCET of infinity is still possible if not every reachable location is guarded by an invariant containing an upper bound on some clock. We then report on a prototype implementation of the algorithms using the model checker opaal [DHJ+11] and present the results of its application by computing the BCET/WCET of a realistic automatic manufacturing plant taken from [DY95] when analysing it under different configurations. We show that the proposed algorithms can outperform the conventional binary search approach used
4.2 Approaches for Verifying WCET of TA

In this section we describe two approaches for computing WCET of TA models. The first approach is the classical binary search approach which may take multiple runs of the zone construction to yield the answer. Then we describe a zone-based approach that can compute the WCET of the system in only one run of the zone construction.

4.2.1 The Binary Search Approach

We give here a description of the classical binary search approach [Met04, DOT+10] that can be used to find the WCET of a given system. Figure 2 presents a simple and straightforward algorithm for computing WCET of systems where the algorithm verifies repeatedly TCTL formulas until they hold. The algorithm takes as inputs a timed automaton \( A \) and returns the least upper bound for termination of \( A \).

The algorithm uses the TCTL formula \( \text{AG}(\delta \leq \text{guess}) \) to verify the least upper bound for termination of TA with the global variable \text{guess} which is initially set to 0. The formula represents a quantitative property whose verification requires numerical computations to be performed. In order to verify the formula we need first to substitute a value for \text{guess} then the property can be model checked. The formula needs to be verified repeatedly until we find the first value of \text{guess} at which the formula holds.

The time complexity of the algorithm is PSPACE-Complete. The result is mainly based on the well-known complexity of the model-checking problem for TCTL logic [ACD93], in which the authors have shown that reachability problem of timed automata is PSPACE-complete. Each iteration of the loop involves a model checking call \((\text{check}(A, \text{AG}(\delta \leq \text{guess})))\) to verify the formula \( \text{AG}(\delta \leq \text{guess}) \) against the
model $\mathcal{A}$ after introducing a new value for the variable $\text{guess}$. If the verification fails at iteration $i$, we go to iteration $i+1$ where we increase the value of the variable $\text{guess}$. In order to speed up the analysis process, the algorithm uses a mixture of brute-force or exhaustive search of doubling $\text{guess}$ (i.e. $\text{guess} = 1, 2, 4, 8, 16, \ldots$) until the formula is satisfied and then doing a binary search for the smallest value of $\text{guess}$. The goal is to reduce the number of times we call the model checker which represents the most expensive part of the algorithm. After determining the first time the formula $\Box G(\delta \leq \text{guess})$ holds using brute-force search of doubling $\text{guess}$, we use binary search to determine the smallest value of $\text{guess}$ at which the formula holds which will be the tightest WCET of the system. Note that in each step, the \texttt{binary_search} algorithm (see Figure 3) verifies whether the formula $\Box G(\delta \leq \text{guess})$ holds at $\text{guess}$ equals to the middle value of the interval $[\text{min}, \text{max}]$ and if it holds it repeats its actions on the sub-interval $[\text{min}, \text{mid}]$. Otherwise, if the formula fails at $\text{guess} = \text{mid}$, the algorithm then repeats its action on the second half of the interval $[\text{mid} + 1, \text{max}]$. Note that the algorithm continues to repeat itself until the remaining interval to check has the form $[\text{min} = x_i, \text{max} = x_i + 1]$ where $x_i \in \mathbb{N}$. The algorithm finally needs to verify whether the formula holds at $\text{guess} = x_i$, and if it holds then $x_i$ will be the tightest WCET, otherwise the tightest WCET will be $x_i + 1$.

\begin{verbatim}
Input: ($\mathcal{A}$)
  int guess := 0
  Bool sat := false
Output: guess
if \neg\text{check}($\mathcal{A}, \Box G(\delta \leq \text{guess})$) then guess := 1 else sat := true
while \neg sat do
  \{ if \text{check}($\mathcal{A}, \Box G(\delta \leq \text{guess})$) then \{sat := true; break\}
    double(guess)
  \}
if (\text{guess} = 0 \lor \text{guess} = 1 \lor \text{guess} = 2) return \text{guess}
int min := ( (\text{guess}/2)+1), max := \text{guess}
\text{guess} := \text{binary_search} ($\mathcal{A}, \Box G(\delta \leq \text{guess}), \text{min}, \text{max}$)
return \text{guess}
\end{verbatim}

\textbf{Algorithm 2:} Computing WCET using parametric TCTL
4.2. APPROACHES FOR VERIFYING WCET OF TA

int function binary_search (A, AG(δ ≤ guess), min, max)
{
    if (max - min = 0) return max
    if (max - min = 1)
    {
        guess := min
        if check(, AG(δ ≤ guess)) return min else return max
    }
    else
    {
        int mid := midpoint(min, max)
        guess := mid
        if check(, AG(δ ≤ guess)) then
            binary_search (, AG(δ ≤ guess), min, mid)
        else binary_search (, AG(δ ≤ guess), mid + 1, max)
    }
}

Algorithm 3: The Binary Search Algorithm

4.2.2 Some Limitations of The Binary Search Approach

Computing WCET of systems using conventional binary search approach has several major limitations which can be summarised as follows.

- The approach may require multiple runs of the zone construction before yielding the answer since it depends on augmenting the models with a clock variable and querying it while guessing an upper bound for the clock. The model checker is then used to either check that the upper bound is correct or generate an error trace that will help in determining a next guess (trial) for the upper bound.

- The approach is highly inefficient to be used for large models or models that contain largely repetitive finite cycles. However, the approach might be feasible for small models or acyclic models.

- The approach can not recognize the cases where WCET is infinity. Note that the WCET of an automaton can be infinity if it contains an infinite cycle.
(loop) or an unconstrained location (i.e. a location that is not guarded with an invariant). However, we will discuss in the following chapter how one can detect on-the-fly such cases using the zone-based approach.

4.2.3 A modified Version of The Zone Approach

A more efficient approach to compute the WCET of systems is to look at the value of the constraint \((D_{i,0}, \prec_{i,0})\) in the DBM \(D\) obtained in the final states, where \(\prec \in \{<, \leq\}\). Before discussing how one can solve the WCET problem in a single run of the zone construction it is necessary first to summarise how the zone graph of a given automaton can be constructed using the new paradigm where zones are not abstracted during the analysis and that abstraction is only used during inclusion checking. Let \(D\) be a DBM in canonical form. We want to compute the successor of \(D\) w.r.t to a transition \(e = (l, l', a, \lambda, \phi)\). can be obtained using a number of elementary DBM operations which can be described as follows.

1. Let an arbitrary amount of time elapse on all clocks. In a DBM this means all elements \(D_{i,0}\) are set to \(\infty\).
2. Take the intersection with the invariant of location \(l\) to find the set of possible clock assignments that still satisfy the invariant.
3. Take the intersection with the guard \(\phi\) to find the clock assignments that are accepted by the transition.
4. Canonicalize the resulting DBM and check the consistency of the matrix.
5. Set all the clocks in \(\lambda\) that are reset by the transition to 0.
6. Take the intersection with the location invariant of the target location \(l'\).
7. Canonicalize the resulting zone at the target location \(l'\) and check the consistency of the matrix.

Combining all of the above steps into one formula, we obtain

\[
\text{succ}(D, e) = \text{Canon(Canon}((D^\phi) \land I(l)) \land \phi)[\lambda := 0]) \land I(l')
\]
where Canon represents a canonicalization function that takes as input a DBM and returns a canonicalized matrix in the sense that each atomic constraint in the matrix is in the tightest form, $I(l)$ is the invariant at location $l$, and \( \uparrow \) denotes the elapse of time operation. As one can see the above zone approach will yield at the end a real (non-extrapolated) zone graph since extrapolation is not applied during the construction of the zone graph.

The algorithms depicted in Figures 4 and 5 represent respectively the zone-based algorithms for calculating the BCET and WCET of real-time distributed systems. The algorithms take as input an automaton $A$ for the system to be analysed. The algorithm consists of three basic steps, computing the state space of the automaton $A$, searching for the set of final states in $A$, and then performing some operations on that set in order to determine the minimum and the maximum value that the additional clock $\delta$ can reach at that set of states. Each node in the computed tree is of the form $(l_i, Z_i)$ where $l_i$ is a location in the automaton and $Z_i$ is the corresponding non-extrapolated zone. It uses two data structures WAIT and PASSED to store symbolic states waiting to be examined, and the states that are already examined, respectively. The WAIT set is instantiated with the initial symbolic state $(l_0, Z_0)$. The global variable $\text{BCET}$ holds the currently best known shortest execution time of reaching the final location; initially it is $\infty$. Similarly, the global variable $\text{WCET}$ holds the currently best known longest execution time of reaching the final location; initially it is 0. The global clock $\delta$ keeps track of the execution time of the system.

In each iteration of the while loop, the algorithm selects a symbolic state from WAIT, checking if the state is a final state. If the state $s = (l_i, Z_i)$ is a final state such that $l_i \in L_f$ we update the best known $\text{BCET}$ to the lower bound of $\delta$ at $Z_i$ if it is smaller than the current value of $\text{BCET}$. On the other hand, if the state is not a final state and the lower bound of the clock $\delta$ in the successors of that state is less than the intermediate $\text{BCET}$, we add these successors to WAIT and continue to the next iteration. Note that the algorithm computes the state space of $A$ step by step using the operation $\text{post}_\alpha(l_i, Z_i)$ which computes the successors of the given symbolic state.
The operation \texttt{lowerBound}(Z, \delta) returns the lower bound of the clock \delta in the zone Z which is equivalent to the value of \((-D_{0,i})\) in the corresponding DBM. The operation \texttt{upperBound}(Z, \delta) returns the upper bound of the clock \delta in the zone Z which is equivalent to the value of \((D_{i,0})\) in the corresponding DBM.

Note that before adding a new state \((l, Z)\) to WAIT we check if \((Z \setminus \text{UC}) \subseteq \text{closure}_a(Z' \setminus \text{UC})\) for any state \((l, Z') \in \text{WAIT}\) since when \((Z \setminus \text{UC}) \subseteq \text{closure}_a(Z' \setminus \text{UC})\) then all states reachable from \((l, Z)\) are also reachable from \((l, Z')\), and thus we only need to explore \((l, Z')\), where the set \text{UC} represents the set of constraints in the zones involving the extra clock \delta. Note that it is necessary to check inclusion between zones with respect only to the automaton clocks.

It is worth mentioning here that we treat BCET and WCET differently when computing them. Since the computation of BCET can be improved during the analysis. That is, during the state space exploration, if a non-final state has a lower bound greater than the intermediate BCET, this state does not need to be explored any further and hence we do not add its successors to the WAIT list. On the other hand, if the WCET algorithm encounters a reachable state with unconstrained location then the search can stop immediately since the WCET will be infinity.

\begin{algorithm}
\textbf{Input}: \((M)\)
\textbf{Output}: BCET := \infty
\text{clock} \delta
\text{PASSED} := \emptyset, \text{WAIT} := \{(l_0, Z_0)\}
\text{while} \text{WAIT} \neq \emptyset \\
\text{select} \ (l, Z) \text{ from } \text{WAIT} \\
//Check if \((l, Z)\) is a final node on some branch of the tree \\
\text{if} \quad \text{for all} \ a \in \Sigma \ \text{post}_a((l, Z)) = \emptyset \ \text{then} \\
\text{if lowerBound}(Z, \delta) < \text{BCET} \text{ then } \text{BCET} := \text{lowerBound}(Z, \delta)
\text{add} \ (l, Z) \text{ to } \text{PASSED}
\text{for all} \ (l', Z') \text{ such that} \ (l, Z) \sim (l', Z') \text{ do}
// check if lower bound of \delta in the new zone is less than the best known BCET.
\text{if lowerBound}(Z', \delta) < \text{BCET} \\
\text{if} \ (Z' \setminus \text{UC}) \not\subseteq \text{closure}_a(Z'' \setminus \text{UC}) \text{ for all } (l', Z'') \in \text{PASSED} \\
\text{then add} \ (l', Z') \text{ to } \text{WAIT}
\text{return} \ \text{BCET}
\end{algorithm}

\textbf{Algorithm 4}: Zone-base algorithm for Computing Best-Case Execution Time
4.2. APPROACHES FOR VERIFYING WCET OF TA

Input: \( (\mathcal{A}) \)
Output: \( \text{WCET} := 0 \)

\[ \text{clock } \delta \]
\[ \text{PASSED} := \emptyset, \text{WAIT} := \{ (l_0, Z_0) \} \]
\[ \text{while } \text{WAIT} \neq \emptyset \]
\[ \text{select } (l, Z) \text{ from WAIT} \]

//Check if \( (l, Z) \) is a final node on some branch of the tree
\[ \text{if } \text{for all } a \in \Sigma \text{ post}_a((l, Z)) = \emptyset \text{ then} \]
\[ \text{if } \text{upperBound}(Z, \delta) > \text{WCET} \text{ then } \text{WCET} := \text{upperBound}(Z, \delta) \]
\[ \text{add } (l, Z) \text{ to PASSED} \]
\[ \text{for all } (l', Z') \text{ such that } (l, Z) \sim (l', Z') \text{ do} \]

// if the location of the new state is not guarded with an invariant
\[ \text{if } \text{upperBound}(Z', \delta) = \infty \text{ then } \{ \text{WCET} := \infty; \text{WAIT} := \emptyset; \text{break} \} \]

// Check inclusion between zones
\[ \text{if } (Z' \setminus \mathcal{U}) \not\subseteq \text{closure}_\alpha(Z'' \setminus \mathcal{U}) \text{ for all } (l', Z'') \in \text{PASSED} \]
\[ \text{then add } (l', Z') \text{ to WAIT} \]
\[ \text{return } \text{WCET} \]

**Algorithm 5:** Zone-base algorithm for Computing Worst-Case Execution Time

**Efficient inclusion testing.** The test of the form \( Z \subseteq \text{closure}_\alpha(Z') \) used in the proposed algorithm, where \( \text{closure}_\alpha(Z') \) is the region closure of a set of valuations \( Z' \), is the key difference with respect to the other standard algorithms for TA that make use the tests of the form \( \text{Extra}_\alpha(Z) \subseteq \text{Extra}_\alpha(Z') \). The idea is that instead of considering nodes \( (l, \text{Extra}_\alpha(Z)) \) with set of extrapolated valuations \( \text{Extra}_\alpha(Z) \), one considers a union of the parts (regions) of \( \mathbb{R}^X_{\geq 0} \) that intersect \( Z \). The closure by regions of a zone \( Z \) with respect to a set of regions \( R \) is defined as the smallest set of regions from \( R \) that have a non-empty intersection with \( Z \) (i.e. \( \text{closure}_R(Z) = \{ m \in R \mid Z \cap m \neq \emptyset \} \)) [Bou04]. It is important to note that \( \text{Extra}_\alpha(Z) \) is different than \( \text{closure}_\alpha(Z) \): \( \text{Extra}_\alpha(Z) \) is an extrapolation of a zone which can be computed using the standard extrapolation procedures like \( M \)-extrapolation (see Definition 2.10.9) and hence it is a convex whereas \( \text{closure}_\alpha(Z) \) is a region closure of a zone and hence it can be non-convex [Bou04]. As observed in [HKSW11] to decide whether a region \( R \) intersects a zone \( Z \) it is enough to verify that the projection on every pair of variables is nonempty (see proposition 1). The proofs given in [Bou04, HKSW11] have shown that the inclusion test \( Z \subseteq \text{closure}_\alpha(Z') \) is sound for some operator \( \alpha \) and is as efficient as the test \( \text{Extra}_\alpha(Z) \subseteq \text{Extra}_\alpha(Z') \), and hence the overall
complexity for inclusion checking is still $O(|X|^2)$, where $|X|$ is the number of clocks. The advantages of this are twofold: (1) it allows us to maintain real zones which is necessary for the correctness of our analysis, and (2) it guarantees the correct termination of the search without increasing the computational complexity. Note also that the algorithm for checking $\mathcal{Z} \subseteq \text{closure}_\alpha(\mathcal{Z}')$ neither need to represent nor to compute the closure which may not be a zone (see Theorem 1).

**Definition 4.2.1.** Suppose we have a bound function that assigns to each clock $x$ in $\mathcal{A}$ a bound $\alpha_x \in \mathbb{N}$. A region [AD94] with respect to $\alpha$ is the set of valuations specified as follows:

1. for each clock $x \in X$, one constraint from the set: \( \{ x = c \mid c = 0, ..., \alpha_x \} \cup \{ c - 1 < x < c \mid c = 1, ..., \alpha_x \} \cup \{ x > \alpha_x \} \)
2. for each pair of clocks $x, y$ having interval constraints: $c - 1 < x < c$ and $d - 1 < y < d$, it is specified if $\text{fract}(x)$ is less than, equal or greater than $\text{fract}(y)$.

**Proposition 4.2.2.** [HKSW11] Let $\mathcal{R}$ be a region and $\mathcal{Z}$ be a zone. The intersection $\mathcal{R} \cap \mathcal{Z}$ is empty iff there exist variables $x, y$ such that $\mathcal{Z}_{yx} + \mathcal{R}_{xy} \leq (\leq, 0)$.

where $\mathcal{R}_{xy}$ is the weight of the edge $x \xrightarrow{\mathcal{R}_{xy}} y$ in the canonical distance graph representing $\mathcal{R}$. Similarly for $\mathcal{Z}_{xy}$.

**Theorem 4.2.3.** [HKSW11] Let $\mathcal{Z}, \mathcal{Z}'$ be zones in canonical form. Then $\mathcal{Z} \not\subseteq \text{closure}_\alpha(\mathcal{Z}')$ iff there exists variables $x, y$, both different than $x_0$, such that one of the following conditions holds:

1. $\mathcal{Z}'_{0,x} < \mathcal{Z}_{0,x}$ and $\mathcal{Z}'_{0,x} \leq (\alpha_x, \leq)$, or
2. $\mathcal{Z}'_{x,0} < \mathcal{Z}_{x,0}$ and $\mathcal{Z}'_{x,0} \geq (-\alpha_x, \leq)$, or
3. $\mathcal{Z}_{x,0} \geq (-\alpha_x, \leq)$ and $\mathcal{Z}'_{x,y} < \mathcal{Z}_{x,y}$ and $\mathcal{Z}'_{x,y} \leq (\alpha_y, \leq) + [\mathcal{Z}_{x,0}]$.

Where $[\mathcal{Z}_{x,0}]$ represents the integral part of the entry $\mathcal{Z}_{x,0}$ in the zone $\mathcal{Z}$ and $\alpha_x$ is the corresponding extrapolation constant of the clock $x$. Recall that the entry $\mathcal{Z}_{x,0}$ has
the form \([Z_{x,0}], \prec_{x,0}\). However, to implement the inclusion test given in Theorem 4.2.3 two operations on bounds are needed: comparison and addition. We define that \((n, \prec_1) < (m, \prec_2)\) if \(n < m\) and \((n, <) < (n, \leq)\). Further we define addition as \(n + \infty = \infty\), \((n, \leq) + (m, \leq) = (n + m, \leq)\), and \((n, <) + (m, <) = (n + m, <)\).

**Theorem 4.2.4.** The zone-based Algorithms 4 and 5 compute correctly the minimum and maximum execution times of acyclic and finite cyclic TA.

Note that the algorithm computes the transitive closure of \(\sim\) step by step using the operation \(\text{post}_a(l, Z_i)\) (while extrapolation is disabled) until it reaches the final state of the explored path and then checks whether the lower-bound value of the clock \(\delta\) is smaller than the best known value of BCET, therefore the algorithm guarantees to return with a correct answer. However, the absence of infinite runs in the behaviour of the automaton under analysis (as we assume in this chapter) implies termination.

We now turn to discuss the complexity of the DBM-based algorithms. In Table 2 we summarise the necessary DBM operations used by the algorithms with their complexity. All required operations can be implemented on DBMs with satisfactory efficiency. Note that \(X\) in the table denotes the number of clocks in the analysed model. It is worth mentioning that the BCET/WCET problem requires the DBM at each state in the model to be canonical, otherwise we may not be able to compute correctly the values of BCET/WCET since some of the operations performed by the analysed automaton may break the canonicity of the matrix and therefore the entries \(D_{0,i}\) and \(D_{i,0}\) may not be tight enough to reflect the actual values of BCET and WCET.

Since the computation of the complexity of each given DBM operation is immediate (see [BY04] for more details about each operation), we will go only through one of these operations in order to explain how to compute the complexity. We take the inclusion test operation as an example. Note that in order to decide whether a zone \(Z\) is included in the zone \(Z'\) (i.e. \(Z \subseteq Z'\)) we need to check that the corresponding DBM of zone \(Z\) denoted as \(M\) is included in the corresponding DBM of zone \(Z'\).
denoted as $M'$. Mathematically, we need to check whether all the corresponding entries in the two matrices satisfy the condition $M_{i,j} \leq M'_{i,j}$, which is a necessary and sufficient condition to conclude that $Z \subseteq Z'$. Since the size of the DBM matrix is $n^2$ then the inclusion test operation will cost $O(n^2)$.

Given the time complexity of each DBM operation performed by the algorithms we end up with a polynomial time complexity of the form given in Theorem 4.2.5, where $|Z|$ is the number of generated zones of the automaton under analysis, and $|E|$ is the number of reachable transitions of the automaton.

<table>
<thead>
<tr>
<th>DBM-operation</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inclusion test (i.e. $Z \subseteq \text{closure}_\alpha(Z')$)</td>
<td>$O(</td>
</tr>
<tr>
<td>Consistency test</td>
<td>$O(</td>
</tr>
<tr>
<td>Constraint satisfaction</td>
<td>$O(</td>
</tr>
<tr>
<td>Delay</td>
<td>$O(</td>
</tr>
<tr>
<td>Resetting Clocks</td>
<td>$O(</td>
</tr>
<tr>
<td>Constraint intersection</td>
<td>$O(</td>
</tr>
<tr>
<td>canonicalization</td>
<td>$O(</td>
</tr>
<tr>
<td>Clock-lower/upper bound test</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

Table 1: Complexity of BCET/WCET algorithms in terms of DBM operations

**Theorem 4.2.5.** The BCET zone-based algorithm 4 and the WCET zone-based algorithm 5 compute correctly the minimum and maximum execution times of acyclic and finite cyclic TA in a polynomial time complexity of $O(|E|.|Z|^2.|X|^2)$.

### 4.3 Implementation

In this section we briefly summarise our prototype implementation of the model checking algorithms given in Section 4.2.3. The prototype implementation has been developed using the opaal tool [DHJ+11] (http://opaal-modelchecker.com) which has been designed to rapidly prototype new model checking algorithms. The opaal tool is implemented in Python and is a standalone model checking engine. Models are specified using the UPPAAL XML format. The main step in the implementation of the algorithms is the representation of sets of symbolic state and the operations
required on them. We use the open source UPPAAL DBM library for the internal symbolic representation of time zones in the algorithms.

4.4 Case Studies

We consider here a simple realistic automatic manufacturing plant taken from Daws and Yovine [DY95]. We first give an informal description of the case study then we give the timed automata model of the entire system in UPPAAL, and finally report on the results obtained from running the BCET/WCET algorithms on the case study when considering it under different configurations.

The manufacturing plant that we consider consists of a conveyor belt that moves from left to right, a processing or service station, and two robots that move boxes between the station and the belt. The first robot called D-Robot takes a box from the station and put it on the left end of the belt. The second robot called G-Robot picks the box from the right end of the belt and transfers it to the station to be processed. We are then interested in verifying the minimum and maximum amount of time a box can take to be processed when considering the manufacturing plant under different configurations.

The timed automaton for the D-Robot is given in Figure 8. Initially, the robot waits until a box is ready indicated by the synchronisation label $s$-ready. Next, it picks the box up, turns right and puts the box on the moving belt. It then turns left and returns to its initial position.

The timed automaton for the G-Robot is given in Figure 9. This robot waits at the inspection point at the right end of the belt until a box passes this point. The G-Robot must pick up the box before it falls off the end of the belt. Next, it turns right, waits for the station to finish processing the previous box and then puts the box at the station. Finally, it turns left back to the inspection point. Note that picking the box up by the robot, turning left or right takes time which depends mainly on the speed of the robot.
The timed automaton for the processing station is given in Figure 11. The station is initially empty. Once a box arrives at the station it takes around 8-10 time units to be processed. The box is then ready to be picked up by the D-Robot.

The timed automaton for the box is given in Figure 10. The box initially moves from the left end of the belt to the inspection point. It takes between 133-134 time units for the box to reach the inspection point from the left end of the belt. Then it will be picked up by the G-Robot.

Using the zone-based algorithms we could analyse the manufacturing system up to
4.4. CASE STUDIES

Figure 10: The Box template

Figure 11: The processing station template

<table>
<thead>
<tr>
<th>No. of processes</th>
<th>Run-time/Memory (BCET)</th>
<th>Run-time/Memory (WCET)</th>
<th>BCET</th>
<th>WCET</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0.01s/33,042KB</td>
<td>0.015s/38,072KB</td>
<td>149</td>
<td>153</td>
</tr>
<tr>
<td>6</td>
<td>0.435s/39,470KB</td>
<td>0.922s/45,860KB</td>
<td>149</td>
<td>174</td>
</tr>
<tr>
<td>9</td>
<td>55s/469,463KB</td>
<td>70s/524,096KB</td>
<td>149</td>
<td>204</td>
</tr>
<tr>
<td>14</td>
<td>205s/469,463KB</td>
<td>280s/524,096KB</td>
<td>149</td>
<td>255</td>
</tr>
</tbody>
</table>

Table 2: The BCET/WCET of the manufacturing system for different number of boxes where the two robots move at the same speed.

9 processes (automata) (6 boxes, G-Robot, D-Robot, and a service station). All experiments are conducted on a PC with 32-bit Redhat Linux 7.3 with Intel (R) core CPU at 2.66 GHz and with 4 GB RAM. In Table 2 we verify the performance of the system under the following time constraints: the time required for the box to reach the inspection point is within $[133, 134]$, and the time required to process a box at the station is within $[8, 10]$. In this configuration, we assume that the D-Robot is faster than the G-Robot in the sense that the D-Robot can turn left and right and pick up and put boxes faster than the G-Robot as shown in Figures 8 and
As we expect when we increase the number of boxes in the model the value of WCET varies which implies that the number of boxes in the model impacts directly the WCET. However, it is not the case for the BCET since the value of BCET does not depend on the number of boxes in the model. The reason why the BCET does not change as we increase the number of boxes is because in the best case scenario the box will be processed immediately once it arrives the service station so that there will not be any queuing delay.

In Table 3 we verify the system under the same settings used in Table 2 except that we increase the speed of the two robots and assume that both robots move at the same speed. In this configuration the time the robot takes to pick the box up or to put it down is within \([1, 2]\) time units, and the time it takes to turn left or right is within \([2, 6]\) time units. As shown in Table 3 the performance of the system under this configuration has been improved where the values of BCET and WCET decreased under this configuration.

We compare the performance of the zone-based approach with the classical binary search approach used in [Met04, DOT+10] in which the user needs to repeatedly verify some parametrised temporal formulas until they hold. For example, one can use the temporal formula \(\text{AG}(\delta \leq guess)\) to verify the least upper bound for termination of the models, where \(\text{AG}\) are temporal operators that mean for each reachable state in the model the value of the extra clock \(\delta\) can not exceed the bound \(guess\). Using the binary search approach we could analyse the manufacturing system up to 9 processes (6 boxes, G-Robot, D-Robot, and a service station). As shown in Table 1 the binary search approach is quite competitive to our approach when considering small instances of the system. However, when considering instances with large number of processes the proposed approach outperforms the binary search approach by several order of magnitude, enabling models involving large number of processes to be model checked efficiently.
### 4.5 Conclusion

In this chapter we proposed algorithms for determining the best and worst case “execution time” in timed automata by modifying the underlying model-checking algorithm, rather than analyzing those times by augmenting the models with clock variables and querying those. The proposed algorithms can successfully handle acyclic TA and TA containing cycles given that the automaton under analysis has no run of infinite length. The algorithms avoid the extra computations and the extra canonicalization steps that may be needed if extrapolation is applied at each step of the successor computation.

<table>
<thead>
<tr>
<th>No. of processes</th>
<th>Run-time/Memory (BCET)</th>
<th>Run-time/Memory (WCET)</th>
<th>BCET</th>
<th>WCET</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0.031s/58.05KB</td>
<td>0.055s/65.82KB</td>
<td>158</td>
<td>171</td>
</tr>
<tr>
<td>6</td>
<td>0.635s/256.47KB</td>
<td>2.922s/465.96KB</td>
<td>158</td>
<td>185</td>
</tr>
<tr>
<td>9</td>
<td>285s/896.463KB</td>
<td>425s/1026.096KB</td>
<td>158</td>
<td>215</td>
</tr>
</tbody>
</table>

Table 3: The BCET/WCET of the manufacturing system using binary search approach for different number of boxes where the two robots move at different speeds.
Chapter 5

WCET of Arbitrary Timed Automata

In the previous chapter, we presented an algorithm for computing WCET of TA given that the automaton under analysis has no run of infinite length. In this chapter we present a new algorithm for computing WCET that can work on any arbitrary TA. The algorithm can handle more cases than previously existing algorithms for WCET computation, as it can handle cycles in TA and decide whether they lead to an infinite WCET. To our knowledge, this is the first model checking algorithm that addresses comprehensively the WCET problem of TA. In [BFH+01] Behrmann et al provide an algorithm for computing the minimum cost/time of reaching a goal state in priced timed automata (PTA). The algorithm has been implemented in the well-known model checking tool UPPAAL to compute the minimum time for termination of an automaton. However, we show that in certain circumstances, when infinite cycles exist, the algorithm implemented in UPPAAL may not terminate due to a canonicalization mistake, and we provide a set of examples which UPPAAL fails to verify.
5.1 Introduction

In this chapter, we present a new algorithm for computing the “worst case execution time” (WCET) in timed automata. Given a timed automaton $\mathcal{A}$ with a start location $l_s$ and a final location $l_f$, this problem asks to compute an upper bound on the time needed to reach the final location $l_f$ from the start location $l_s$. The problem is easy to solve in the case of acyclic TA (see the previous chapter), but cycles might introduce unbounded WCET, that need to be detected on-the-fly during the analysis.

The solution proposed here consists in modifying the classical zone-based forward approach for reachability in TAs. The solution proceeds by adding an extra clock $\delta$ that is never reset and never occurs in the guards and invariants of the TA. Then computing the zone graph (with special modified extrapolation and canonicalization procedures) of the TA allows one to obtain the WCET, by inspecting the elements of the zone graph of the form $(l_f, Z_f)$, where $l_f$ is a final location in the automaton, and by computing an upper bound on the clock valuations that satisfy all those zones $Z_f$.

Unfortunately, without the extrapolation operator, the analysis might not terminate in general because the zone graph can be infinite. Also, using extrapolation might introduce coarse over-approximations on the valuations of $\delta$, in case it exceeds the maximal extrapolation constant, for instance.

To solve the problem we propose a combination of (i) new extrapolation and canonicalization operators that keep the valuations of the extra clock $\delta$ precise, and (ii) an on-the-fly analysis of the possible cycles in the TA. The proposed algorithm guarantees termination and computes precisely the WCET and it can detect the cases where the WCET can be infinity. Thus, the proposed solution can be a significant break-through in computing WCET. Finally, we report on a prototype implementation using the model checker oppal [DHJ+11] and evaluate the algorithm on several toy examples. We compare also our prototype against the model checker UPPAAL, which shows that the implementation in UPPAAL may not terminate on models.
with infinite cycles containing unbounded clocks (i.e. WCET is unbounded but it is not obvious in the model).

In [BFH+01] Behrmann et al propose an algorithm that aims to provide a solution to the minimum cost/time reachability problem in TA in the presence of extrapolation. However, the authors of [BFH+01] have not given a formal description of how they extrapolate and canonicalize the zones in the resulting zone graph. Note that the key difficulty in developing a solution to the minimum/maximum termination time problem using the zone approach is to define an abstraction of zones that guarantees termination of the algorithm, while keeping information precise for $\delta$. This involves adapting two classical operations on zones: extrapolation and canonicalization. The later has been forgotten in [BFH+01] leading to non termination. So the key contribution of this chapter is a new canonicalization operation that ensures termination while preserving the value of $\delta$. The modified version of Behrmann et al algorithm with the new canonicalization operation can successfully compute BCET for any arbitrary automaton.

### 5.2 Behrmann et al Minimum Cost Reachability Algorithm

The minimum cost reachability algorithm described in [BFH+01] uses a variant of timed automata called “uniformly priced timed automata” and the DBM data structure to compute the minimum cost of reaching a goal state in the model. A priced timed automata can associate costs with locations, where the costs are multiplied by the amount of time spent in a location. An automaton may be designed so that the total cost corresponds to the execution time, and thus this approach may be used to calculate the best case execution time. However, the authors in [BFH+01] have not given a (detailed) formal description of how they extrapolate and canonicalize priced zones of a constructed priced zone graph when
they add an extra clock (which they call $\delta$), except the following remark given at page 9.

Termination is ensured if all clocks except for $\delta$ are normalised with respect to a maximum constant $M$. It is important that normalisation never touches $\delta$. With this modification, the algorithm in Fig. 1 will essentially encounter the same states as the traditional forward state-space exploration algorithm for timed automata, except for the addition of $\delta$.

The remark above does not constitute a concrete definition of extrapolation (normalisation) and does not mention anything about canonicalization, and hence leaves a number of questions concerning implementation open. In fact, the description of the modified extrapolation given in [BFH+01] is very vague, and should be made more formal. For example, which set of constraints in reachable zones should not be extrapolated? How should reachable zones be canonicalized so that correctness and termination of the analysis are guaranteed? Do we need to apply different canonicalization procedures on the two sets of constraints in zones: the set of extrapolated constraints and the set of non-extrapolated constraints? More precisely, how the partially extrapolated zones (i.e. zones that contain extrapolated and non-extrapolated constraints) should be canonicalized during the analysis? As we know, key operations of the zone abstraction are canonicalization and extrapolation. Canonicalization assigns the tightest possible bound for each pair of clocks whereas extrapolation enlarges bounds that exceed a certain value after which the value of a clock has no effect on the structure of the zone graph. Canonicalization is needed for the comparison of zones and for efficient implementation of several constraint operations, extrapolation guarantees the finiteness of the zone graph. However, the extrapolation and canonicalization procedures and their roles in forward reachability algorithm with respect to certain problems such as the minimum termination time problem and the maximum termination time problem, require extra care and non-trivial arguments for proving both correctness and termination in
particular when partially extrapolating the DBMs.

Suppose we have an automaton $\mathcal{A}$ and we would like to compute its minimum and maximum termination time. Suppose further that $\mathcal{A}$ has two clocks $y$ and $z$. Let us assume that we add an extra clock $\delta$ that is used to compute the minimum and maximum time for termination of $\mathcal{A}$, which is not reset and not extrapolated during the analysis. We can then describe the general form of the partial extrapolated matrix (zone) that can be obtained at each reachable location of $\mathcal{A}$ as follows

$$M_{PE} = \begin{pmatrix}
  x_0 & y & z & \delta \\
  x_0 & . & . & . & * \\
  y & . & . & . & * \\
  z & . & . & . & * \\
  \delta & * & * & * & * 
\end{pmatrix}$$

where the asterisk sign is used to denote a constraint involving $\delta$ which is a constraint that is not touched during extrapolation and the dot sign is used to denote a constraint involving only the automaton clocks which may be extrapolated and it is in $M$-form. We say that a constraint $(D_{i,j}, \prec_{i,j})$ is in $M$-form if $-M \leq D_{i,j} \leq M$.

As one can see all the constraints involving the extra clock $\delta$ (the asterisk entries) are not touched during extrapolation in order to keep the extra clock precise during the construction of the zone graph. On the other hand, the constraints involving the automaton clocks (the dot entries) may be extrapolated during the analysis in order to guarantee termination in particular when infinite cycles exist. Note that during the construction of the zone graph $Z(\mathcal{A})$ the only operation in which the set of non-extrapolated constraints may influence the set of extrapolated constraints and vice versa is the canonicalization operation where the constraints in the matrix are tightened. Note that canonicalization may be repeated three times at each step of the successor computation: one time before resetting clocks (if any), one time after intersecting the matrix with the invariant at the target location, and one time after extrapolation. Hence, canonicalization needs to be performed carefully so that
the constraints involving the extra clock $\delta$ remain precise while guaranteeing termination of the analysis. The key question is then how the partially extrapolated matrix $M_{PE}$ should be canonicalized at each step of the successor computations? In fact, this question has not been studied carefully in the prior literature.

Before answering the above question we show first that in certain circumstances, when infinite cycles exist, the algorithm in [BFH+01] and its implementation in UPPAAL may not terminate. To support our claim we give four examples of TA where the algorithm as described in [BFH+01] does not guarantee termination. However, to support further our theoretical claim we give the results of verifying the examples using the latest version of the tool UPPAAL (4.1.19), Windows version, which show that the tool fails to terminate. Note that in UPPAAL, one can use a global clock GBL and check two properties on system A: $\inf\{ A.\text{end} \}: \text{GBL}$ and $\sup\{ A.\text{end} \}: \text{GBL}$. The sup/inf operators are documented in the Help menu of UPPAAL. The example in Figure 12 shows an automaton with an infinite cycle (loop) where BCET is 20 and WCET is infinity. UPPAAL fails to give an answer for BCET and WCET of that automaton. UPPAAL also fails to handle the simple infinite automaton given in Figure 13 where BCET is 1 and WCET is infinity. For this particular automaton we verify the BCET using the command $\inf\{ A.\text{end} \}: \text{GBL}$ and the WCET using the command $\sup\{ \text{GBL} \}$. However, UPPAAL fails to terminate and hence no answer has been obtained which shows clearly that the algorithm fails in this case. For the automaton in Figure 14 which has a BCET of 11 and an infinite WCET UPPAAL fails also to handle this automaton and no answer has been obtained for BCET and WCET. On the other hand, the example in Figure 15 shows an automaton that contains three finite cycles that have the location start as a common location. It is interesting to note that there are some dependencies between the behaviour of the three cycles. However, as one can see, the three cycles collectively will be executed infinitely often which lead to an infinite WCET. UPPAAL fails to handle such classes of cycles where the operator sup fails to terminate and hence no answer has been obtained.
CHAPTER 5. WCET OF ARBITRARY TIMED AUTOMATA

Figure 12: An automaton with BCET = 20 and WCET = ∞

Figure 13: A simple infinite automaton with BCET = 1 and WCET = ∞

Figure 14: An automaton with a BCET = 11 and WCET = ∞

Figure 15: An automaton where BCET = 0 and WCET = ∞
Let us see what happens when we compute the zones of the automaton in Figure 12 using the partial $M$-extrapolation algorithm proposed by Behrmann et al in [BFH+01] where the extra clock is not touched during extrapolation. Firstly, note that the extrapolation constant $M$ is 20. The automaton has two clocks $y$ and $z$. Let us call the extra clock $\delta$. We give the sequence of zones obtained below. Note that for convenience only the full canonical zone is written. First at location \texttt{start} we have the zone $(\delta = 0 \land y = 0 \land z = 0)$. During the forward traversal of the TA the location \texttt{loop} is reached with the clock zone $(\delta = 0 \land y = 0 \land \delta = 0)$. Clearly, extrapolation is not necessary here since none of the constraints exceeds the extrapolation constant $M$. After taking the transition \texttt{loop} $\rightarrow$ \texttt{loop} a state $(\texttt{loop}, Z_2)$ with $Z_2 = (\delta \leq 10 \land y = 0 \land \delta = z)$ will be added. Again extrapolation is not necessary here. A second loop will add $(\texttt{loop}, Z_3)$ with $Z_3 = (\delta \leq 20 \land y = 0 \land \delta = z)$. Extrapolation is not needed here as well. A third loop will add $(\texttt{loop}, Z_4)$ with $Z_4 = (\delta \leq 30 \land y = 0 \land \delta = z)$. Before proceeding further, note that the zone $Z_4$ needs to be extrapolated since there are some constraints that exceed the value of the extrapolation constant $M$. Recall that in the partial extrapolation approach we do not change all the asterisk entries in the matrix (i.e. the entries involving $\delta$) in order to keep them precise and we just extrapolate the dot entries (i.e. the entries involving the automaton clocks). One can check this would give the zone $Z'_4 = (\delta \leq 30 \land y = 0 \land \delta - z = 0 \land z = \infty)$. Note that the zone $Z_4$ is not on a canonical form. We use the Floyd-Warshall algorithm to canonicalize the zone $Z'_4$. We obtain the zone $Z''_4 = (\delta \leq 30 \land y = 0 \land z \leq 30)$ which is the one we obtained before extrapolation. If we continue computing the zones in this way we will find ourselves dealing with real zones rather than abstracted ones and hence the loop can be taken infinitely often enlarging the state space such that a fixed point will never be reached. This happens because the constraints involving $\delta$ have not been changed during extrapolation and then during the canonicalization step the value of these constraints influenced the value of the constraints involving the automaton clocks. This explains why verification of the above four examples in UPPAAL does not terminate and that no answer can be obtained in such cases!
5.3 Computing WCET of Cyclic Real-time Systems

Similar to Behrmann et al solution our proposed solution proceeds by adding an extra clock (let us call it \( \delta \)) to the automaton under analysis that acts as an observer. Then one computes the zone graph of the automaton (involving \( \delta \)), by means of a standard forward analysis using DBMs. To get the WCET, the algorithm needs to look at the value of the constraint \((D_{i,0}, <_{i,0})\) in every reachable state including the initial state since the delay of a run can be infinity if there is an unconstrained location along that run or if there is an infinite cycle in which time can elapse (see Lemma 3.2.4).

5.3.1 Solving The Problem Using Partial Extrapolation

We discuss now an extrapolation procedure that can be used to keep the extra clock precise to the end of the analysis. We use the term “partial extrapolation” for such a procedure. Let us denote the sub-DBM that consists in the asterisk entries as \( \mathcal{M}_{PE}^* \), and the sub-DBM that consists in the dot entries as \( \mathcal{M}_{PE}^\dot{\cdot} \) which may be extrapolated during the extrapolation steps. So to solve the problem we choose to split the DBM \( \mathcal{M}_{PE} \) into two sub-DBMs \( \mathcal{M}_{PE}^* \) and \( \mathcal{M}_{PE}^\dot{\cdot} \). Note that such splitting is possible since DBMs are sets of constraints. We give now the conditions that are necessary to ensure correctness and termination of the analysis using the partial extrapolation approach.

1. (Condition C1: special extrapolation procedure of reachable zones). During the extrapolation steps, extrapolate only the dot entries in the matrix and leave all the asterisk entries unextrapolated. Note that this is necessary in order to keep the constraints involving the extra clock precise to the end of the analysis. For greater convenience we will use the notation \( \mathcal{M}_{\text{Extra}}(\mathcal{D}) \) to denote the modified extrapolation operation in which all the asterisk entries
are not touched during extrapolation. Let us assume that the clock \( \delta \) takes
index \( i \) in DBMs. We can then compute the modified extrapolation function
\( \text{MExtra}_M(D') \) of the matrix \( D' = (d'_{j,k}, \prec'_{j,k})_{j,k=0,...,n} \) as follows.

\[
(d'_{j,k}, \prec'_{j,k}) = \begin{cases} 
(\infty, <) & \text{if } d_{j,k} > M \land j, k \neq i, \\
(-M, <) & \text{if } d_{i,j} < -M \land j, k \neq i, \\
(d_{j,k}, \prec_{j,k}) & \text{otherwise.}
\end{cases}
\]

In fact, there is some lack of precision in the description of the extrapolation
procedure given in [BFH+01] since not just the clock \( \delta \) (i.e. the lower and
upper bound of \( \delta \)) must not be touched during extrapolation but also all the
diagonal constraints involving \( \delta \). Note that during the time elapse operation
the upper bound of all the clocks are set to \( \infty \) and hence the upper bound
of the clock \( \delta \) may become imprecise. However, during canonicalization and
with the help of the diagonal constraints involving \( \delta \), which remain precise,
the exact upper bound of \( \delta \) can be reconstructed since the relationship of \( \delta \) to
all other automaton clocks is preserved. It is easy to see that if the diagonal
constraints involving \( \delta \) get extrapolated during the construction of the zone
graph then the lower and upper bound of \( \delta \) will lose their precise values during
canonicalization steps.

2. **(Condition C2: special canonicalization procedure for dot entries).**

The dot entries in the matrix should be canonicalized independently or sep-
arrately from the asterisk entries. That is, during the canonicalization steps,
the asterisk entries should not participate in the process of canonicalizing the
dot entries. Otherwise, termination may not be guaranteed. To see why
condition C2 is necessary, consider the case where extrapolating a constraint
\((d_{j,k}, \prec_{j,k}) \in M_{PE} \) yields \((\infty, <)\). Now if the asterisk entries participate in the
process of canonicalizing the constraint \((\infty, <)\) it is possible to end up with a
constraint \((d'_{j,k}, \prec'_{j,k})\) which may not be in \( M \)-form (i.e. extrapolated form) in
the sense that \( d'_{j,k} > M \) or \( d'_{j,k} < -M \). Note that extrapolation only increases
bounds, never lowers them. Note also that canonicalization computes minimum and hence it is possible for the asterisk entries to influence the dot entries during canonicalization steps which should be avoided in order not to lose termination. This is what happened when analysing the example in Figure 12 while applying the partial $M$-extrapolation approach used in [BFH+01]. Recall that extrapolation is used to ensure termination and that termination of the analysis of an automaton depends on the constraints involving the automaton clocks.

3. **(Condition C3: canonicalizing asterisk entries in the matrix).** For the asterisk entries in the matrix it is necessary to canonicalize them using dot and asterisk entries. That is, to canonicalize the asterisk entries using normal or standard canonicalization procedure. Since the extra clock does not participate in the computations of the automaton and hence it is necessary to involve the constraints involving the automaton clocks in the canonicalization process. This ensures that the extra clock will advance at the same rate as the automaton clocks. Note that although the constraints involving the extra clock will be canonicalized using the entire set of constraints in the zones including the extrapolated constraints, the extra clock does not lose its precision in the end as one might expect. The reason is that during the construction of a zone graph the zones are canonicalized before they get extrapolated and that during canonicalization, a minimum is calculated. When extrapolation is applied it only increases bounds in the zone (i.e. extrapolation enlarges zones so that $D \subseteq M_{\text{Extra}}(D)$) [Rok93] and that the extra clock is not touched during extrapolation. Moreover, the relationships of the extra clock to the other automaton clocks are preserved by partial extrapolation where all the constraints involving the extra clock will not be touched during extrapolation (see condition C1). The key idea is that increasing bounds during extrapolation does not affect a function that was calculated using a minimum. For greater
5.3. COMPUTING WCET OF CYCLIC REAL-TIME SYSTEMS

convenience we will use the notation $\text{MC}_{\text{Canon}}(D)$ to denote the modified canonicalization procedure and to distinguish it from the standard canonicalization operation. Note that the operation $\text{MC}_{\text{Canon}}(D)$ is a specialisation of Floyd’s algorithm in which the dot entries are tightened separately from the asterisk entries, while the asterisk entries are tightened using asterisk and dot entries. A formal description of $\text{MC}_{\text{Canon}}(D)$ is given in Algorithm 6.

4. (Condition C4: special checks for handling cycles in TA). To handle cycles (loops) properly we propose to use what we call fixed point abstraction of zones (see Definition 5.3.1) rather than inclusion abstraction when handling the generated zones inside cycles. That is, during the analysis of a cycle in a TA we check whether the search can reach identical states with respect to the automaton clocks and whether the extra clock (i.e. non-extrapolated clock) can advance during the analysis of the cycle. If such situation happens we set the extra clock $\delta$ to infinity and terminate since the WCET of the automaton will be infinity. Note that the partial extrapolation approach with conditions (C1-C3) together with inclusion abstraction (see Chapter 4) can compute correctly the maximum termination time of an automaton only in the case the reachability analysis terminates in a target state, thus ignoring (infinite) cycles. Note that the approach visits all the reachable states of an automaton while ignoring the value of the extra clock for termination. Thus the reachability may end because of the widening of clocks, but the value of the global extra clock may not be exact when cycles exist. Therefore, we need an extra condition to handle (infinite) cycles properly. Consider an automaton with a clock $x$, a global clock $\delta$, a single location with invariant $x \leq 3$ and a self loop where $x$ is reset. The symbolic reachability ends with a zone that says that $x \leq 3$ and also that the global clock $\delta \leq 3$, due to the invariant constraint. However, the real value of the global clock is infinity.

Definition 5.3.1. (Fixed Point Abstraction). Let $A = (\Sigma, L, L_0, L_F, X, I, E)$ be a timed automaton, let $E_{\pi} = (e_0, ..., e_{n-1})$ be the sequence of edges of a cycle
\( \pi \) in \( \mathcal{A} \). Suppose that the operation \( \text{succ}(Z, E_\pi) \) computes the successor zone of \( Z \) after executing the sequence of edges in \( E_\pi \) which is equivalent to executing the cycle \( \pi \) one full iteration. We say that \( Z \) is a fixed point of \( \pi \) and \( \pi \) is an infinite cycle if \( \text{succ}(Z, E_\pi) = Z \). That is, if the cycle starts and ends with the same zone then the cycle is an infinite cycle.

Note that extrapolation is typically applied during the construction of the zone graph in order to enforce the termination of the forward searching in the automaton. Note that because of extrapolation it is always guaranteed to reach a fixed point of infinite cycles after a finite number of iterations. One can use extrapolation techniques such as \( M \)-extrapolation [Pet99] or \( LU \)-extrapolation [BBLR06] to enforce, on the fly, the convergence of fixed point computations.

Conditions C2 and C3 in the above described procedure may not be straightforward conditions as the other ones so it may be worth providing some formal argument why these conditions are necessary for the correctness of the procedure. To explain formally why conditions C2 and C3 are necessary we need some preliminary observations. The first observation is that extrapolation only increases bounds, never decreases them. The second observation is that canonicalization only decreases bounds, never increases them. The third observation is that the constraints involving the extra clock \( \delta \) are not touched (enlarged) during extrapolation. From these three observations it is easy to see that if a constraint \( c \) in a canonical zone \( Z \) has not been touched during extrapolation then the weight of \( c \) in the canonicalized extrapolated matrix can not be smaller than its weight in the canonicalized non-extrapolated matrix since extrapolation increases bounds and never lowers them (i.e. \( Z \subseteq \mathsf{MExtra}(Z) \)). In fact, the weight of the constraint \( c \) in both matrices will be the same. Hence, the asterisk entries will not be changed neither during the extrapolation step nor during the canonicalization step that fixes the non-tightness introduced by extrapolation. However, the above observations lead to lemma 5.3.2 which is interesting since it discusses a result that has not been noticed in the prior literature and it helps to simplify the proof of the main theorem.
of the chapter (Theorem 5.3.4).

However, conditions C2 and C3 can be formalized as described in Algorithm 6 where the asterisk entries in the matrix are canonicalized using asterisk and dot entries while the dot entries are canonicalized using only dot entries. This is necessary in order to guarantee the termination of the analysis and to ensure that the clock $\delta$ will advance at the same rate as the automaton clock as discussed in conditions C2 and C3. Recall that we assume that the clock $\delta$ takes index $i$ in DBMs.

```
for $p := 0$ to $n$ do
  for $q := 0$ to $n$ do
    if $p = i \lor q = i$ then
      for $k := 0$ to $n$ do
        $D_{p,q} := \min(D_{p,q}, D_{p,k} + D_{k,q})$
      end
    else
      for $k := 0$ to $n$ do
        if $k \neq i$ then
          $D_{p,q} := \min(D_{p,q}, D_{p,k} + D_{k,q})$
        end
      end
    end
  end
end
```

Algorithm 6: Special canonicalization procedure at steps 4 and 7 of the zone approach

**Lemma 5.3.2.** Let $Z$ be a zone on canonical form. Let $(c_{i,j}, \prec_{i,j})$ be a constraint in $Z$. Suppose that the zone $Z$ has been partially extrapolated using the $M$-extrapolation procedure (see Definition 2.10.9) and that the constraint $(c_{i,j}, \prec_{i,j})$ has not been touched during extrapolation. Then the weight of $(c_{i,j}, \prec_{i,j})$ in the canonical matrix $Z$ is equal to its weight in the matrix $\text{MCanon}(\text{MExtra}_M(Z))$ and hence $(c_{i,j}, \prec_{i,j})$ needs not to be recanonicalized after extrapolation.

**Proof.** Let first us denote the matrix $\text{MCanon}(\text{MExtra}_m(Z))$ as $Z'$. Let us denote also the constraint $(c_{i,j}, \prec_{i,j})$ after extrapolating and canonicalizing it as $(c'_{i,j}, \prec'_{i,j})$. We need to show that $(c_{i,j}, \prec_{i,j}) = (c'_{i,j}, \prec'_{i,j})$. From the assumption that $Z$ is on canonical form and by definition of the tightening algorithm [Rok93] we know that the weight of the constraint $(c_{i,j}, \prec_{i,j})$ in $Z$ is the tightest weight that can
be derived from the set of constraints in $Z$. By the definition of extrapolation we
know that the bounds of the constraints that are extrapolated are in fact increased.
Note that when the bound of a constraint is above the extrapolation constant $M$
then the extrapolation function $\text{MExtra}_M(Z)$ sets it to $\infty$, which is an increase, and
when the bound is less than $-M$ it sets it to $-M$, which is still an increase, and
therefore the function $\text{MExtra}_M(Z)$ only increases bounds. Now since the constraint
$(c_{i,j}, \prec_{i,j})$ has not been increased during extrapolation and that $Z \subseteq Z'$ and the
function $\text{MCanon}(Z)$ computes minimum it is easy to see then that $c_{i,j} = c'_{i,j}$ and
hence no need to recanonicalize $(c_{i,j}, \prec_{i,j})$. \hfill \qed

The above lemma leads to the optimised canonicalization procedure described in
Algorithm 7, which states that after extrapolation only the constraints that have
been changed need to be recanonicalized. Note that the list $\text{Changed}$ maintains the
list of constraints that have been changed during extrapolation represented as pairs
of indices. Recall that all the dot entries in the matrix need to be canonicalized
separately from the asterisk entries (see Condition C2).

\begin{algorithm}[ht]
\begin{algorithmic}
\For{$k := 0$ to $n$}
\If{$k \neq i$}
\For{$(p, q) \in \text{Changed}$}
\State $D_{p,q} := \min(D_{p,q}, D_{p,k} + D_{k,q})$
\EndFor
\EndIf
\EndFor
\end{algorithmic}
\caption{Special canonicalization procedure after extrapolation}
\end{algorithm}

Note that the above procedure (i.e. the modified zone based approach with condi-
tions C1-C4) may yield zones that are partially extrapolated and partially canoni-
ialized due to conditions C1 and C2. Since the asterisk entries in the matrix will
not be touched during extrapolation and that during canonicalization the dot en-
tries will be canonicalized using only dot entries while the asterisk entries will be
canonicalized using dot and asterisk entries. Note that this is necessary for the
correctness and the termination of the WCET analysis since the observable clock
$\delta$ that does not interfere with the guards of the automaton must not be touched
5.3. **COMPUTING WCET OF CYCLIC REAL-TIME SYSTEMS**

during extrapolation and has to advance at the same rate as the automaton clocks. However, the procedure guarantees termination since there is a finite number of sub-DBM $M_{PE}$ due to conditions C1, C2, and C4. The procedure also keeps information precise for $\delta$ due to conditions C1, C3, and C4. This is what we prove in Theorem 5.3.4.

Note that lemma 5.3.2 explains to us why condition C3 in the above procedure is sound and why the canonicalization operation at step 8 of the zone approach (see Section 2.10) will not affect adversely the constraints involving the extra clock $\delta$ (the asterisk entries) and hence they remain precise. In fact, the asterisk entries will not be changed at step 8 of the zone approach. From the above observations it is easy to see also that condition C2 is necessary since the dot entries may be increased during extrapolation and then during the canonicalization operation the asterisk entries may influence the dot entries in a way they may lose their $M$-form. This can affect adversely termination of the analysis in particular when infinite cycles exist (see examples in Section 5.2).

Note that the only steps in which the dot entries may influence the asterisk entries are the canonicalization operations at steps 4 and 7 of the standard zone approach (see Section 2.10). However, these canonicalization steps are safe in the sense that they will not affect adversely the asterisk entries (i.e. the asterisk entries will not lose their precise value during these canonicalization operations). Firstly, note that the first canonicalization step (step 4 of the zone approach) is applied to fix the non-tightness introduced by intersecting the zone with a transition guard and before resetting any automaton clock (if any), and the second canonicalization (step 7 of the zone approach) is applied to fix the non-tightness introduced by intersecting the zone with the location invariant at the target location and before extrapolating the zone. However, given that the relationship of the extra clock to each automaton clock is preserved by partial extrapolation it is easy to see that the extra clock will remain precise during these operations. Note that the canonicalization operations at steps 4 and 7 are necessary operations in order to allow the extra clock to advance.
at the same rate as the automaton clocks. Hence, we say that the modified zone-based approach (i.e. the standard zone approach with the set of conditions C1-C4) guarantees that the reachability analysis will terminate and the value of the extra clock will be exact in the end. This is what we prove in Theorem 5.3.4. In fact, a zone-based algorithm can be developed using the proposed partial extrapolation procedure that can solve the WCET problem for a general class of diagonal-free TA.

**Remark 5.3.3.** Let $D$ be a partially extrapolated matrix so that some constraints in $D$ are in $M$-form and some are not. Let $(D_{j,k}, \prec_{j,k})$ be a constraint in $M$-form. Canonizing $(D_{j,k}, \prec_{j,k})$ using the standard operation $\text{Canon}(Z)$ in which the entire matrix is tightened using the standard Floyd-Warshall algorithm may yield constraints not in $M$-form.

Remark 5.3.3 summarises the mistake that [BFH+01] makes during the analysis. As an example of remark 5.3.3 see the analysis of the automaton in Figure 12 at Section 5.2. However, one may argue that time elapse operation can affect the upper bound of $\delta$ in a way it becomes imprecise. Note that time elapse does only affect the upper bound of $\delta$, but not it’s relationship with other clocks. After time elapse, if there is just one other clock whose upper bound is not infinity due to extrapolation, the relationship of $\delta$ to this clock is preserved and thus the exact upper bound of $\delta$ can be reconstructed during canonicalization. During canonicalization, a minimum is calculated, thus the smallest upper bound dominates all others. However, if all the upper bounds of the clocks are set to infinity by extrapolation so that none of the clocks remains tight after extrapolation, then there is a path in the automaton that has $\text{WCET} = \infty$, as all clocks used in guards or invariants are beyond the biggest constant they ever compared against, thus the automaton must be in a state without an upper bound of the location.

**Theorem 5.3.4.** The partial extrapolation algorithm that satisfies conditions (C1-C4) keeps the observable clock $\delta$ precise to the end (i.e. the clock $\delta$ preserves its actual value) and guarantees termination.
5.3. COMPUTING WCET OF CYCLIC REAL-TIME SYSTEMS

Proof. Theorem 5.3.4 can be proved by reasoning about how $\land$, $\triangleright$, reset, $\text{MExtra}$, and $\text{MCanon}$ operations together with the conditions (C1-C4) modify the zones of the resulting graph of an automaton. The Theorem can be proved by induction on the length of the transition sequences. Suppose we have an automaton $A = (\Sigma, L, L_0, L_F, X, I, E)$ with a set of symbolic runs $R$. Let $r$ be an arbitrary run of $R$ which can be either finite or infinite run. As induction hypothesis, assume that the entries $-D_{0,i}^k$ and $D_{0,0}^k$ maintain respectively the precise infimum and supremum accumulated delays of the automaton $A$ up to $k$-transitions (i.e. $\langle l_0, D_0 \rangle \sim_{\alpha^k} \langle l_k, D_k \rangle$, where $\alpha$ is either a delay or discrete action). Assume further that $D_{i,j}^k$ and $D_{j,i}^k$, where $j \neq i$ and $j = 0, \ldots, n$, maintain the precise upper and lower bound difference between the extra clock and each other automaton clock up to $k$-transitions. From the definition of the zone approach we can write the entries $D_{0,i}^k$ and $D_{i,0}^k$ as follows: $D_{0,i}^k = ((D_{0,i}^{k-1} \land D_{0,j}(I(l_{k-1}))) \uparrow \land D_{0,i}(I(l_{k-1})) \land D_{0,i}(\psi_{k-1}))$ and $D_{i,0}^k = ((D_{i,0}^{k-1} \land D_{i,0}(I(l_{k-1}))) \uparrow \land D_{i,0}(I(l_{k-1})) \land D_{i,0}(\psi_{k-1}))$. From the semantic definition of DBMs and the fact that time can only elapse at locations the two entries can be simplified as follows: $D_{0,i}^k = x_0 - (\delta + \Sigma_{j=0}^{k-1}(\inf(d_j)))$ and $D_{i,0}^k = (\delta + \Sigma_{j=0}^{k-1}(\sup(d_j))) - x_0$, where $d_j$ is the allowed delay interval at location $l_j$ such that for all $v \in d_j$ the invariant $I(l_j)$ holds. Since $x_0$ has always the value 0 and $\delta$ has initially the value 0, we can then simplify the entries as follows: $D_{0,i}^k = -\Sigma_{j=0}^{k-1}(\inf(d_j))$ and $D_{i,0}^k = \Sigma_{j=0}^{k-1}(\sup(d_j))$. Now assume $\langle l_k, D_k \rangle \sim_{\alpha} \langle l_{k+1}, D_{k+1} \rangle$. We need to prove that after executing transition $(k + 1)$ the extra clock $\delta$ remains precise. That is, the entries $D_{0,i}^{k+1} = D_{0,i}^k - \inf(d_{k+1})$ and $D_{i,0}^{k+1} = D_{i,0}^k + \sup(d_{k+1})$, where $d_{k+1}$ is the allowed delay interval at location $l_{k+1}$. As we know from the semantic definition of the zone approach the upper bound of the clocks may become (temporarily) imprecise at each step of the successor computation due to the application of the delay operation which sets the upper bound of all clocks to $\infty$ and hence the extra clock $\delta$ may become imprecise. However, during canonicalization and with the help of the diagonal constraints of the form $(D_{i,j}, \prec_{i,j})$ and $(D_{j,i}, \prec_{j,i})$, where $j \neq i$ and $j = 0, \ldots, n$, the exact upper bound of $\delta$ can be reconstructed. Hence, to show that the entries $(D_{0,0}, \prec_{0,0})$ and $(D_{0,i}, \prec_{0,i})$
remain precise in the end we need to show also that all the diagonal constraints $(D_{i,j}, \prec_{i,j})$ and $(D_{j,i}, \prec_{j,i})$ remain precise during the analysis. However, since there are two types of transitions in TA: delay $\alpha = \epsilon(d)$ and action $\alpha \in \Sigma$ we need to consider two cases.

- (Delay $\alpha = \epsilon(d)$). By the assumption $\langle l_k, D_k \rangle \sim^{\epsilon(d)} \langle l_k, D_k + \epsilon(d) \rangle$ we know that $D_k + \epsilon(d) \models I(l_k)$. From the definition of $\models$ and by delay we have $\langle l_k, D_k \rangle \sim^{\epsilon} \langle l_k, D_{i+1} \rangle$. Expansion by the definition of $\models$ and with the invariant at the location $l_k$ we get $D_{k+1} \in ((D_k \land I(l_k))^\uparrow \land I(l_k))$. By the definition of canonicalization and following condition C2 and C3 we get $D_{k+1} \in (\text{Mcanon}(D_k \land I(l_k))^\uparrow \land I(l_k))$. From the semantic definition of DBMs and after executing the above operations we get $D_{0,i}^{k+1} = D_{0,i}^k - \inf(d)$ and $D_{i,0}^{k+1} = D_{i,0}^k + \sup(d)$. From the semantic definition of the zone approach we know that the first canonicalization operation will fix the non-tightness introduced by the delay operation and hence the exact lower and upper bound of $\delta$ will be reconstructed during this operation. By induction hypothesis we know that $D_{0,i}^k$ and $D_{i,0}^k$ maintain the accumulated delays of $A$ up to $k$-transitions. Thus the value of the entries $D_{0,i}^{k+1}$ and $D_{i,0}^{k+1}$ in the matrix $(\text{Mcanon}(D_k \land I(l_k))^\uparrow \land I(l_k))$ are precise and hence the delay transition does not affect adversely the constraints involving $\delta$.

- (Action $\alpha \in \Sigma$). By the assumption $\langle l_k, D_k \rangle \sim^\alpha \langle l_{k+1}, \text{reset}[\lambda]D_k \rangle$ we know $l_k \xrightarrow{\psi,\text{reset}[\lambda]} l_{k+1}$. From the definition of $\models$ we have $\langle l_k, D_k \rangle \sim^\alpha \langle l_{k+1}, D_{k+1} \rangle$ by $l_k \xrightarrow{\psi,\text{reset}[\lambda]} l_{k+1}$ if $D_{k+1} \in (D_k \land I(l_k))^\uparrow \land I(l_k) \land \psi)$. Expansion by the definition of $\models$ and with the guard at the transition $l_{k+1}$ we get $D_{k+1} \in \text{Mcanon(MExtra(Mcanon((D_k \land I(l_k))^\uparrow \land I(l_k) \land \psi)(\lambda := 0))))}$. Expanding this by the reset operation we get $D_{k+1} \in \text{Mcanon(MExtra(Mcanon(Mcanon((D_k \land I(l_k))^\uparrow \land I(l_k) \land \psi)(\lambda := 0))))}$. Expand this by intersecting the resulting zone with the target invariant of location $k+1$ and extrapolate and canonicalize afterwards we get $D_{k+1} \in \text{Mcanon(MExtra(Mcanon(Mcanon(Mcanon((D_k \land I(l_k))^\uparrow \land I(l_k) \land \psi)(\lambda := 0)))) \land I(l_{k+1}))}$. From the semantic definition of the
zone approach we know that the second canonicalization operation will fix the non-tightness introduced by intersecting the zone with the guard $\psi$ and will not affect adversely the constraints involving $\delta$. Thus the value of the entries $D_{k+1}^{0,i}$ and $D_{k+1}^{i,0}$ in the matrix $(M_{\text{Canon}}(M_{\text{Canon}}((D_k \land I(l_k)) \land I(l_k) \land \psi)[\lambda := 0]))$ are still precise. From Lemma 5.3.2 we know that all the constraints involving $\delta$ will not be changed during the last canonicalization operation since the constraints involving $\delta$ will not be touched during extrapolation and that extrapolation only increases bounds while canonicalization computes a minimum.

From the semantic definition of $\land$ and the reset operation and that $\delta_i \not\in \lambda$ it is easy to see that the weight of the constraints involving $\delta$ in the matrix $M_{\text{Canon}}(M_{\text{Extra}}(M_{\text{Extra}}(M_{\text{Canon}}((D_k \land I(l_k)) \land I(l_k) \land \psi)[\lambda := 0]) \land I(l_{k+1})))$ are precise. Thus the value of the entries $D_{k+1}^{0,i}$ and $D_{k+1}^{i,0}$ after executing the action transition $(k + 1)$ represent the precise infimum and supremum accumulated delays of $A$ up to $(k + 1)$-transitions.

It remains to show that the partial extrapolation approach ensures termination and correctness when the run $r$ is infinite or when there is an infinite cycle in the behaviour of $A$. Note that the modified zone approach visits all the reachable states of an automaton while ignoring the value of the extra clock $\delta$ (the non-extrapolated clock) for termination. Termination is guaranteed also because there are finitely many sets of the form $M_{\text{Extra}}(\hat{D})$. Also by condition C4 we know that cycles in $TA$ will be treated differently during the analysis where the fixed point abstraction will be used rather than the inclusion abstraction when handling the generated zones inside cycles. This allows us to detect whether the cycle can lead to an infinite WCET (see Theorem 5.3.5). By Theorem 5.3.5 we know that it is guaranteed to reach a fixed point of infinite cycles after a finite number of iterations and in case the run visits an infinite cycle the extra clock will be set to $\infty$ which indicates that the upper bound for termination of $A$ is not bounded. Hence, the partial extrapolation approach ensures that the clock $\delta$ will be exact in the end even when there is a run with infinite length. \qed
Lemma 5.3.5. Let \( \pi \) be a cycle in an automaton \( A \) that can be run infinitely often. Then after a finite number of iterations a fixed point of \( \pi \) will be reached given that the \( M \)-extrapolation operation is applied during the construction of the zone graph.

Proof. In order to prove this Lemma we need to show that after a finite number of iterations of \( \pi \) while applying the \( M \)-extrapolation the search will reach a fixed point of the cycle. Let \( E_\pi = (e_0, ..., e_{m-1}) \) be the sequence of edges of the cycle \( \pi \) and that \( M(A) \) is the largest integer constant that appears in the guard and the location invariants of \( A \). Since \( \pi \) is an infinite cycle we can describe its behaviour using the recursive function \( f_\pi(Z^n_s) = \text{succ}(Z^n_s, E_\pi) = Z^{n+1}_s \), where \( Z^n_s \) is the corresponding zone at the start location of \( \pi \) at iteration \( n \). We need to show that the zone approach with the \( M \)-extrapolation operator guarantee the convergence of fixed point computations of \( \pi \). That is, there will be two distinct iterations \( j, k \) of \( \pi \) such that \( Z_j^i = Z_k^j \). Before proceeding further in the proof, let us recall how a zone \( Z \) is extrapolated when some clocks in \( Z \) exceeds the bound \( M(A) \). From the definition of \( M \)-extrapolation we know that when \( D_{0,y} < -M(A) \) the extrapolation function sets \( D_{0,y} \) to \( -M(A) \) and when \( D_{y,0} > M(A) \) it sets \( D_{y,0} \) to \( \infty \). Note that the domain of the lower bound of the clocks is still finite after extrapolation where for each clock \( y \in X \) we have \( (0 \leq D_{0,y} \leq 0) \). The domain of the upper bound of the clocks is also finite after extrapolation. It is necessary to note that \( \infty \) is just a special value that we assign to a clock variable when it exceeds the bound \( M(A) \). From the assumption that \( \pi \) is infinite we know that there will be an infinite sequence of zones of the form \( \text{MExtra}_M(Z^1_s), \text{MExtra}_M(Z^2_s), ..., \text{MExtra}_M(Z^n_s), .. \) every time \( \pi \) is executed. Now given that the number of clocks in \( A \) is bounded \((|X| < \infty) \) and the number of edges of \( \pi \) is bounded \((|E_\pi| < \infty) \) and that each clock has a finite domain where for each \( y \in X \) we have \((0 \leq D_{y,0} \leq M(A)) \) or \( D_{y,0} = \infty \) and \((-M(A) \leq D_{0,y} \leq 0) \) and from the fact that \( \pi \) is infinite in the sense that each location in \( \pi \) will be visited infinitely often then it is easy to see that there will be two distinct iterations \( i \) and \( j \) of \( \pi \) where \( \text{MExtra}_M(Z^i_s) = \text{MExtra}_M(Z^j_s) \) and hence \( \text{succ}(Z^i_s, E_\pi) = \text{succ}(Z^j_s, E_\pi) \).
On the other hand, for finite cycles the search will not reach a fixed point but there will be an iteration $i$ of a cycle $\pi$ where the search encounters a blocking edge (i.e. an edge that can not be executed since the guard of that edge gives inconsistent zone) and hence the cycle can not be repeated any further. However, since we seek a solution to the problem in the presence of extrapolation it is necessary then to ensure that the search does not leave finite cycles before executing them the precise number of times. More concretely, we need to ensure that if a finite cycle can be repeated $n$ times in the non-extrapolated graph, where $n < \infty$, then it can be repeated also $n$ times in the extrapolated graph and that the minimum and maximum total execution time of $\pi$ in the extrapolated graph are equal to those obtained in the non-extrapolated graph and hence the computations of WCET of finite cycles in the presence of extrapolation are precise.

**Theorem 5.3.6.** [Pet99, BY04] Let $(l_0, Z_0)$ be an initial state of an automaton $A$ where $l_0 \in L_0$ and $Z_0$ is the corresponding initial clock zone. Let $M = \max(A)$ be the maximal integer that appears in the guards and the location invariants of $A$ and $\Rightarrow_M$ be the transitions resulting from the $M$-extrapolation. Let $B(X)$ be the set of logical formulae generated by the syntax $g := y \sim c | g \land g$ where $\sim \in \{<, \leq\}$. Assume that $D^f \in B(X)$.

- (Soundness) whenever $(l_0, Z_0) \Rightarrow_M (l_f, Z_f)$ then $(l_0, Z_0) \Rightarrow (l_f, Z_f)$ for all $Z_f \in D^f$
- (Completeness) whenever $(l_0, Z_0) \Rightarrow (l_f, Z_f)$ then $(l_0, Z_0) \Rightarrow_M (l_f, Z_f)$ for all $Z_f \in D^f$

The following corollary is interesting since it shows that the precise time delays that the automaton can spend at each reachable location are respected by the $M$-extrapolation operator.

**Corollary 1.** Let $\pi$ be a cycle in an automaton $A$. If $\pi$ can be repeated $n$ times in the graph $Z(A)$ then $\pi$ can be repeated also $n$ times in the graph $M_{\text{Extra}}_M(Z(A))$
and that the lower and upper total delays of \( \pi \) in \( Z(\mathcal{A}) \) are equal to those obtained in \( \text{MExtra}_M(Z(\mathcal{A})) \).

**Proof.** To prove the first part of the Corollary we need to show that if the cycle \( \pi \) starts its first iteration with the zone \( Z_0^1 \) and that \( \pi \) can be repeated \( n \) times from the zone \( Z_0^1 \) then \( \pi \) can be repeated also \( n \) times if it starts with the zone \( \text{MExtra}(Z_0^1) \).

Let us assume that \( \pi \) has the sequence of edges \( e_0, \ldots, e_{m-1} \) where \( \text{src}(e_0) \) is the start location of \( \pi \). Assume further that \( M(\mathcal{A}) \) is the largest integer constant that appears in the guard and the location invariants of \( \mathcal{A} \). To prove this part we need to show that if the sequence \( Z_1^0, Z_1^1, \ldots, Z_1^{m-1}, \ldots, Z_n^0, Z_n^1, \ldots, Z_n^{m-1} \) can be constructed in the real semantic (i.e. non-extrapolated zone graph), where \( Z_i^j \) denotes the corresponding clock zone of the location \( \text{src}(e_i) \) at iteration \( j \), then the sequence \( \text{MExtra}_M(Z_1^0), \text{MExtra}_M(Z_1^1), \ldots, \text{MExtra}_M(Z_{m-1}^1), \ldots, \text{MExtra}_M(Z_0^n), \text{MExtra}_M(Z_1^n), \ldots, \text{MExtra}_M(Z_{m-1}^n) \) can be also constructed in the extrapolated semantic. Let \( \mathcal{B}(X) \) be the set of logical formulae that can be defined in the clock constraints of the automaton \( \mathcal{A} \). We need to show that whenever a clock constraint \( g \in \mathcal{B}(X) \) where \( g \) can be either an invariant or a guard in \( \pi \) is evaluated to \( b \) where \( b \in \{\text{true}, \text{false}\} \) using the zone \( Z_i^j \) then \( g \) is evaluated to \( b \) also using the corresponding extrapolated zone \( \text{MExtra}_M(Z_i^j) \). From Definition 2.9.8 we know that the extrapolated zones \( \text{MExtra}_M(Z_i^j) \) at any iteration \( j \) contain constants inbetween \(-M\) and \( M \) or \( \infty \) and hence for all \( i = 0, \ldots, m-1 \) we have \( Z_i \subseteq \text{MExtra}_M(Z_i) \). Since according to Definition 2.9.8 when a clock \( y \in X \) is extrapolated its lower bound \( Z_{0,y} \) is set to \(-M\) if \( Z_{0,y} < -M \) and its upper \( Z_{y,0} \) is set to \( \infty \) if \( Z_{y,0} > M \). From definition 2.9.8 also we know that for all possible time assignments \( u \in Z(I(\text{src}(e_i)))_{i=0,\ldots,m-1} \) and \( u \in Z(\psi(e_i))_{i=0,\ldots,m-1} \) we have max\(_y\)(\( u(y) = d \))\(_{y\in X} \) and \( d < M \). We need to show now that if we have a zone \( Z \) such that \( Z \models I(\text{src}(e_i)) \) for some \( e_i \in E_\pi \) then also \( \text{MExtra}_M(Z) \models I(\text{src}(e_i)) \). Let us denote the set of clocks that participate in \( I(\text{src}(e_i)) \) as \( Y \) where \( Y \subseteq X \). First recall that the invariants in TA can bound clocks only from above. Now since \( Z \models I(\text{src}(e_i)) \) we know that none of the clocks in \( Y \) exceeds the bound \( M \), otherwise we can not have \( Z \models I(\text{src}(e_i)) \). From Definition
we know that for all clock assignments $u$ we have $u \in Z(I(src(e_i)))_{i=0,\ldots,m-1}$ with $\max_y(u(y) = d)_{y \in X}$ and $d < M$. Given this and from Definition 2.9.8 we know that the clocks in $Y$ have not been extrapolated since the last time they have been reset and hence we can conclude that for all $y \in Y$ we have $Z_{y,0} = \text{MExtra}_M(Z_{y,0})$ and $Z_{0,y} = \text{MExtra}_M(Z_{0,y})$ and therefore $\text{MExtra}_M(Z) \models I(src(e_i))$. Next, we need to show that if $Z \models \psi(e_i)$ for some $e_i \in E$ then also $\text{MExtra}_M(Z) \models \psi(e_i)$. Note that since the guards in TA can bound clocks from below and from above we need then to consider two cases. The first case is when the guard $\psi(e_i)$ bounds clocks from above. In this case it is easy to see that if $Z \models \psi(e_i)$ then also $\text{MExtra}_M(Z) \models \psi(e_i)$ for the same reasoning given for the invariants case. The second case is when $\psi(e_i)$ bounds some clocks from below. This case is also trivial since $Z \subseteq \text{MExtra}_M(Z)$ and from the fact that the clocks that are bounded from below do no longer affect the evaluation of guards once they exceed their bounds and from the assumption that $Z \models \psi(e_i)$ we can conclude that $\text{MExtra}_M(Z) \models \psi(e_i)$. To complete the proof of this part we need to show also that if $Z \not\models I(src(e_i))$ for some $e_i \in E$ then also $\text{MExtra}_M(Z) \not\models I(src(e_i))$ and if $Z \not\models \psi(e_i)$ for some $e_i \in E$ then also $\text{MExtra}(Z) \not\models \psi(e_i)$. Proving this part is necessary since it shows that the cycle in the extrapolated semantic can not be executed more times than in the real semantic. However, this part is a bit tricky since as we know $Z \subseteq \text{MExtra}_M(Z)$ (i.e. extrapolation in fact enlarges zones). However, to prove this part we need to consider all possible forms that the invariants in TA can take and all possible forms that the guards in TA can take. Recall again that the invariants in TA can be of the form $\phi := (y \sim c | \phi \land \phi)$ where $\sim \in \{<,\leq,\geq\}$ (i.e. they only bound clocks from above). It is easy to see then that if $Z \not\models I(src(e_i))$ at iteration $j$ for some $e_i \in E$ then also $\text{MExtra}_M(Z) \not\models I(src(e_i))$ at iteration $j$ since $Z \subseteq \text{MExtra}_M(Z)$. However, the guards requires more careful analysis since they can take more forms than the invariants where the guards can be of the form $\phi := (y \sim c | \phi \land \phi)$ where $\sim \in \{<,\leq,=,>,\geq\}$. Again we need to consider here two cases since the guards can bound clock from below and from above. The first case to consider is when the guard $\psi(e_i)$ bounds only clocks from above. In this case it is easy to see that
if $Z \not\models \psi(e_i)$ then also $\text{MExtra}(Z) \not\models \psi(e_i)$ for the same reasoning given for the invariants case above. The second case is when the guard $\psi(e_i)$ bounds a set of clocks from below. Let us first denote the set of clocks that are bounded from below in $\psi(e_i)$ as $Y$. Now since the clock that is bounded from below has the form $y \sim c$ and $\sim \in \{>, \geq\}$ and from the assumption that $Z \not\models \psi(e_i)$ we know that there exists some $y \in Y$ such that $Z_{y,0} < c$ and $Z_{0,y} > -c$, where $c$ is the bound that $y$ is compared to in $\psi(e_i)$. This is because the evaluation of $\psi(e_i)$ fails by assumption. From the fact that the bound $c$ can not be greater than $M$ we know that there exists some $y \in Y$ such that $0 \leq Z_{y,0} \leq M$ and $-M \leq Z_{0,y} \leq 0$. This implies that in the extrapolated semantic there will be some clocks $y \in Y$ in the zone $\text{MExtra}_M(Z)$ that have not been extrapolated since the last time they have been reset and hence their values are equivalent to those in the real semantic. Given this and from the assumption that $Z \not\models \psi(e_i)$ we can conclude that $\text{MExtra}_M(Z) \not\models \psi(e_i)$ as well.

This completes the proof of the first part of the Corollary.

In order to prove the second part of the Corollary let us pick any iteration $1 \leq i \leq n$ of $\pi$ we need to show that if $\text{delay}(\pi, i) = T$, where $T$ has the form $\langle >a, b < \rangle$ where $0 \leq a \leq b \leq \infty$, in case the cycle at iteration $i$ starts with $Z_s^i$, then $\text{delay}_{\text{Extra}}(\pi, i) = T$ in case the cycle at iteration $i$ starts with $\text{MExtra}_M(Z_s^i)$. Let us introduce first another delay function $\text{delay}(\pi, i, \text{src}(e_j))$ that returns the delay interval that the cycle can spend at location $\text{src}(e_j)$ at iteration $i$. As induction hypothesis, assume $\Sigma_{j=1}^{k}(\text{delay}(\pi, i, \text{src}(e_j))) = \Sigma_{j=1}^{k}(\text{delay}_M(\pi, i, \text{src}(e_j)))$ up to $k$-transitions (i.e. $\langle l_0, D_0 \rangle \sim^{\alpha,k} \langle l_k, Z_k \rangle$, where $\alpha$ is either a delay or discrete action) where $\text{delay}(\pi, i, \text{src}(e_j))$ represents the time delay that $\pi$ can spend at location $l_j$ at iteration $i$ in the real semantic and $\text{delay}_M(\pi, i, \text{src}(e_j))$ is the same as $\text{delay}(\pi, i, \text{src}(e_j))$ but in the extrapolated semantic. Assume $\langle l_k, Z_k \rangle \xrightarrow{\psi, \text{reset}[\lambda]} \langle l_{k+1}, Z_{k+1} \rangle$. We want to show that after executing $\langle l_k, Z_k \rangle \xrightarrow{\psi, \text{reset}[\lambda]} \langle l_{k+1}, Z_{k+1} \rangle$ and $\langle l_k, \text{MExtra}_M(Z_k) \rangle \xrightarrow{\psi, \text{reset}[\lambda]} \langle l_{k+1}, \text{Extra}_M(Z_{k+1}) \rangle$ we have $\Sigma_{j=1}^{k+1}(\text{delay}(\pi, i, \text{src}(e_j))) = \Sigma_{j=1}^{k+1}(\text{delay}_M(\pi, i, \text{src}(e_j)))$. That is, the amount of time that elapses at step $(k + 1)$ in the real and extrapolated semantic is the same. From the zone approach
described at Section 2.9 we know that \( D_k \in (\text{MCanon}(\text{MCanon}(D_{k-1} \land I(l_{k-1}))^\oplus \land I(l_{k-1}) \land \psi)[\lambda_{k-1} := 0]) \) and \( \text{MExtra}_M(Z_k) \in (\text{MCanon}(\text{MExtra}_M(\text{MCanon}((Z_{k-1} \land I(l_{k-1}))^\oplus \land I(l_{k-1}) \land \psi)[\lambda_{k-1} := 0]))) \). We know also that the successor \( Z_{k+1} \) can be computed in the real semantic as \( Z_{k+1} \in (\text{MCanon}(\text{MCanon}(Z_k \land I(l_k)) \land I(l_k) \land \psi)[\lambda_k := 0]) \) and in the extrapolated semantic as \( \text{MExtra}_M(Z_{k+1}) \in (\text{MCanon}(\text{MExtra}_M(\text{MCanon}((Z_k \land I(l_k))^\oplus \land I(l_k) \land \psi)[\lambda_k := 0]))) \). Let \( I(l_k) \in B(X) \) be the clock invariant at location \( l_k \) and \( \psi(e_k) \in B(X) \) be the guard at transition \( e_k \). Again from Definition 2.9.8 we know that the extrapolated zone \( \text{MExtra}_M(Z_k) \) contains constants inbetween \(-M\) and \( M \) or \( \infty \) and hence \( Z_k \subseteq \text{MExtra}_M(Z_k) \). From definition 2.9.8 also we know that for all possible time assignments \( u \in D_{I(l_k)} \) and \( u \in D_{\psi(e_k)} \) we have \( \max_y(u(y) = d)_{y \in X} \) and \( d < M \). From the fact that the extrapolation operator defined in Definition 2.9.8 does not affect time assignment \( u \in Z_{I(l_k)} \) and \( u \in Z_{\psi(e_k)} \) with \( \max_y(u(y) < M)_{y \in X} \) and from the monotonicity of the \( \uparrow \) operator and the monotonicity of the reset-operator and from the definition of the \( \land \) and the definition of the canonicalization operation it is easy to see that the precise delay interval that \( \mathcal{A} \) can spend at the location \( l_k \) is respected by the operator \( M \).

Therefore, \( \Sigma_{j=1}^{k+1}(\text{delay}(\pi, i, \text{src}(e_j))) = \Sigma_{j=1}^{k+1}(\text{delay}_{\text{MExtra}}(\pi, i, \text{src}(e_j))) \) and hence we can conclude that if \( \text{delay}(\pi, i) = T \) then also \( \text{delay}_{\text{Extra}}(\pi, i) = T \). \( \Box \)

5.4 A Zone-based Algorithm for Computing WCET of TA

Algorithm 8 (\text{WCETAlgor}) gives a zone-based algorithm for calculating the WCET of real-time distributed systems. The algorithm takes as input an automaton \( \mathcal{A} \) for the system to be analyzed. Each node in the computed tree is of the form \((l_i, D_i, sts)\) where \( l_i \) is a location in the automaton, \( D_i \) is the corresponding zone, and \( sts \) is an integer variable which is assigned to each state in order to detect whether there exists a cycle on locations in the behaviour of the automaton. The variable \( sts \) can take values from the set \( \{0, 1, 2\} \). When it is 0 it means that the location has
CHAPTER 5. WCET OF ARBITRARY TIMED AUTOMATA

not been visited before, when it is 1 it means the location has been visited before but not fully explored, and when it is 2 it means that everything reachable from that location have been explored. The variable $sts$ is updated as described in the classical DFS algorithm and we assume that the reader is familiar with the DFS algorithm with the labelling process of nodes to unvisited (0), being explored (1), and finished (2) and hence we omit these details. The algorithm uses two data structures WAIT and PASSED to store symbolic states waiting to be examined, and the states that already examined, respectively. The WAIT set is instantiated with the initial symbolic state $(l_0, D_0, 0)$. The global variable $WCET$ holds the currently best known longest execution time of reaching the final location; initially it is 0. The global clock $\delta$ keeps track of the execution time of the system. In each iteration of the while loop, the algorithm selects a symbolic state $s$ from WAIT, checking if the state is a final state. If the state does not evolve to any new state then we consider it as a final state of some branch in the graph. If the state $s$ is a final state we update the best known $WCET$ to the upper bound value of $\delta$ at $s$ if it is greater than the current value of $WCET$. If the state is not a final state, we add all successors of $s$ to WAIT and continue to the next iteration. During the search, if the algorithm discovers that there exists an infinite cycle in the automaton in which time can progress then it stops immediately since the WCET will be infinity. On the other hand, if the search encounters an infinite cycle in which time can not progress then the search will skip that cycle. However, it is interesting to note that the extrapolation procedure used in the algorithm is the partial extrapolation that satisfies conditions C1-C4. Note that we write $s.\dot{D}$ to refer to the sub-DBM $\dot{D}$ that consists in the dot entries in the matrix $D$ (i.e. the entries involving the automaton clocks) in the state $s$.

Note that we do not use the inclusion abstraction [BY04] in the algorithm since inclusion abstraction may lead to stop the exploration when some zone is “included” inside an already explored zone. This way of cutting exploration based on inclusion is correct for reachability properties, but not when properties critically involve
5.4. A ZONE-BASED ALGORITHM FOR COMPUTING WCET OF TA

Theorem 5.4.1. The zone-based Algorithm 8 computes correctly the WCET of diagonal-free TA and guarantees termination.

Proof. Theorem 5.4.1 can be proved by induction on the length of transition sequences. However, the proof of the theorem is a straightforward combination of Theorem 5.3.4 and Lemma 5.3.5 and Corollary 1. From Theorem 5.3.4 (the partial extrapolation theorem) we know that the extra clock $\delta$ remains precise to the end and is not influenced by extrapolation. From Lemma 5.3.5 we know that if there is an infinite cycle in the behaviour of $\mathcal{A}$ then it will be detected during the analysis since the algorithm checks at each iteration of the loop whether the search has reached a fixed-point of the discovered cycle. Now by checking the upper bound of the extra clock (the non-extrapolated clock) the algorithm can detect whether the reached infinite cycle can lead to an infinite WCET. From Corollary 1 we know that if there is a finite cycle in the behaviour of $\mathcal{A}$ then the cycle will be repeated the maximum allowed number of times and that the precise delays at each visited
location are respected by partial extrapolation and hence it guarantees the precise calculations of WCET. Finally, termination is ensured since there are finitely many sets of the form $M_{\text{Extra}}(D)$ where the algorithm visits all the reachable states of an automaton while ignoring the value of the extra clock $\delta$ for termination.

\[ \square \]

5.5 Implementation

In this section we briefly summarise our prototype implementation of the model checking algorithms given in Section 5.4. It is important to note that the goal of our implementation is to validate the presented algorithms, rather than to devise an efficient implementation; this will be the subject of our future work.

The prototype implementation has been developed using the opaal tool [DHJ+11] (http://opaal-modelchecker.com) which has been designed to rapidly prototype new model checking algorithms. The opaal tool is implemented in Python and is a standalone model checking engine. Models are specified using the UPPAAL XML format. We use the open source UPPAAL DBM library for the internal symbolic representation of time zones in the algorithms.

We have verified the WCET of the four TA given in Figures 12, 13, 14, and 15, which have an infinite WCET. The algorithm handles successfully these TA in a very reasonable time where each one of them has been verified in a few seconds. On the other hand, UPPAAL fails to terminate when verifying the automata in Figures 12, 13, 14, and 15 and hence no answer has been obtained. In fact, we have verified in UPPAAL several other examples of TA which show that when infinite cycles exist, the algorithm implemented in UPPAAL may not terminate.
5.6 Conclusion

In this chapter we proposed an algorithm for determining the WCET in timed automata that can work on any arbitrary TA including those containing infinite realizable cycles. The algorithm uses a modified abstraction which we call partial extrapolation that keeps the extra clock precise to the end of the analysis. A key issue with the verification of WCET of TA is the use of fixed point abstraction since the use of this abstraction helps to detect on-the-fly infinite cycles in TA. It is why our algorithm succeeds to terminate even when infinite cycles exist, while other tools like UPPAAL, which is based on similar technology, does not guarantee termination. Note that the proposed algorithm may not perform well when largely repetitive finite cycles exist. However, we address this issue in chapter 7 where we develop acceleration techniques that improve the computation of WCET of TA with large number of cycle iterations.
Chapter 6

Partial Canonicalization of Clock Zones

Key operations of the zone abstraction in timed automata are canonicalization and extrapolation. These operations are routinely performed in model checking timed automata. Canonicalizations are needed for the comparison of zones and for efficient implementation of several constraint operations, extrapolation is needed to guarantee the finiteness of the zone graph. This chapter presents a new operation for improving the reachability analysis and the WCET analysis of Timed Automata (TA) using the Difference Bound Matrices (DBMs), namely the partial canonicalization of DBMs. The partial canonicalization allows one to fix the non-tightness introduced by extrapolation by updating only the clock constraints that have been changed during extrapolation, thus reducing the impact on the run time of the reachability algorithm and the WCET algorithm of the canonicalization step. The proposed partial canonicalization algorithms are specializations of Floyd’s algorithm that have a time complexity of $O(c \times n)$ instead of $O(n^3)$ for the standard approach, where $n$ is the number of clocks in the automaton and $c$ is the number of variables that have been changed during extrapolation. We demonstrate that model checking TA with partial canonicalization can speed-up considerably the verification time of
several interesting examples including the Philips audio control protocol and Fischer’s protocol.

6.1 Introduction

Since their introduction by Alur and Dill [AD94], timed automata (TA) have become one of the most well-known models for real-time systems with well-established theory and development of model checking tools such as UPPAAL, Kronos, and RED. The Alur-Dill timed automata model has not only been used for verifying safety and liveness properties of timed systems but also they have been used for solving several other interesting problems such as scheduling, code synthesis, optimal resource analyses, and parametric analysis. However, in the semantic definition of timed automata, clock constraints in the invariant of a location or in the guard of a transition can contain arbitrary non-negative real numbers and, at first glance, this would seem to make the model checking task intractable as the number of states is infinite. Therefore, most real-time model checking tools apply extrapolation algorithms based on so-called zones in order to guarantee termination of the analysis.

By extrapolation, we mean the process of transforming zones (i.e. sets of valuations of the timed automaton clocks) that contain arbitrarily large constants to equivalent zones containing only bounded constants. Given a zone $Z$ with a set of clock constraints $C = \{c_1, .., c_n\}$, then all the constraints $c_i$ that exceed a certain bound $M$ derived from the graph of the automaton under analysis will be extrapolated (enlarged). The observation is that if the clock in an automaton is never compared in a guard or invariant to a constant greater than $M$, then the value of the clock will have no impact on the computation of the automaton $A$ once it exceeds $M$ [DT98, BY04]. This operation is known in the literature as the $M$-extrapolation operation and it is a standard operation implemented in several tools like UPPAAL and Kronos. However, in [BBLR06] it has been observed that by distinguishing
the maximal lower and upper bounds to which clocks of the timed automaton are compared one can obtain a significantly coarser abstraction of TA.

Since the representation of a clock zone by a difference bound matrix (DBM) [Dil90] is not unique (i.e. there are an infinite number of zones sharing the same solution set), it is necessary to have a unique representation of the zone where no atomic constraint in the matrix representing the zone can be tightened any further. At which we obtain what we call a canonical form of the matrix. In fact, canonical forms simplify some operations over zones like the test for inclusion between zones and checking whether a given zone is nonempty. Canonical forms are also important for efficient implementation of several constraint operations such as delays and resets.

So by canonicalization, we mean the process of computing the tightest possible bound for each pair of clocks in a zone which is equivalent to finding the shortest path between their nodes in the graph interpretation of the zone.

It is commonly agreed that extrapolation does not preserve the canonical form of a DBM (i.e. a form of the matrix in which each atomic constraint is in its tightest form) and the best way to put the matrix back on canonical form is to use the Floyd-Warshall algorithm [Pet99, BY04]. However, extrapolation does not always break the canonical form of the entire matrix. That is, part of the matrix may maintain its canonical form after extrapolation and hence it is not always necessary to re-canonicalize the entire matrix every time extrapolation is performed. A careful look at the way the extrapolation functions [DT98, BBLR06] change the bounds in the matrix show that in certain circumstances the standard Floyd-Warshall algorithm may perform some unnecessary computations when canonicalizing an extrapolated matrix. This may lead to unnecessary overhead during the analysis of TA. So by \textit{partial canonicalization}, we mean the process of fixing the non-tightness introduced by extrapolation by recanonicalizing only the set of constraints that have been changed during extrapolation since the constraints that are not touched during extrapolation remain tight. The key observation is that extrapolation only increases bounds, never decreases them, while canonicalization only decreases bounds, never
6.1. INTRODUCTION

increases them. Hence, when a constraint $p$ in a canonical DBM $D$ is not touched (enlarged) during extrapolation then the value of $p$ in the canonicalized extrapolated matrix can not be smaller than its value in the canonicalized non-extrapolated matrix and therefore no need to tighten $p$.

Based on the above observation we develop partial canonicalization algorithms, which are specializations of Floyd’s algorithm, that can run in $O(c \times n)$ instead of $O(n^3)$ for the standard approach, where $n$ is the number of clocks in the automaton under analysis and $c$ is the number of constraints that have been changed during extrapolation. During our analysis, we detect also some special cases in which extrapolation does not break the canonical form of the matrix and hence there is no need to recanonicalize the matrix after extrapolation. More interestingly, when studying the four well-known extrapolation procedures: the $M$-extrapolation procedure [DT98], the $LU$-extrapolation procedure [BBLR06], the $M^+$-extrapolation procedure [BBLR06], and the $LU^+$-extrapolation procedure [BBLR06], we find that the cost of fixing the non-tightness introduced by extrapolation can vary depending on the specific applied extrapolation procedure. That is, we find that the cost of tightening an $M^+$-extrapolated matrix or an $LU^+$-extrapolated matrix is much cheaper than the cost of tightening an $M$-extrapolated matrix or an $LU$-extrapolated matrix. In fact, the $M^+$-extrapolation procedure and the $LU^+$-extrapolation procedure have some unique characteristics that make the cost of tightening them much cheaper than the other procedures.

We discuss also some lack of precision in the canonicalization and extrapolation procedures used in previously published algorithms for timed automata, where surprisingly we discover that reducing an extrapolated matrix to canonical form can give a non-extrapolated matrix. This may cause problems when checking inclusion between zones. According to [Pet99, BBFL03, BY04, Bou04, BLR05b, BBLR06] the test for inclusion of DBMs should be checked syntactically on the canonical form of DBMs and hence the extrapolated DBMs need to be canonicalized before
checking inclusion between them. Recall that extrapolation together with inclusion checking are used to enforce the termination of the forward searching in the timed automaton. However, we show that checking inclusion between DBMs using the non-canonical extrapolated DBMs is a safe operation since termination using this form of test is guaranteed, while on the other hand, checking inclusion between DBMs using the canonical extrapolated DBMs is an unsafe operation since termination may not be guaranteed. This may look surprising to the reader. This is mainly due to the fact that canonicalizing extrapolated DBMs can give non-extrapolated DBMs which can cause problems during inclusion checking and hence it may affect adversely termination of the analysis. We then propose some modifications to the standard zone approach that improve both efficiency and reliability of the approach.

The structure of the chapter is as follows. We begin in Section 2 by reviewing the existing extrapolation procedures of TA and discuss their role in forward reachability algorithms. In Section 3, we introduce what we call partial canonicalization of DBMs that allows one to fix the non-tightness introduced by extrapolation by canonicalizing only the constraints that have been changed during extrapolation. We then describe a specialisation of Floyd’s algorithm for the \( M \)-extrapolation and the \( LU \)-extrapolation procedures and another specialisation of Floyd’s algorithm for the \( M^+ \)-extrapolation and the \( LU^+ \)-extrapolation procedures. We then describe a new form of inclusion checking between DBMs using un-canonicalized versions of extrapolated DBMs and prove its correctness. In Section 5, we describe an implementation of the reachability algorithm using the new canonicalization procedures and describe the associated verification results on a set of examples. Finally, in Section 6, we draw some conclusions and discuss future directions.

### 6.2 Extrapolation Algorithms for Timed Automata

We now review the existing extrapolation algorithms for TA. To obtain a finite zone graph most model checkers use some kind of extrapolation of zones. In the last few
years, there has been a considerable development in the extrapolation procedure for TA for the purpose of providing coarser abstractions of TA [Rok93, BBFL03, BY04, Bou04, BBLR06, HKSW11, MS12]. Note that the extrapolation algorithm must be sound, complete, and finite. The soundness means that if an initial state \((l_0, Z_0)\) leads to a final state \((l_f, Z_f)\) in the concrete or real operational semantics, it should be possible to conclude this in the extrapolated operational semantic. The finiteness means that the transition relation \(\sim_{\text{Extr}}\) is finite and hence the generated state space of the extrapolated TA is finite.

### 6.2.1 Classical Maximal Bounds.

One of the first proposed extrapolation algorithms for TA is the so-called \(M\)-extrapolation [DT98], i.e. the zone is extrapolated with respect to the maximum constant each clock is compared to in the automaton. That is, if the clock is never compared to a constant greater than \(M\) in a guard or invariant, then the value of the clock will have no impact on the computation of the automaton once it exceeds \(M\). The \(M\)-extrapolation algorithm has been implemented in the early version of UPPAAL [BDL04]. The procedure to obtain the \(M\)-extrapolation of a given zone is to remove all upper bounds higher than the maximum constant and lowering all lower bounds higher than the maximum constant down to the maximum constant.

**Definition 6.2.1.** Let \(Z\) be a zone represented by a DBM in a canonical form \(D = (m_{i,j}, \prec_{i,j})_{i,j=0..n}\) and \(M\) be the maximum constant appearing in the invariants or guards of the automaton. We can compute the extrapolation function \(\text{Extra}_M(D')\) of the zone \(D' = (m'_{i,j}, \prec'_{i,j})_{i,j=0..n}\) as follows:

\[
(m'_{i,j}, \prec'_{i,j}) = \begin{cases} 
(\infty, <) & \text{if } m_{i,j} > M, \\
(-M, <) & \text{if } m_{i,j} < -M, \\
(m_{i,j}, \prec_{i,j}) & \text{otherwise}.
\end{cases}
\]

**Lemma 6.2.2.** [Pet99] For diagonal-free TA, the symbolic set \((l, \text{Extra}_M(Z))\) and
the transitions $\sim_M$ resulting from the $M$-extrapolation are sound and complete with respect to reachability and the transition relation is finite.

A maximal constant can be computed for each clock in the automaton in a similar way, which could make the state space much smaller. A considerable gain in efficiency can be obtained by analysing the graph of the automaton and calculating maximum bounds specific for each clock and state of the automaton [BBFL03]. That is, the maximum constants not only depend of the particular clock but also of the particular location of the TA. An even more efficient approach is the so called $LU$-extrapolation that distinguishes between upper and lower bounds [BBLR06]. This is the method used in the current implementation of UPPAAL.

### 6.2.2 Lower and Upper Maximal Bounds.

In [BBLR06] it has been observed that by distinguishing the maximal lower and upper bounds to which clocks of the timed automaton are compared one can obtain a significantly coarser abstraction of TA.

**Definition 6.2.3.** Let $Z$ be a zone represented by a DBM in a canonical form $D = (m_{i,j}, \prec_{i,j})_{i,j=0...n}$. For each clock $x_i \in X$ in $\mathcal{A}$ the maximal lower bound $L(x_i)$, (resp. maximal upper bound of $x_i U(x_i)$) is the maximal constant $M$ such that there exists a constraint $x > M$ or $x \geq M$ (resp. $x < M$ or $x \leq M$) in a guard of a transition or in an invariant of some location in $\mathcal{A}$. If such a constant does not exist, we set $L(x_i)$, (resp. $U(x_i)$) to $-\infty$. The $LU$-extrapolation of the zone $D' = (m'_{i,j}, \prec'_{i,j})_{i,j=0...n}$ can be computed as follows.

$$(m'_{i,j}, \prec'_{i,j}) = \begin{cases} 
\infty & \text{if } m_{i,j} > L(x_i), \\
-U(x_j), < & \text{if } -m_{i,j} > U(x_i), \\
(m_{i,j}, \prec_{i,j}) & \text{otherwise.}
\end{cases}$$

Note that the $LU$-extrapolation benefits from the properties of the two different maximal bounds. It does generalise the $M$-extrapolation (i.e. $\forall x \in X (M(x) =$
max(L(x), U(x))). For every zone Z, it holds that Z ⊆ Extra_M(Z) ⊆ Extra_LU(Z) [BBLR06]. The experiments given in [BBLR06] demonstrate the significant speedup obtained from using lower and upper bounds of clocks in the abstraction. Note that the M-extrapolation and the LU-extrapolation operations will not preserve the canonical form of the DBM, and in this case the best way to put the result back on canonical form is to use the Floyd-Warshall algorithm.

**Lemma 6.2.4.** [BBLR06] For diagonal-free TA the LU-extrapolation is sound, complete, finite with respect to reachability and effectively computable.

However, in [BBLR06] the authors have discussed also two other extrapolation procedures that can provide coarser abstraction of TA, namely Extra^+_M(Z) and Extra^+_LU(Z). The improvement proposed in these procedures is based on the observation that when the whole zone is above the maximum bound of some clock, then one can remove some of the diagonal constraints of the zones, even if they are not themselves above the maximal bound. Formally, we can define the Extra^+_M(Z) operation as follows.

\[
(m'_{i,j}; \prec'_{i,j}) = \begin{cases} 
\infty & \text{if } m_{i,j} > M(x_i), \\
\infty & \text{if } -m_{0,i} > M(x_i), \\
\infty & \text{if } -m_{0,j} > M(x_j), i \neq 0 \\
(-M(x_j), <) & \text{if } -m_{i,j} > M(x_j), i = 0 \\
(m_{i,j}, \prec_{i,j}) & \text{otherwise.}
\end{cases}
\]

Similarly, we can define the Extra^+_LU(Z) operation as follows.

\[
(m'_{i,j}; \prec'_{i,j}) = \begin{cases} 
\infty & \text{if } m_{i,j} > L(x_i), \\
\infty & \text{if } -m_{0,i} > L(x_i), \\
\infty & \text{if } -m_{0,j} > U(x_j), i \neq 0 \\
(-U(x_j), <) & \text{if } -m_{i,j} > U(x_j), i = 0 \\
(m_{i,j}, \prec_{i,j}) & \text{otherwise.}
\end{cases}
\]
Note that the operators $\text{Extra}_M^+(Z)$ and $\text{Extra}_{LU}^+(Z)$ set a lot of constraints to infinity during extrapolation. In fact, this can be advantageous when canonicalizing an $M^+$-extrapolated matrix or $LU^+$-extrapolated matrix. As we can see from the properties of the operators $\text{Extra}_M^+(Z)$ and $\text{Extra}_{LU}^+(Z)$ one can develop a highly efficient algorithm for canonicalizing $M^+$-extrapolated matrices and $LU^+$-extrapolated matrices.

It is commonly agreed that extrapolation does not preserve the canonical form of the matrix and the best way to put the matrix back in canonical form is to use the Floyd-Warshall algorithm [BBFL03, BY04, Bou04, BBLR06]. However, it has been claimed in [Pet99, BBFL03, BY04, Bou04, BLR05b] that the standard zone approach (see Section 2.10) will give in the end an extrapolated zone graph. That is, if the $M$-extrapolation is applied during the analysis then the resulting graph will be in $M$-form (i.e. all the constraints in all the generated DBMs of the graph are $M$-bounded) and if the $LU$-extrapolation is applied then the resulting graph will be in $LU$-form (i.e. all the constraints in all the generated DBMs of the graph are $LU$-bounded). This claim is only true if the extrapolated zones represented as DBMs do not lose their extrapolated form during canonicalization. The authors of [Pet99, BBFL03, BY04, Bou04, BLR05b] have not reported that canonicalizing an extrapolated matrix can give a non-extrapolated matrix and hence the final resulting graph may not be in an extrapolated form. To explain this issue we need some preliminary observations. The first observation is that extrapolation only increases bounds, never decreases them. The second observation is that canonicalization only decreases bounds, never increases them. To explain formally the issue let us assume that we have a canonical matrix $D$ and that the matrix $D$ has been extrapolated using the standard $M$-extrapolation procedure and that the constraint $(D_{i,j}, \prec_{i,j})$ has been changed (enlarged) during extrapolation. Suppose further that the constraint $(D_{i,j}, \prec_{i,j})$ has been reduced during canonicalization. Let us denote the constraint $(D_{i,j}, \prec_{i,j})$ after extrapolation and canonicalization as $(D'_{i,j}, \prec'_{i,j})$. Then we have two cases to consider.
• If the bound $D_{i,j}$ is set to $\infty$ during extrapolation (enlargement) then we know that its value before extrapolation was greater than $M$ (see Definition 6.2.1). Now since we assume that $D_{i,j}$ has been reduced during canonicalization then we know that $M < D'_{i,j} < \infty$. Note that $D'_{i,j}$ can not be smaller than or equal to $M$ since $D \subseteq D'$ and $D_{i,j} > M$. This is because extrapolation only increases bounds. In this case we can see that if the bound $D_{i,j}$ is reduced during canonicalization then the resulting matrix will not be in $M$-form since $D'_{i,j}$ will be greater than $M$.

• If the bound $D_{i,j}$ is set to $-M$ during extrapolation (enlargement) then we know that its value before extrapolation was less than $-M$ (see Definition 6.2.1). Now since we assume that $D_{i,j}$ has been reduced during canonicalization then we know that $D'_{i,j} < -M$. Note that $D'_{i,j}$ can not be greater than or equal to $-M$ since canonicalization only decreases bounds, never increases them. In this case we can see that if the bound $D_{i,j}$ is reduced during canonicalization then the resulting matrix will not be in $M$-form since $D'_{i,j}$ will be smaller than $-M$.

From the above discussion it is easy to see that if a single constraint in the extrapolated matrix is reduced during canonicalization then the resulting matrix will no longer be in an extrapolated form. Unfortunately, the majority of the existing extrapolation procedures including the $M$-extrapolation and the $LU$-extrapolation do not satisfy the property that when an extrapolated DBM is reduced to canonical form it preserves its extrapolated form. Recall that after extrapolation and canonicalization of DBMs a test for inclusion will be performed in order to check whether the new generated DBMs are already covered by some previously generated DBMs. Note that the test for inclusion of DBMs is syntactically checked on the canonical form of DBMs so the extrapolated DBMs need to be canonicalized before inclusion checking is performed. But this can give non-extrapolated DBMs. The question is then does this affect termination of the forward reachability analysis of TA? Does it cause problems with inclusion checking?
We now give an example of an automaton by which we demonstrate how reducing extrapolated zones to canonical form can yield non-extrapolated zones which can affect adversely termination of the analysis. The automaton in Figure 16 has three infinite loops loop1, loop2, and loop3 and three clocks $x$, $y$, and $z$. Let us see what happens when we compute the zones of the automaton using the standard zone approach [Pet99, BBFL03, BY04, Bou04, BLR05b]. Firstly, note that the extrapolation constants of the automaton are $(M_x = 5, M_y = 15, M_z = 250)$. We give now the sequence of zones obtained below. Note that for convenience only the full canonical zone is written. First at location start we have the zone $(x = 0 \land y = 0 \land z = 0)$. During the forward traversal of the TA the location loop1 is reached with the clock zone $(x \leq 0 \land y = z = 5)$. Clearly, extrapolation is not necessary here since none of the constraints exceeds its corresponding extrapolation constant. After taking the transition $\text{loop1} \rightarrow \text{loop1}$ four times a state $(\text{loop}, Z)$ with $Z = (x \leq 0 \land y = z = 20 \land (x_0 - y) \leq -20)$ will be added. Before proceeding further, note that the zone $Z$ needs to be extrapolated since the clock $y$ exceeds its corresponding extrapolation constant. We get then the zone $Z' = (x \leq 0 \land y = \infty \land z \leq 20 \land (x_0 - y) \leq -15)$. Since the zone $Z'$ is not in canonical form we need to tighten it. Using the Floyd-Warshall algorithm we get the new zone $Z'' = (x \leq 0 \land y \leq 20 \land z \leq 20 \land (x_0 - y) \leq -20)$ which is exactly the zone we have before extrapolation. Note that the zone $Z''$ is not in an extrapolated form since again the clock $y$ exceeds its corresponding extrapolation constant. This means that we have not benefited from the previous extrapolation step since when we canonicalized the extrapolated zone $Z'$ we obtained the same zone that we have before extrapolation. Note that this can affect adversely termination in TA since the standard test for inclusion of zones is syntactically checked on the canonical form of extrapolated zones. Note also that the resulting final zone graph will not be in extrapolated form as claimed in the prior literature since canonicalizing extrapolated zones will give non-extrapolated zones.

Note that if the extrapolation constant $M_z$ is too large then the loops will be taken
6.3. PARTIAL CANONICALIZATION OF DIFFERENCE BOUND MATRICES

A large number of times enlarging the state space in a way it slows down termination of the analysis. In fact, the analysis of TA using the standard form of inclusion checking based on canonicalized extrapolated zones may suffer from the state space explosion, which may lead to a low efficiency or resource exhaustion when cycles or loops exist. Note that the problematic step is to reduce the extrapolated DBMs to canonical form before checking inclusion between DBMs since this gives non-extrapolated DBMs. However, as we will show in the following section checking inclusion between DBMs using un-canonicalized extrapolated DBMs (i.e. extrapolated DBMs that are not necessarily in canonical form) is safe and it guarantees the correct termination of the analysis. This is what we prove in the following section where we describe improved canonicalization algorithms for fixing the non-tightness introduced by extrapolation. However, checking inclusion between DBMs using un-canonicalized extrapolated DBMs requires also some modifications to the standard zone approach and to the standard reachability algorithms.

6.3 Partial Canonicalization of Difference Bound Matrices

As one can see from the zone approach described in Chapter 2 there are places where one can develop specialised canonicalization algorithms that can fix the non-tightness introduced by extrapolating a clock or applying a delay which could be cheaper than $O(n^3)$ for standard Floyd-Warshall algorithm. It is important to
note that the 8-steps of the zone approach described in Chapter 2 will be repeated at each transition of the graph and that canonicalization may be repeated three times at each step of the successor computation. Note also that computation of canonical forms is a very expensive operation and avoiding them or reducing their costs wherever possible will accrue gains in performance. However, the work in [ZLZ05] presents an algorithm that improves the canonicalization of the matrix after the intersection operation which reduces the complexity to \( O(n^2) \). Hence, the first two canonicalization operations in the standard zone approach (see Section 2.10) can be performed in \( O(n^2) \). However, to our knowledge there is no special procedure for improving the canonicalization of the matrix after extrapolation.

Before discussing a procedure for improving the canonicalization of the matrix after extrapolation it is necessary first to review the main characteristics of extrapolation and canonicalization operations. Note that extrapolation only increases bounds, never decreases them and hence we say that extrapolation enlarges zones. On the other hand, canonicalization only decreases bounds, never increases them. During canonicalization, a minimum is calculated, thus the smallest bound dominates all others. It is easy to see that if a constraint \( p \) in a canonical DBM \( D \) has not been touched during extrapolation then the value of \( p \) in the canonicalized extrapolated matrix can not be smaller than its value in the canonicalized unextrapolated matrix since extrapolation increases bounds and never lowers them (i.e. \( D \subseteq \text{Extra}(D) \)).

The above observations lead to the following lemma which is interesting since it leads to a special canonicalization procedure which can help sometimes in reducing substantially the cost of fixing the non-tightness introduced by extrapolation.

**Lemma 6.3.1.** Let \( D \) be a DBM in canonical form. Let \((c_{i,j}, \prec_{i,j})\) be a constraint in \( D \). Suppose that the DBM \( D \) has been extrapolated using the \( M \)-extrapolation procedure and that the constraint \((c_{i,j}, \prec_{i,j})\) has not been changed during extrapolation. Let us denote the matrix \( D \) after being extrapolated and canonicalized as \( D' \). Then the value of \((c_{i,j}, \prec_{i,j})\) in the canonical matrix \( D \) is equal to its value in the canonicalized extrapolated matrix \( D' \) and hence \((c_{i,j}, \prec_{i,j})\) does not need to be
6.3. PARTIAL CANONICALIZATION OF DIFFERENCE BOUND MATRICES

recanonicalized after extrapolation.

**Proof.** Let us denote the constraint \((c_{i,j}, \prec_{i,j})\) after extrapolating and canonic-

alizing it as \((c'_{i,j}, \prec'_{i,j})\). We need to show that \((c_{i,j}, \prec_{i,j}) = (c'_{i,j}, \prec'_{i,j})\). From the as-

sumption that \(D\) is on canonical form and by definition of the tightening algorithm

[Rok93] we know that the weight of the constraint \((c_{i,j}, \prec_{i,j})\) in \(D\) is the tightest

weight that can be derived from the set of constraints in \(D\). By the definition of

extrapolation we know that the bounds of the constraints that are extrapolated are

in fact increased. Note that when the bound of a constraint is above the extrapo-

lation constant \(M\) then the extrapolation function \(\text{Extra}_M(D)\) sets it to \(\infty\), which

is an increase, and when the bound is less than \(-M\) it sets it to \(-M\), which is still

an increase, and therefore the function \(\text{Extra}_M(D)\) only increases bounds. Now

since the constraint \((c_{i,j}, \prec_{i,j})\) has not been increased during extrapolation and that

\(D \subseteq D'\) and the function \(\text{Canon}(D)\) computes a minimum it is easy to see then that

\(c_{i,j} = c'_{i,j}\) and hence no need to recanonicalize \((c_{i,j}, \prec_{i,j})\).

To make the presentation more formal let us assume that we have an automaton \(A\)

with the set of clocks \(\{x_1, \ldots, x_{n-1}\}\). Suppose that at a particular location \(l\) in \(A\) we

obtain the extrapolated matrix \(D\) as described below, where the dash sign represents

a constraint that has been extrapolated and the asterisk sign represents a constraint

that has not been touched during extrapolation. Let us denote the sub-DBM that

consists in the dash constraints as \(\bar{D}\) and the sub-DBM that consists in the asterisk

constraints as \(\tilde{D}\). Then to fix the non-tightness introduced by extrapolation it

suffices to canonicalize the sub-DBM \(\bar{D}\).

\[
D = \begin{pmatrix}
  x_0 & x_1 & \cdots & x_{n-1} \\
  x_0 & 0 & \cdots & - \\
  x_1 & * & 0 & \cdots & * \\
  \vdots & \vdots & \ddots & \vdots \\
  x_{n-1} & - & * & \cdots & 0
\end{pmatrix}
\]
Lemma 6.3.1 leads to the following theorem which states that after extrapolation the dash entries in the matrix (the entries that have been extrapolated) need to be recanonicalized while the asterisk entries (the entries that have not been touched during extrapolation) need not to be recanonicalized since they remain tight.

**Theorem 6.3.2.** Let $D$ be a DBM in canonical form. Suppose that $D$ has been partially extrapolated so that some constraints in $D$ have been changed. Let $\tilde{D}$ be the sub-DBM that consists in the constraints that have been extrapolated and $\hat{D}$ be the sub-DBM that consists in the constraints that have not been touched during extrapolation. Then to fix the non-tightness introduced by extrapolation it suffices to recanonicalize the entries in $\tilde{D}$ since the entries in $\hat{D}$ remain tight.

Note that the above lemma and theorem benefit from the fact that extrapolation only increases bounds and that canonicalization computes minimum and hence it only decreases bounds. Note also that the above results are valid for all extrapolation procedures mentioned above since $\text{Extra}_M(D) \subseteq \text{Extra}^+_M(D) \subseteq \text{Extra}_{LU}(D) \subseteq \text{Extra}^+_{LU}(D)$ [BBLR06].

However, there are some special cases where extrapolation does not break the canonical form of the matrix and hence there is no need to recanonicalize after extrapolation, as discussed in the following observation.

**Observation 6.3.3.** Suppose that extrapolation changes only the entries in the first column (i.e. the upper bounds of the clocks) and that all the entries in that column have been changed during extrapolation. Then there is no need to recanonicalize the matrix in this case. However, if extrapolation changes some other constraints in the matrix in addition to the entries in the first column then during canonicalization the entries in the first column need not to be recanonicalized since none of them will be reduced and therefore they should not be used to tighten the other constraints. Recall that when an entry in the first column is extrapolated it will be set to $(\infty, <)$ and when all the entries in that column are set to $(\infty, <)$ then none of them will be changed during canonicalization regardless of the weights of the other constraints in the matrix. This is due to the fact that all acyclic paths of constraints that can
be used to tighten the constraint \( (D_{i,0}, \prec_{i,0}) \) will pass through a constraint of the form \( (D_{j,0}, \prec_{j,0}) \), where \( j \neq i \), and since \( D_{j,0} \) is \( \infty \) then \( D_{i,0} \) will remain \( \infty \) after canonicalization and hence no need to recanonicalize \( (D_{i,0}, \prec_{i,0}) \).

As we can see from observation 6.3.3 knowing which constraints have been extrapolated and how many constrains have been extrapolated can sometimes affect the cost of fixing the non-tightness introduced by extrapolation. In fact, the cost of canonicalizing an extrapolated matrix may vary depending on the set of constraints in \( \tilde{D} \). A careful look at the way the extrapolation functions change the bounds in the matrix show that in certain circumstances the standard Floyd-Warshall algorithm may perform some unnecessary computations when canonicalizing an extrapolated matrix, as discussed in the above observation. This may lead to unnecessary overhead during the analysis of an automaton in particular when cycles (loops) exist.

We now show how one can come up with a canonicalization algorithm that runs in \( O(c_1 \times n) \) using Theorem 6.3.2, where \( c_1 \) represents the number of constraints that have been changed during extrapolation. First recall that Floyd-Warshall algorithm uses three nested loops, because tightening looks at all acyclic paths between two constraints, not just those of length 2. Floyd-Warshall achieves this by doing \( n \) iterations of comparing the direct edge to path of length 2, each time tightening at least one constraint.

**Theorem 6.3.4. (Effectiveness of partial canonicalization for M-operator and LU-operator).** Using Theorem 6.3.2 and observation 6.3.3 a specialization of Floyd's algorithm can be derived allowing re-canonicalization of M-extrapolated matrix or LU-extrapolated matrix in \( O(c_1 \times n) \) instead of \( O(n^3) \), where \( c_1 \) is bounded by \( n^2 \) (see Algorithm 9).

Note that in case that all the upper bounds of the clocks have been changed during extrapolation (i.e. all have the form \( (\infty, <) \)) then these constraints should not be considered during canonicalization and should not be used to tighten any constraint (see Algorithm 9). Note also that the complexity of the algorithm is not fixed since it
bool skipUpperBounds := false

if forall i = 1..n ((i, 0) ∈ \(\tilde{D}\)) then
  \(\tilde{D} := \tilde{D} \setminus \{(i, 0) | i = 1..n\};\) skipUpperBounds = true

if skipUpperBounds and |\(\tilde{D}\)| ≠ 0
{
  for k := 1 to n do
    for each (i, j) ∈ \(\tilde{D}\) do
      \(D_{i,j} := \min(D_{i,j}, D_{i,k} + D_{k,j})\)
    end
  end
}
else if ¬skipUpperBounds
{
  for k := 0 to n do
    for each (i, j) ∈ \(\tilde{D}\) do
      \(D_{i,j} := \min(D_{i,j}, D_{i,k} + D_{k,j})\)
    end
  end
}

Algo\(\text{rithm 9:}\) Specialisation of Floyd’s algorithm when tightening an extrapolated matrix

varies depending on the kind of the changes that have occurred during extrapolation and the size of the list \(\tilde{D}\). In fact, there are cases where the complexity or the cost of fixing the non-tightness introduced by extrapolation can be zero (i.e. no need to recanonicalize the matrix) though that the number of constraints that have been extrapolated is greater than one (see observation 6.3.3). There are cases also where the cost can be \(O(n)\). This happens when only a single constraint is extrapolated. However, the general complexity of the algorithm is \(O(c_1 \times n)\). It remains to mention that the complexity of the algorithm in the worst-case scenario (i.e. when all the constraints in the matrix are extrapolated) is in practice much smaller than \(O(n^3)\). From observation 6.3.3 we know that the constraints in the first column need not to be recanonicalized if all of them are extrapolated and they should not be used to canonicalize the other constraints since they can not tighten any constraint. Note that one can skip safely during canonicalization all the paths that can lead to ∞
as a total cost since such paths can not tighten any constraint.

We now turn to discuss a specialised canonicalization procedure for the extrapolation operators $\text{Extra}_M^+(D)$ and $\text{Extra}_{LU}^+(D)$ since these operators have some distinguishing properties in comparison with the standard $M$-extrapolation and $LU$-extrapolation procedures. In fact, the operators $\text{Extra}_M^+(D)$ and $\text{Extra}_{LU}^+(D)$ set a lot of constraints to $\infty$ during extrapolation. This can be advantageous when canonicalizing a matrix that is extrapolated using these operators. We first discuss some observations about the operators $\text{Extra}_M^+(D)$ and $\text{Extra}_{LU}^+(D)$ that can lead to more efficient canonicalization algorithm than this described in 9.

Observation 6.3.5. Note that the operators $\text{Extra}_M^+(D)$ and $\text{Extra}_{LU}^+(D)$ have some nice properties that can lead to surprising results during canonicalization. First, the operators $\text{Extra}_M^+(D)$ and $\text{Extra}_{LU}^+(D)$ set a lot of constraints to $\infty$ during extrapolation. If the upper bound of a clock $x_i$ has been removed during extrapolation then all the diagonal constraints of the form $(D_{i,j}, <_{i,j})$ will be removed as well. Second, if the lower bound of a clock $x_i$ or clock $x_j$ is above its corresponding extrapolation constant then all the diagonal constraints of the form $(D_{i,j}, <_{i,j})$ will be removed. From these properties we can see that if the entire matrix is extrapolated using these operators then the resulting matrix will be in canonical form as discussed in the following lemma.

Lemma 6.3.6. Let $D$ be a DBM in canonical form. Suppose that $D$ has been fully extrapolated (i.e. all the constraints in $D$ have been extrapolated) using the operators $\text{Extra}_M^+(D)$ or $\text{Extra}_{LU}^+(D)$. Then the resulting extrapolated matrix will be in canonical form and there is no need to recanonicalize the matrix.

Proof. To prove lemma 6.3.6 we need to show that none of the constraints in the extrapolated matrix can be reduced during canonicalization.

- (None of the upper bound of the clocks will be reduced during canonicalization). From observation 6.3.3 it is easy to see that this case is true since all the upper bounds of the clocks will have the form $(\infty, <)$. 

(None of the lower bound of the clocks will be reduced during canonicalization). Since we assume that the entire matrix has been extrapolated using the operators $\text{Extra}_M^+(D)$ or $\text{Extra}_{LU}^+(D)$ then we know that all the constraints in the matrix except the lower bound of the clocks (i.e. the first row in the matrix) will be of the form $(\infty, <)$ and by definition of the tightening algorithm [Rok93] we know that all acyclic paths that can be used to tighten a constraint of the form $(D_{0,j}, <_{0,j})$, for all $j = 1..n$, will pass through a constraint of the form $(\infty, <)$ and hence none of the lower bounds of the clocks can be reduced during canonicalization.

( None of the diagonal constrains will be reduced during canonicalization). Similarly, since we assume that the entire matrix has been extrapolated using the operators $\text{Extra}_M^+(D)$ and $\text{Extra}_{LU}^+(D)$ then we know that all the diagonal constraints in the matrix will be of the form $(\infty, <)$. By definition of the tightening algorithm [Rok93] we know that all acyclic paths that can be used to tighten the diagonal constraint $(D_{i,j}, <_{i,j})$ will pass through a constraint of the form $(\infty, <)$ and hence none of the diagonal constraints can be reduced during canonicalization.

The soundness of lemma 6.3.6 is due to the fact that when the entire matrix is extrapolated using the operators $\text{Extra}_M^+(D)$ and $\text{Extra}_{LU}^+(D)$ then all the diagonal constraints and all the constraints of the form $(D_{j,0}, <_{j,0})$ in the matrix will be of the form $(\infty, <)$. Unfortunately, this result has not been observed in [BBLR06]. However, a more careful analysis of the way the operators $\text{Extra}_M^+(D)$ or $\text{Extra}_{LU}^+(D)$ change the bounds in DBMs reveals some interesting observations about these operators that can be used to develop a highly efficient canonicalization algorithm for these cases.

Observation 6.3.7. From the definition of the operators $\text{Extra}_M^+(Z)$ or $\text{Extra}_{LU}^+(D)$ we know that when the entire zone $Z$ is above the maximum constant of some
clock for example \( x_i \), then all the diagonal constraints of the form \((D_{i,j}, \prec_{i,j})\) will be removed. That is, if \((D_{i,0}, \prec_{i,0})\) has been extrapolated then all the constraints in the \(i\)-row will be extrapolated as well. It is easy to see then if any of the upper bounds of the clocks has been extrapolated then there is no need to recanonicalize it since it can not be reduced during canonicalization. Note also that if the diagonal constraint \((D_{i,j}, \prec_{i,j})\) has been extrapolated then also all the constraints in the \(i\)-row have been extrapolated and they have the form \((\infty, <)\). This implies that none of the diagonal constraints that have been extrapolated can be reduced during canonicalization.

From the above analysis we can see that the only case in which the \(M^+\)-extrapolated matrix or the \(LU^+\)-extrapolated matrix may need to be recanonicalized is when some of the lower bounds of the clocks have been changed during extrapolation since they may be reduced during canonicalization.

**Theorem 6.3.8.** *(Effectiveness of partial canonicalization for \( \text{Extra}^+_M(D) \) and \( \text{Extra}^+_{LU}(D) \)).* Using Theorem 6.3.2 and observations 6.3.5 and 6.3.7 a specialization of Floyd’s algorithm (see Algorithm 10) can be derived allowing recanonicalization of \(M^+\)-extrapolated matrix or \(LU^+\)-extrapolated matrix in \(O(c_2 \times n)\) instead of \(O(n^3)\), where \(c_2\) represents the number of constraints of the form \((D_{0,j}, \prec_{0,j})\) that have been changed during extrapolation, which is bounded by \(n\).

```plaintext
if |\(\bar{D}\)| ≠ n
   { 
      for k := 0 to n do 
         for each \((0, j) \in \bar{D}\) do 
            \(D_{0,j} := \min(D_{0,j}, D_{0,k} + D_{k,j})\)
         end
      end
   }

Algorithm 10: Specialisation of Floyd’s algorithm when tightening an \(M^+\)-extrapolated matrix or an \(LU^+\)-extrapolated matrix
```

Note that the list \(\bar{D}\) used in Algorithm 10 contains only the constraints of the form
Extrapolation procedure | canonicalization complexity
---|---
The \( \text{Extra}_M(D) \) procedure | \( O(c_1 \times n) \), where \( c_1 \leq n^2 \)
The \( \text{Extra}_M^+(D) \) procedure | \( O(c_2 \times n) \), where \( c_2 \leq n \)
The \( \text{Extra}_{LU}(D) \) procedure | \( O(c_1 \times n) \), where \( c_1 \leq n^2 \)
The \( \text{Extra}_{LU}^+(D) \) procedure | \( O(c_2 \times n) \), where \( c_2 \leq n \)

Table 1: Complexity of canonicalization for the four different extrapolation procedures.

\((D_{0,j}, \prec_{0,j})\) that have been changed during extrapolation since the other extrapolated constraints can not be reduced during canonicalization as discussed in observation 6.3.7 and hence no need to consider them during canonicalization. Note also that when \( |\tilde{D}| = n \) then we know that all the lower and upper bounds of the clocks have been extrapolated and from the definition of \( \text{Extra}_M^+(D) \) and \( \text{Extra}_{LU}^+(D) \) we know that if the upper bound of all the clocks have been removed during extrapolation then all the diagonal constraints have been removed as well. From observation 6.3.5 we know that when the entire matrix has been extrapolated using \( \text{Extra}_M^+(D) \) or \( \text{Extra}_{LU}^+(D) \) then the resulting matrix will be in canonical form.

Note that the canonicalization algorithm 10 benefits from two observations: the first is that the operators \( \text{Extra}_M^+(D) \) and \( \text{Extra}_{LU}^+(D) \) set a lot of constraints to \( \infty \) during extrapolation and the second is that the constraints that are not changed during extrapolation remain tight and hence no need to tighten them (see Lemma 6.3.2). These two observations together lead to a highly efficient algorithm for canonicalizing \( M^+ \)-extrapolated matrices and \( LU^+ \)-extrapolated matrices. As we can see from the above analysis the cost of fixing the non-tightness introduced by extrapolation depends mainly on the applied extrapolation function. Since as we observe the cost of canonicalizing an \( M \)-extrapolated matrix or an \( LU \)-extrapolated matrix is \( O(c_1 \times n) \), where \( c_1 \) is bounded by \( n^2 \), while the cost of canonicalizing an \( M^+ \)-extrapolated matrix or an \( LU^+ \)-extrapolated matrix is \( O(c_2 \times n) \), where \( c_2 \) is bounded by \( n \).

Note that among the above four extrapolation procedures, the \( LU^+ \)-extrapolation yields the coarser abstraction of TA and it has the cheapest canonicalization cost.
6.4. CHECKING INCLUSION BETWEEN ZONES

<table>
<thead>
<tr>
<th>Operation</th>
<th>Canonicalization Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBM operation</td>
<td>canonicalization complexity</td>
</tr>
<tr>
<td>The intersection operation</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>The extrapolation operation</td>
<td>$O(c \times n)$, where $c \leq n$ (this chapter)</td>
</tr>
<tr>
<td>The delay and reset operations</td>
<td>No need for canonicalization [Yov98, BY04]</td>
</tr>
</tbody>
</table>

Table 2: Complexity of canonicalization for the elementary operations of the zone approach

Hence, the abstraction based on the $LU^+$-extrapolation will result in a smaller symbolic representation of the state space of TA in comparison with the other extrapolation procedures and it has the lowest canonicalization complexity which helps to reduce the overall analysis time of TA. For these reasons the $LU^+$-extrapolation procedure can be considered as the most efficient abstraction for diagonal-free TA.

6.4 Checking Inclusion between Zones

Another key operation in state space exploration is inclusion checking for the solution sets of two zones. However, before discussing a modification to the standard inclusion checking operation we review first the standard forward reachability algorithm for TA implemented in the tools UPPAAL and KRONOS. Reachability is a fundamental problem in verification. For timed automata, it is stated as follows: given a timed automaton $\mathcal{A}$ and a set of locations $L_f$, does there exist a run leading to some state $(l, Z)$ with $l \in L_f$? This problem has been proved decidable with a PSPACE-complete by Alur and Dill [AD94]. A formal description of the forward reachability algorithm of TA is given in Algorithm 11.

The algorithm uses two data structures WAIT and PASSED to store symbolic states waiting to be examined, and the states that were already examined, respectively. The WAIT set is instantiated with the initial symbolic state $(l_0, Z_0)$. Each node in the computed tree is of the form $(l_i, \text{Extra}_M(Z_i))$ where $l_i$ is a location in the automaton, $\text{Extra}_M(Z_i)$ is the corresponding extrapolated zone, and $M$ is an extrapolation constant. The algorithm terminates because there are finitely many
Input: \( \mathcal{A}: \text{TA}; l_f: \text{location}; M: \text{integer} \)

\[
\text{PASSED} := \emptyset, \text{WAIT} := \{(l_0, Z_0)\}
\]

while \( \text{WAIT} \neq \emptyset \) do 

\[
\text{select } (l, Z) \text{ from WAIT}
\]

if \( l = l_f \land Z \cap Z_f \neq \infty \) then return “YES”

add \((l, \text{Extra}_M(Z))\) to \(\text{PASSED}\)

for all \((l', \text{Extra}_M(Z'))\) such that \((l, \text{Extra}_M(Z)) \sim (l', \text{Extra}_M(Z'))\) do

if \(\text{Extra}_M(Z') \not\subseteq \text{Extra}_M(Z'')\) for all \((l', \text{Extra}_M(Z''))\) \(\in \text{PASSED}\) 

then add \((l', \text{Extra}_M(Z'))\) to \(\text{WAIT}\)

return “NO”

Algorithm 11: Forward reachability Analysis Algorithm for Timed Automata

bounded or extrapolated zones that can be computed for each location of the automaton [BY04]. More specifically, the use of extrapolation algorithms together with the zone inclusion test \(\text{Extra}_M(Z) \subseteq \text{Extra}_M(Z')\) is the key reason why termination is guaranteed in TA. Note that before adding a new state \((l, Z)\) to \(\text{WAIT}\) we check if \((l, Z) \subseteq (l, Z')\) for any state \((l, Z') \in \text{PASSED}\) since when \((l, Z) \subseteq (l, Z')\) then all states reachable from \((l, Z)\) are also reachable from \((l, Z')\), and thus we only need to explore \((l, Z')\). Note that the algorithm computes step by step an approximation of the set of reachable states and then checks if the computed extrapolated state intersects with the set of final state. If a location \(l_f\) is found reachable, the algorithm returns “yes”, otherwise, it returns “no”.

However, as discussed in Section 6.2 checking inclusion between DBMs using canonicalized extrapolated version of DBMs is an unsafe operation since termination using this form of test may not be guaranteed. We now show that checking inclusion between DBMs using un-canonicalized extrapolated DBMs is safe and it guarantees the correct termination of the analysis. Note that the problematic step is to reduce the extrapolated DBMs to canonical form before checking inclusion between DBMs, since canonicalizing extrapolated DBMs may yield non-extrapolated DBMs. This can affect adversely termination of the analysis of TA in particular when cycles exist. To explain why the test of inclusion using un-canonicalized extrapolated DBMs is safe we need some preliminary observations. From lemma 6.3.2 we know that all
the constraints that are not changed during extrapolation preserve their canonical
form and they are $M$-bounded constraints or $LU$-bounded constraints depending
on which extrapolation technique is used. Note also that whenever a particular
constraint $(D_{i,j}, \prec_{i,j})$ in a DBM $D$ is extrapolated (enlarged) it will always have
the same form. For example, if the $M$-extrapolation procedure is applied then
whenever the lower bounds of the clocks are extrapolated they will have the form
$(-M, <)$ and whenever the upper bounds of the clocks are extrapolated they will
have the form $(\infty, <)$. Moreover, as discussed before, if an extrapolated constraint
is reduced during canonicalization then it can not be smaller than its correspond-
ing extrapolation constant. Furthermore, as we know from [DT98, BY04] when a
clock in an automaton is never compared in a guard or invariant to a value greater
than its corresponding extrapolation constant, then the value of the clock will have
no impact on the computation of the automaton once it exceeds that constant. It
is easy to see that checking inclusion between DBMs can be performed correctly
without reducing them to canonical form. This is what we prove in Lemma 6.4.1.

**Lemma 6.4.1.** Checking inclusion between extrapolated DBMs directly without
reducing them to canonical form is safe and it guarantees correct termination.

**Proof.** First recall that in order to conclude that $D' \subseteq D$ we need to check that
for all $i,j$, such that $i \neq j$, $D'_{i,j} \leq D_{i,j}$. However, since inclusion checking is an
entry-wise operation (i.e. entry-wise comparison of two matrices) one can pick any
arbitrary entry to prove the lemma. Hence, there are several cases to consider.

1. (Case 1). If the entries $(D'_{i,j}, \prec'_{i,j})$ and $(D_{i,j}, \prec_{i,j})$ have not been touched
during extrapolation of $D$ and $D'$. This case is trivial. From lemma 6.3.2 we
know that $(D'_{i,j}, \prec'_{i,j})$ is already the tightest for $D'$ and $(D_{i,j}, \prec_{i,j})$ is already
the tightest for $D$ since they have not been touched during extrapolation and
hence checking whether $D'_{i,j} \leq D_{i,j}$ will be performed correctly.

2. (Case 2). If the constraint $(D'_{i,j}, \prec'_{i,j})$ has been changed during extrapolation
of \( D' \) while the constraint \((D_{i,j}, \prec_{i,j})\) has not been touched during extrapolation of \( D \). This case is also trivial. Note that since \((D_{i,j}, \prec_{i,j})\) has not been touched during extrapolation then the constraint \((D_{i,j}, \prec_{i,j})\) is already the tightest for \( D \) and its value is less than its corresponding extrapolation constant. On the other hand, since \((D'_{i,j}, \prec'_{i,j})\) has been extrapolated then we know that its value before extrapolation was greater than its corresponding extrapolation constant. Now since extrapolation only increases bounds then the tightest form of \((D'_{i,j}, \prec'_{i,j})\) can not be smaller than what it was before extrapolation (i.e. the tightest value of \((D'_{i,j}, \prec'_{i,j})\) will be greater than its corresponding extrapolation constant) and given that both \((D'_{i,j}, \prec'_{i,j})\) and \((D_{i,j}, \prec_{i,j})\) have the same corresponding extrapolation constant then the test \( D'_{i,j} \leq D_{i,j} \) can be performed correctly without reducing \((D'_{i,j}, \prec'_{i,j})\) to canonical form since computing the tightest form of \((D'_{i,j}, \prec'_{i,j})\) does not change the comparison result.

3. (Case 3). If both constraints have been changed during extrapolation. This case is interesting since both constraints have been enlarged during extrapolation. Note that since both constraints have been extrapolated then we know that their values before extrapolation were greater than their corresponding extrapolation constant. Now given that extrapolation only increases bounds then we know that the tightest form of \((D_{i,j}, \prec_{i,j})\) and \((D'_{i,j}, \prec'_{i,j})\) can not be smaller than what they were before extrapolation. From [DT98, BY04] we know that if a clock exceeds the maximum constant that is compared to in the guards of an automaton then it has no effect on the computation of the automaton and given that both \((D'_{i,j}, \prec'_{i,j})\) and \((D_{i,j}, \prec_{i,j})\) have the same corresponding extrapolation constant then knowing the exact (tightest) value of these constraints are not important for the correctness of the comparison. That is, if both extrapolated constraints have the form \((\infty, <)\) or \((-M, <)\) then we can safely conclude that \((D'_{i,j}, \prec'_{i,j})\) and \((D_{i,j}, \prec_{i,j})\) are equivalent.
even that their tightest values are not equal. This will not harm the analysis since their tightest values will not be smaller than their corresponding extrapolation constant.

However, checking inclusion between DBMs using an un-canonicalized extrapolated form requires some modifications to the standard reachability algorithms of TA. Since in order to ensure termination the list PASSED must maintain the explored states without reducing their extrapolated zones (DBMs) to canonical form. On the other hand, the WAIT list must maintain the states while ensuring that the zones of the states are in canonical form. This is necessary for computing efficiently the successors of the states in WAIT.

Note that the above lemma can lead to a considerable saving during the analysis of TA in particular when cycles (loops) exist since it states that the zones of the states maintained in the list PASSED need not to be in canonical form. This helps to reduce the cost of model checking real-time systems since it minimizes the number of times canonical forms are explicitly constructed. However, it would be advantageous to have an extrapolation procedure that satisfies the property that whenever an extrapolated zone (matrix) is reduced to canonical form it preserves its extrapolated form. As mentioned before, the well-known $M$-extrapolation and $LU$-extrapolation procedures do not satisfy this property and hence it is necessary to perform inclusion checking between zones using un-canonicalized extrapolated zones. It would be more advantageous if we can have an extrapolation procedure that gives extrapolated zones in canonical form so that no need to recanonicalize the matrix after extrapolation. This might not be easy to achieve.
6.5 Implementation and Experiment

In this section we briefly summarise our prototype implementation of the model checking algorithms given in Section 6.3. The prototype implementation has been developed using the opaal tool [DHJ+11] which has been designed to rapidly prototype new model checking algorithms. The opaal tool is implemented in Python and is a standalone model checking engine. Models are specified using the UPPAAL XML format. We use the open source UPPAAL DBM library for the internal symbolic representation of time zones in the algorithms.

Note that the proposed partial canonicalization algorithms are aimed to reduce the verification time that is needed to analyse an automaton using the zone approach. However, since improvement only affects the successor computation at step 10 of the operations described in Section 2.9, and in particular only after an extrapolation operation, it will not improve the reachability analysis of every TA. Thus, the proposed improvement is best suited for cyclic real-time systems or for systems where extrapolation is necessary for the termination of the analysis. We made several experiments on some interesting examples and applications of TA including the Fischer’s protocol and the Philips audio control protocol (see below for details). The experiments have been performed first without using optimisations and then with optimisations. The results show that the improved canonicalization operators give a verification speed-up of order of at least \((25 - 30)\%\) compared to the classical canonicalization procedure in particular for the Fischer’s protocol and the Philips audio control protocol. The models and the formal properties of the protocols have been taken from the distributed installation package of the tool UPPAAL [BDL04].

6.5.1 The Fischer’s Protocol

The Fischer’s protocol [Lam87] ensures mutual exclusion among \(N\) processes using real-time clocks and a shared variable ‘lock’. We verified safety and liveness properties of the protocol. We verified the protocol using the \(LU^+\)-extrapolation
### 6.5. IMPLEMENTATION AND EXPERIMENT

<table>
<thead>
<tr>
<th>System</th>
<th>Property</th>
<th>without optimisations</th>
<th>with optimisations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fischer</td>
<td>Safety</td>
<td>0.47</td>
<td>0.29</td>
</tr>
<tr>
<td>(3 process)</td>
<td></td>
<td>0.55</td>
<td>0.34</td>
</tr>
<tr>
<td>Fischer</td>
<td>liveness</td>
<td>3.27</td>
<td>1.21</td>
</tr>
<tr>
<td>(4 process)</td>
<td></td>
<td>5.27</td>
<td>3.11</td>
</tr>
<tr>
<td>Fischer</td>
<td>Safety</td>
<td>39.5</td>
<td>12.5</td>
</tr>
<tr>
<td>(5 process)</td>
<td></td>
<td>46.5</td>
<td>24.3</td>
</tr>
<tr>
<td>Fischer</td>
<td>liveness</td>
<td>307.9</td>
<td>207.7</td>
</tr>
<tr>
<td>(6 process)</td>
<td></td>
<td>447</td>
<td>335.2</td>
</tr>
</tbody>
</table>

**Table 1**: Comparative performance of difference canonicalization procedures for the Fischer’s protocol

procedure (the most general extrapolation procedure) first without optimisations and then with the optimisations turned on. Table 1 presents the verification time (in seconds) for verifying several properties of the Fischer’s mutual exclusion protocol. The measurements have been done on Sun Solaris OS with 6GB of memory. Recall that computing canonical forms dominates the cost of model checking real-time systems and avoiding them or reducing their costs wherever possible will accrue gains in performance. Hence when verifying the safety formula of the Fischer’s protocol with 3-6 processors it takes 0.29, 2.21, 12.5, and 207.7s respectively when the optimisations are enabled, while it takes 0.47, 3.27, 39.5, and 307 respectively when the optimisations are disabled. On the other hand, when verifying the liveness properties of the protocol with 3-6 processors it takes 0.34, 3.11, 14.3, and 335.2s respectively when the optimisations are enabled, while it takes 0.55, 5.27, 46.5, and 547 respectively when the optimisations are disabled. The verification results of the Fischer’s protocol show that the analysis can be considerably faster using the new optimised canonicalization procedures.

#### 6.5.2 The Philips Audio Control Protocol

The Philips audio control protocol is a real-life protocol that is widely reported in the literature [BGK+96]. The protocol specifies communication between devices over an interface bus without a central controller. We verified two properties of
CHAPTER 6. PARTIAL CANONICALIZATION OF CLOCK ZONES

<table>
<thead>
<tr>
<th>System</th>
<th>Property</th>
<th>without optimisations</th>
<th>with optimisations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Philips</td>
<td>Property 1</td>
<td>1.27</td>
<td>0.37</td>
</tr>
<tr>
<td>(3 process)</td>
<td>Property 2</td>
<td>2.25</td>
<td>0.57</td>
</tr>
<tr>
<td>Philips</td>
<td>Property 1</td>
<td>4.67</td>
<td>2.51</td>
</tr>
<tr>
<td>(4 process)</td>
<td>Property 2</td>
<td>7.26</td>
<td>4.15</td>
</tr>
<tr>
<td>Philips</td>
<td>Property 1</td>
<td>75.2</td>
<td>35.5</td>
</tr>
<tr>
<td>(5 process)</td>
<td>Property 2</td>
<td>85.4</td>
<td>52.6</td>
</tr>
<tr>
<td>Philips</td>
<td>Property 1</td>
<td>451.5</td>
<td>305.8</td>
</tr>
<tr>
<td>(6 process)</td>
<td>Property 2</td>
<td>953</td>
<td>536</td>
</tr>
</tbody>
</table>

Table 2: Comparative performance of difference canonicalization procedures for the Philips audio control protocol

the protocol using our proposed optimised canonicalization procedures. The first property states that the receiver never enters error state, and the second is that the receiver receives entire message. The proposed optimisations lead also to considerable improvement in the verification times of the Philips protocol which is relatively a complex protocol (see the model of the protocol in the distributed package of UP-PAAL). The verification results reported in Table 2 show that the analysis of the Philips audio control protocol can be considerably faster using the new optimised canonicalization procedures. Note that the optimisation is effective whenever there are multiple (infinite) cycles or loops in the transition system. Since in such cases extrapolation is needed in order to guarantee termination.

6.6 Conclusion

In this chapter we proposed a set of optimisations for improving the reachability analysis and the WCET analysis of Timed Automata (TA) using the Difference Bound Matrices (DBMs), namely the partial canonicalization of DBMs. The partial canonicalization allows one to fix the non-tightness introduced by extrapolation by updating only the clock constraints that have been changed during extrapolation, thus reducing the impact on the run time of the reachability algorithm of the canonicalization step. The proposed partial canonicalization algorithms are specializations of Floyd’s algorithm that have a time complexity of $O(c \times n)$ instead of
$O(n^3)$ for the standard approach, where $n$ is the number of clocks in the automaton and $c$ is the number of variables that have been changed during extrapolation. We also reviewed the four well-known extrapolation procedures: the $M$-extrapolation procedure, the $LU$-extrapolation procedure, the $M^+$-extrapolation procedure, and the $LU^+$-extrapolation procedure, and showed that among these procedures, the $LU^+$-extrapolation procedure provides the coarser abstraction with the lowest cost of recanonicalization. We then demonstrated that model checking TA with partial canonicalization based on the $LU^+$-extrapolation procedure can speed-up considerably the verification time of several interesting examples including the Philips audio control protocol and Fischer’s protocol.
Chapter 7

Accelerating WCET Analysis of TA Models

Extrapolation abstraction is a powerful established technique in model checking timed automata (TA) that can be used to avoid divergence arising in the reachability analysis in a satisfactory way. Extrapolation can be viewed as an acceleration technique in the sense that it helps to obtain answers quickly for the reachability questions in TA even in the presence of infinite runs. The chapter presents a new efficient algorithm for computing worst case execution time (WCET) of systems modelled as timed automata (TA). The algorithm uses a set of acceleration and abstraction techniques that improve significantly the efficiency of WCET analysis of TA models with cyclic behaviour. We show that the proposed accelerations are exact with respect to the WCET problem in the sense that the WCET computed in the accelerated model is equal to the one computed in the non-accelerated model. We compare also our algorithm with the one implemented in the model checker UPPAAL which shows that when infinite cycles (i.e. cycles that can be run infinitely often) exist, UPPAAL’s algorithm may not terminate, and when largely repetitive finite cycles (i.e. cycles that can be run a finite number of times) exist, UPPAAL’s algorithm suffers from the state space explosion, thus leading to a low efficiency or resource exhaustion.
7.1 Introduction

In this chapter, we reconsider the problem of computing the “worst case execution time” (WCET) in timed automata. Given a timed automaton $A$ with a start location $l_s$ and a final location $l_f$, this problem asks to compute an upper bound on the time needed to reach the final location $l_f$ from the start location $l_s$. The problem is easy to solve in the case of acyclic TA [BRF14], but cycles might introduce unbounded WCET, that needs to be detected on-the-fly during the analysis. In general, WCET analysis is undecidable: it is undecidable to determine whether or not an execution of a system will eventually halt. However, for TA models one can use model-checking techniques to analyse the system and compute the WCET.

Relations of abstraction are valuable tools in coping with state explosion [CGL94]. In this approach we relate a structurally large complex or even infinite model to a simple abstract finite model by means of a relation that is known to preserve the properties that we wish to verify in the concrete model. Abstraction relations can be classified into three categories based on the amount of information that is preserved during the abstraction of the concrete model [CGP01]: (1) under-abstraction techniques where the abstract model contains less information than the concrete model, (2) over-abstraction techniques where the abstract model contains more information than the concrete model, and (3) exact abstraction techniques where no loss of information can happen during abstraction. However, for the WCET problem it is necessary to ensure that the applied abstraction do not affect adversely the correctness of the WCET computation. More precisely, the information that is preserved in the abstract model must be sufficient to compute the correct WCET of systems. Let $M$ be a timed model and $M^{abst}$ be the corresponding abstract model that results from applying the abstraction $abst$ on the model $M$. We then say that the abstraction $abst$ is exact w.r.t. the WCET problem in the sense that it does not introduce coarse over-approximation or coarse under-approximation on the value of WCET if $WCET(M) = WCET(M^{abst})$. That is, if the WCET of the model $M$ is equal to $T$ time units, where $0 \leq T \leq \infty$, then the WCET of the abstract model
\( M^{\text{abst}} \) is also equal to \( T \) time units (i.e. \( WCET(M) = WCET(M^{\text{abst}}) = T \)). Model checkers can be expected to run more efficiently on the abstract model than on the concrete model, and abstraction is often used to bring the verification problem within the bound of feasibility for model checking.

Our contribution in this chapter is to establish the correctness of a set of abstraction and acceleration techniques with respect to the WCET problem. In Chapter 5, we proposed an algorithm for computing WCET for a general class of diagonal-free TA. To avoid loss of precision in WCET calculations, the algorithm uses partial extrapolation of zones rather than full extrapolation and checks for exact fixed points (zones) rather than inclusion between zones in particular when computing zones inside reachable cycles. The problem of computing a fixed point of infinite cycles in a zone graph arises in the study of the WCET problem. It is easy to see that when the search reaches an exact fixed point (i.e. encounters two identical states) during the analysis of a cycle then the cycle is an infinite cycle. The existence of a fixed point of an infinite cycle is guaranteed by extrapolation (abstraction) and the fixed point is generally obtained by a monotone iteration of the extrapolation operator \( \text{Extra}_M(Z) \) [DT98] or \( \text{Extra}_{LU}(Z) \) [BBLR06], where \( Z \) is a zone to be extrapolated. However, the WCET analysis of systems can be sometimes very slow in particular when the system under analysis contains cycles (loops) that can be iterated a large number of times. In fact, verifying WCET of systems with cyclic behaviour can cause the state explosion problem if we traverse all the cycles iterations during the analysis in particular when there is a largely repetitive finite cycle in the behaviour of the analysed system.

To improve the efficiency of WCET analysis of systems with cyclic behaviour we propose to use a combination of four abstractions, namely: (1) extrapolation abstraction [DT98], (2) activity abstraction [DT98], (3) fixed point abstraction, and (4) partial fixed point abstraction. This combination of abstractions help to guarantee the efficient termination of the WCET analysis without adversely affecting the WCET computations. The proposed algorithm guarantees termination and
7.2. ABSTRACTIONS FOR TA

computes precisely the WCET and can detect the cases where the WCET can be infinity. Thus, the provided solution can be a significant break-through in computing WCET. Finally, we report on a prototype implementation using the model checker Opaal \[DHJ+11\] and evaluate the algorithm on several toy examples. We compare also our prototype against the model checker UPPAAL, which shows that the implementation in UPPAAL may not terminate on models with infinite cycles containing unbounded clocks (i.e. WCET is unbounded but it is not obvious in the model).

7.2 Abstractions for TA

In this section we discuss some abstraction techniques for TA and show how each of these abstractions can be used during the WCET analysis.

7.2.1 The Extrapolation Abstraction

As discussed in Chapter 5, the standard approaches for analysing a timed automaton that depend on computing the zone graph of the automaton while extrapolating zones at each step of the successor computation, will give abstract zones and hence result in abstract values of the execution times. Therefore, the classical abstraction used for verification of reachability problem is not correct for the WCET problem. The main difficulty is then to define an abstraction of zones that guarantees termination of the algorithm, while keeping information precise for the extra clock \( \delta \) that is used to measure the execution times of the automaton. We therefore proposed what we call partial extrapolation of zones. By *partial extrapolation* we mean the process of extrapolating a subset of the clock constraints in the zones in the sense that we allow some constraints to exceed the bound \( M \) without being extrapolated. In this case the zone may contain both extrapolated and non-extrapolated constraints and hence the zone can be considered as a partially extrapolated zone. We refer
the reader to Chapter 5 for more details and examples of the partial extrapolation process.

7.2.2 The Activity Abstraction

The activity abstraction allows us to eliminate or ignore clocks that are inactive at some point during the exploration. We now discuss when a clock in an automaton $\mathcal{A}$ can be considered as an inactive clock with respect to a cycle $\pi$ in $\mathcal{A}$.

**Definition 7.2.1.** Let $\mathcal{A}$ be a timed automaton with a set of clocks $\{x_1, \ldots, x_n\}$. Suppose that $\mathcal{A}$ contains a cycle (loop) $\pi$ in its behaviour. We say that a clock $x_i$ is inactive with respect to $\pi$ if $x_i$ does not appear in any of the invariants or guards of $\pi$. However, if $x_i$ is reset at some edge in $\pi$ but never gets tested inside $\pi$ then $x_i$ can be considered also as an inactive clock with respect to $\pi$.

In fact, the activity abstraction can help sometimes in accelerating the WCET analysis. We give now an example of a timed automaton by which we demonstrate how the activity abstraction can help in accelerating the WCET analysis. Consider the automaton in Figure 17 in which we have a diamond followed by an infinite cycle followed by a final or terminal location. The automaton has three clocks $x, y$ and $z$. The extrapolation constants of the clocks are as follows $M_x = 10000$, $M_y = 1000$, and $M_z = 1$. Let us denote the set of active clocks in the cycle as $\text{act}$ and the set of inactive clocks as $\text{inact}$. For this example the set $\text{act}$ is $\{z\}$ and the set $\text{inact}$ is $\{x, y\}$. Note that if we use the extrapolation abstraction only without the activity abstraction then the convergence to a fixed point of the cycle will be very slow since the extrapolation constants $M_x$ and $M_y$ are large and the clocks $x$ and $y$ are not active in the cycle. On the other hand, if we use the extrapolation abstraction together with the activity abstraction the search will reach a fixed point of the cycle within two iterations since the fixed point of the cycle will be checked with respect only to the clock $z$. That is, after two iterations we get $(Z_{L_0}^1 \setminus \text{inact}) = (Z_{L_0}^2 \setminus \text{inact})$ since the clock $z$ is reset at the last edge of the cycle, where $Z_{L_0}^1$ represents the
7.2. ABSTRACTIONS FOR TA

Figure 17: An example by which we demonstrate the usefulness of the activity abstraction

clock zone at location L0 at iteration one. Note that this is sufficient to conclude that the cycle is an infinite cycle.

In general, the active clocks inside a cycle can be classified into three categories: (a) active clocks that contribute only to the delay of the cycle, (b) active clocks that contribute only to the termination of the cycle, and (c) active clocks that contribute to both the delay and the termination of the cycle. As we will see later such classification of the active clocks of a cycle can help sometimes in accelerating the computations of WCET of cycles in particular for finite cycles that have constant delays.

7.2.3 The Fixed Point Abstraction

The fixed point abstraction considered in this chapter is defined in terms of a state \((l, Z)\), where \(l\) is a control location in the TA graph and \(Z\) is an \(n \times n\) matrix and that each entry \(z \in Z\) has the form \((d_{j,k}, \preceq_{j,k})\) and that \(d_{j,k}\) can take any arbitrary integer number or \(\pm \infty\). Suppose we have a cycle \(\pi\) that can be repeated infinitely often. We can then describe its behaviour using a recursive function \(f_\pi(Z^n_s) = \text{succ}(Z^n_s, E_\pi) = Z^{n+1}_s\) for \(n = 0, 1, 2, \ldots\), where \(Z^n_s\) is the corresponding zone at the start location of the cycle \(\pi\) at iteration \(n\). So, if there are two states \(s_1 = (l, Z)\) and \(s_2 = (l, Z')\) in the state space of a cycle \(\pi\) such that \(Z = Z'\) then we say that the state \((l, Z)\) is a fixed point of the cycle and the cycle is an infinite cycle. Note that it is guaranteed to reach a fixed point of \(\pi\) after a finite number of iterations.
using some extrapolation operator \( \text{Extra}(Z) \). We wish then to use an abstraction or a combination of abstractions to find as fast as possible a fixed point of \( \pi \). Fixed point problems of infinite cycles arise during the analysis of the WCET problem (see Chapter 5). However, since we are going to use extrapolation techniques to enforce the convergence of fixed point computations of infinite cycles the above function can be rewritten as \( f^\text{Extra}_\pi(Z^n_s) = \text{Extra}_M(\text{succ}(Z^n_s), E_\pi) = \text{Extra}_M(Z^n_{s+1}) \), where each entry \( z \in \text{Extra}(Z) \) can take values from the finite domain \([-M(A), M(A)]\) or the value \( \infty \), and \( M(A) \) is the maximum constant appears in the invariants or guards of the automaton of the cycle \( \pi \). We note that the use of abstraction is essential in order to guarantee the convergence of fixed point computations since the domain of the entries in the extrapolated matrix is finite.

**Definition 7.2.2. (Fixed Point Abstraction)**. Let \( A = (\Sigma, L, L_0, L_F, X, I, E) \) be a timed automaton and let \( E_\pi = (e_0, \ldots, e_{n-1}) \) be the sequence of edges of a cycle \( \pi \) in \( A \). Suppose that the function \( \text{succ}(Z, E_\pi) \) computes the successor of the zone \( Z \) w.r.t sequence of edges \( E_\pi \), which is equivalent to executing the cycle \( \pi \) one full iteration. We say that \( Z \) is a fixed point of \( \pi \) and \( \pi \) is an infinite cycle if \( \text{succ}(Z, E_\pi) = Z \).

Fixed point abstraction is a powerful abstraction that can be used safely inside cycles when computing WCET. To demonstrate the power of fixed point abstraction in detecting infinite behaviour in TA we present here some particularly tricky examples of TA. In Figure 18 we give an example of an automaton that contains three finite cycles that have the start location \( \text{start} \) as a common location. It is interesting to note that there are some dependencies between the behaviour of the three cycles. However, as one can see, the three cycles collectively will be executed infinitely often which lead to a WCET of infinity. Note that the upper bound on the time needed to reach any of the locations of that automaton is infinity. Note also that after a finite number of iterations the search will reach a fixed point at the location \( \text{start} \) and using the extra clock one can conclude that the WCET of the automaton is infinity. On the other hand, the automaton in Figure 19 represents an
7.2. ABSTRACTIONS FOR TA

Figure 18: Three finite dependent cycles that lead to infinite WCET

Figure 19: Three self-cycles that lead to WCET of 20

automaton with four finite independent self-cycles where WCET of the automaton is 20. For this automaton the search will not reach a fixed point and the standard forward analysis technique will terminate while computing correctly the WCET of the automaton. Note that the start and the final location of that automaton is the same.

The convergence to a fixed point of an infinite cycle $\pi$ (i.e. $\text{succ}(Z_s^{j+1}, E_{\pi}) = \text{succ}(Z_s^j, E_{\pi})$) can be intuitively understood due to the finite number of extrapolated zones that can be obtained at each location of the cycle. However, the fixed point computations of infinite cycles can be significantly improved if we apply both the extrapolation abstraction and the activity abstraction. Note that the number of
iterations needed to reach a fixed point of an infinite cycle $\pi$ is bounded by the number of distinct zones that can be obtained at the start location of the cycle. Note that since the starting zone of the cycle at the first iteration can be an arbitrary zone then the bound on the number of iterations needed to reach a fixed point of $\pi$ will be $|Z_s| + 1$, where $|Z_s|$ is the number of distinct zones that can be obtained at the start location of the cycle. Note that $|Z_s|$ is finite due to the application of the extrapolation procedures.

We discuss now two special cases of infinite cycles where the search can reach a fixed point within two iterations. We formalize these two cases in the following lemmas.

**Lemma 7.2.3.** (Fixed points of infinite non-realizable cycles). Let $E_\pi = (e_0, \ldots, e_{m-1})$ be the sequence of edges of an infinite non-realizable cycle $\pi$ in an automaton $A$. Then a fixed point of $\pi$ can be reached within two iterations.

**Proof.** Let $\text{succ}(Z, E_\pi)$ be a function that computes the successor of $Z$ after executing the sequence of edges in $\pi$ which is equivalent to executing the cycle one full iteration. Suppose that the cycle $\pi$ starts its first iteration with the zone $Z$ in which the clock valuation $v(x_i) = c_i$ where $x_i \in X$ and $c_i \in \mathbb{R}_{\geq 0, \infty}$. Recall that zones provide a representation of sets of clock interpretations. We need to show that after two iterations of $\pi$ the search can reach a fixed point. That is, $\text{succ}(\text{succ}(Z, E_\pi), E_\pi) = \text{succ}(Z, E_\pi)$. From the assumption that $\pi$ is a non-realizable cycle we know that time can not elapse at any iteration of the cycle. Hence, $\text{delay}(\pi) = \sum_{i=0}^{m-1} d_i = \triangleright 0, 0 \triangleleft$, where $d_i$ is the delay interval spent at location $\text{src}(e_i)$ and therefore none of the automaton clocks will be advanced. So after the first iteration we have either $x_i = c_i$ or $x_i = 0$ in case $x_i$ is reset inside $\pi$. It is easy to see then that $\text{succ}(\text{succ}(Z, E_\pi), E_\pi) = \text{succ}(Z, E_\pi)$ since at each iteration of $\pi$ we have $\text{delay}(\pi) = \triangleright 0, 0 \triangleleft$ and that none of the clocks in the zone $\text{succ}(Z, E_\pi)$ will be changed during the second iteration. That is, if the clock $x_i$ is reset inside $\pi$ then its value at the end of the second iteration will be 0 which is the same value it has at the end of the first iteration and if it is not reset it will be $c_i$ which is the
value it has before executing the cycle. Therefore, the search can indeed reach a fixed point of $\pi$ after two iterations.

**Lemma 7.2.4.** (Fixed points of infinite self-cycles). Let $\pi$ be an infinite self-cycle in a timed automation $A$. Then using the extrapolation abstraction together with the activity abstraction a fixed point of $\pi$ can be reached within two iterations.

**Proof.** Since $\pi$ is a self-cycle then $E_\pi = \{e_0\}$ and $src(e_0) = trg(e_0) = l$ so there is only a single edge $e_0$ and a single location $l$ in $\pi$. Suppose that the cycle $\pi$ starts its first iteration with the zone $Z$ in which $v(x_i) = c_i$ where $x_i \in X$ and $c_i \in ]0, \infty[$. Suppose further that $inact \in X$ represents the set of clocks of the automaton that are inactive in $\pi$. We need to show that after two iterations of $\pi$ the search can reach a fixed point w.r.t active clocks in $\pi$. That is, $(succ(succ(Z, E_\pi), E_\pi) \setminus inact) = (succ(Z, E_\pi) \setminus inact)$. Now given that $\pi$ is an infinite realizable self-cycle we know that every active clock $x_i$ in $\pi$ will be reset at edge $e_0$, otherwise $x_i$ will become a blocking clock or inactive clock at some iteration. Note that at the end of the first iteration we have $v(x_i) = 0$ for all $x_i \in (X \setminus inact)$ due to the reset operation. Similarly, at the end of the second iteration we have $v(x_i) = 0$ for all $x_i \in (X \setminus inact)$. From the semantic definition of the zone approach it is easy to see that $(succ(succ(Z, E_\pi), E_\pi) \setminus inact) = (succ(Z, E_\pi) \setminus inact)$ since for all $x_i \in (X \setminus inact)$ we have $v(x_i) = 0$ in both the zone $succ(Z, E_\pi)$ and the zone $succ(succ(Z, E_\pi), E_\pi)$ due to the reset operation. Therefore, the search can reach a fixed point of $\pi$ w.r.t. active clocks of $\pi$ after two iterations.
7.3 Accelerating WCET Calculations of Finite Cycles

In the above sections we discussed how one can use a set of abstraction techniques to handle efficiently infinite cycles in TA regardless of their topological structure or timing behaviour. The second important issue to consider when accelerating WCET analysis is finite cycles with large number of iterations. Note that when verifying WCET of TA with cyclic behaviour it would cause the state explosion problem if we traverse all the cycle’s iterations during the analysis in particular when there is a largely repetitive finite cycle in the behaviour of the analysed automaton. The question is then how one can accelerate the computations of such cycles?

Note that for finite cycles the classical known abstraction techniques can not be used to accelerate the computations of WCET as we have done for the infinite cycles and hence new acceleration techniques need to be developed for finite cycles. This is mainly because every new execution of the cycle gives rise to new symbolic states and in case there is a clock that is bounded from above (i.e. the clock has the form $x \sim c$ where $\sim \in \{<, \leq\}$) and that the clock is not reset inside the cycle and that the bound that the clock is compared to is large then the extrapolation, inclusion, and fixed point abstractions are not useful in this case. See for example the automaton in Figure 20 where the extrapolation, inclusion, and fixed point abstractions can not be used to accelerate the WCET analysis.

However, in this work we limit our attention to two forms of finite cycles: finite cycles in which there is one clock that determines the delay of the cycle and one clock determining the termination of the cycle, and finite cycles in which there are more than one clock determine the delay of the cycle while only one clock determines the termination of the cycle. So our proposal for accelerating the computations of WCET of finite cycles is to classify (when possible) the active clocks in the cycle into active clocks for determining delay and active clocks for determining termination. We show that this way of classification is sometimes
7.3. ACCELERATING WCET CALCULATIONS OF FINITE CYCLES

Figure 20: A largely repetitive finite cycle

possible and feasible and can be used to fasten the state space exploration of largely repetitive finite cycles.

7.3.1 Finite Cycles with Simple Structure

The majority of the cycles in real-time systems, embedded systems, and control systems have a well-defined and simple structure [HL02]. In this section we discuss a class of finite cycles which we call finite cycles with constant delays, where we assume that there is one clock determining the delay of the cycle, let us denote it $x_1$, and another clock determining the termination of the cycle, let us denote it $x_2$.

We give now a definition of a simple syntactic structure of finite cycles where the WCET computations can be accelerated on-the-fly during the analysis.

Definition 7.3.1. Let $\mathcal{A} = (\Sigma, L, L_0, L_F, X, I, E)$ be a timed automaton, let $E_\pi = (e_0, ..., e_{n-1})$ be the sequence of edges of a cycle $\pi$ and let $X_\pi = \{x_1, x_2\} \in X$. Suppose that we have two functions $\text{in}(l_i)$ and $\text{out}(l_i)$ that return respectively the set of incoming edges and outgoing edges of location $l_i$. We say that the tuple $(E_\pi, X_\pi)$ is a WCET acceleratable finite cycle, if

- $I(l)$ has the form $x_1 \sim c$ where $\sim \in \{<, \leq\}$ for all $l \in L_\pi$,
- for all $(l, l', a, \lambda, \phi) \in E_\pi$ then $\phi$ has the form $x_1 \sim_1 c_1$ where $\sim_1 \in \{=, >, \geq\}$ or
$x_2 \sim_2 c_2$ where $\sim_2 \in \{<, \leq\}$ or $x_1 \sim_1 c_1 \land z \sim_2 c_2$ and $\lambda$ is either empty or contains only $x_1$,

- there exists an edge $e = (l, l', a, \lambda, \phi) \in E_\pi$ such that $\phi$ has the form $x_2 \sim_2 c_2$ or $x_1 \sim_1 c_1 \land x_2 \sim_2 c_2$,

- for all $l_i \in L_\pi$ and for all $e \in \text{out}(l_i)$ we have either $e \in \text{in}(l_{i+1})$ and $l_{i+1} \in L_\pi$ or $e \in \text{in}(l_j)$ and $l_j \not\in L_\pi$ and the guard $\phi$ in $e$ has the form $x_2 > c_2$, and

- $x_1$ is reset on all in-going edges to source($e_0$).

Note that in Definition 7.3.1 we assume that none of the outgoing edges of the locations of the cycle can be enabled during intermediate iterations. That is, every time the cycle is visited it will be repeated the maximum allowed number of times before the automaton can leave the cycle. This assumption guarantees that accelerating the execution of a cycle by compressing its iterations will not harm the rest of the analysis of the automaton in the sense that no miss of runs can occur during acceleration. We now discuss when an acceleration operation $\gamma$ can be said to be exact w.r.t the WCET problem.

**Definition 7.3.2.** We say that the acceleration $\gamma$ is exact with respect to the WCET problem if the WCET of the reduced zone graph of an automaton $A$ that results from applying $\gamma$ is equal to the WCET of the concrete zone graph of $A$.

We say that the zone graph $Z^\gamma(A)$ is a reduced zone graph of $A$ if $|Z^\gamma(A)| < |Z(A)|$, where $|Z^\gamma(A)|$ represents the number of states in $Z^\gamma(A)$. Note that the reduced zone graph $Z^\gamma(A)$ that results from accelerating the execution of $A$ by compressing the iterations of the cycles in $A$ will make the verification process more efficient since the reduced graph $Z^\gamma(A)$ will contain fewer states in comparison with the graph $Z(A)$. So the aim of acceleration is to reduce the number of states that are considered in the model checking process, while computing correctly the WCET of the system.
7.3. ACCELERATING WCET CALCULATIONS OF FINITE CYCLES

Since in timed automata model checking we deal with symbolic states rather than concrete states then the delay of a cycle \( \pi \) at iteration \( j \) namely \( \text{delay}(\pi, j) \) will be a time interval of the form \( \triangleright a, b \triangleleft \), where \( 0 \leq a, b \leq \infty \) and hence we need to distinguish between what we call lower delay and upper delay of the cycle at a particular iteration \( j \), namely \( \text{lowerdelay}(\pi, j) \) and \( \text{upperdelay}(\pi, j) \) as described in the following definition.

**Definition 7.3.1.** Let \( E_\pi = (e_0, ..., e_{m-1}) \) be the sequence of edges of a cycle \( \pi \) in an automaton \( A \). Let \( d^j_i = \triangleright a^j_i, b^j_i \triangleleft \) be the delay interval that the automaton can spend at location \( \text{src}(e_i) \) at iteration \( j \). Then \( \text{lowerdelay}(\pi, j) \) and \( \text{upperdelay}(\pi, j) \) can be defined formally as follows.

\[
\text{lowerdelay}(\pi, j) = \sum_{i=0}^{m-1} \inf(d^j_i) = \sum_{i=0}^{m-1} (a^j_i)
\]
\[
\text{upperdelay}(\pi, j) = \sum_{i=0}^{m-1} \sup(d^j_i) = \sum_{i=0}^{m-1} (b^j_i)
\]

For example the cycle in the automaton in Figure 21 has \( \text{lowerdelay}(\pi, j) = 0 \) and \( \text{upperdelay}(\pi, j) = 1 \) for all \( j \geq 1 \). Note that the parameters \( \text{lowerdelay}(\pi, j) \) and \( \text{upperdelay}(\pi, j) \) can be computed on-the-fly during the search using the extra clock \( \delta \) as follows.

\[
\text{lowerdelay}(\pi, j) = |(D_{0,\delta}^{s,j} - D_{0,\delta}^{s,j-1})|
\]
\[
\text{upperdelay}(\pi, j) = (D_{0,\delta}^{s,j} - D_{0,\delta}^{s,j-1})
\]

where \( D_{0,\delta}^{s,j} \) represents the lower bound of the clock \( \delta \) at the start location of the cycle at iteration \( j \), and \( D_{0,\delta}^{s,j} \) represents the upper bound of the clock \( \delta \) at the start location of the cycle at iteration \( j \).

**Lemma 7.3.3.** Let \( A = (\Sigma, L, L_0, L_F, X, I, E) \) be a timed automaton and let \( E_\pi = (e_0, ..., e_{m-1}) \) be the sequence of edges of a cycle \( \pi \) in \( A \). Suppose that the set \( \text{Act}_{\text{delay}} \) represents the set of clocks in \( A \) that affect the delay of \( \pi \) and that all the clocks in \( \text{Act}_{\text{delay}} \) are reset at edge \( e_{m-1} \) (i.e. the last edge of \( \pi \)). Then if \( \text{delay}(\pi, j) = \triangleright \lambda_L, \lambda_U \triangleleft \) for some \( j \geq 2 \), where \( 0 \leq \lambda_L, \lambda_U \leq \infty \), then \( \text{delay}(\pi, k) = \triangleright \lambda_L, \lambda_U \triangleleft \) for all \( k > j \).
Proof. From the assumption that the clocks in $Act_{delay}$ are reset at edge $e_{n-1}$ then every time the location $src(e_0)$ (i.e. the starting location of the cycle) is visited we have $v(x) = 0$ for all $x \in Act_{delay}$ and hence for any two distinct iterations $j,k$ we have $(Z^k_s \setminus inact) = (Z^j_s \setminus inact)$, where $Z^j_s$ is the starting zone of $\pi$ at iteration $j$ and $inact$ represents the set of clocks in $A$ that do not affect the delay of $\pi$. From the semantic definition of the zone approach and from the assumption that the clocks in $Act_{delay}$ are reset at edge $e_{n-1}$ we know that at any iteration $k$ such that $k > j$ the corresponding sequence of generated zones at locations $src(e_0), src(e_1),..., src(e_{m-1})$ at iteration $k$ will be identical w.r.t. $Act_{delay}$ to those obtained at iteration $j$. That is, for any two distinct iterations $j,k$ then after executing the first edge of $\pi$ we have $(\text{succ}(Z^k_s, e_0)) \setminus inact = (\text{succ}(Z^j_s, e_0)) \setminus inact$, and after executing the second edge of $\pi$ we have $(\text{succ}(\text{succ}(Z^k_s, e_1), e_1)) \setminus inact = (\text{succ}(\text{succ}(Z^j_s, e_1), e_{i+1})) \setminus inact,...$, and after executing the last edge of $\pi$ we have $(\text{succ}^{m-2}(\text{succ}(Z^k_s, e_1),...,e_{m-1})) \setminus inact = (\text{succ}^{m-2}(\text{succ}(Z^j_s, e_1),...,e_{m-1})) \setminus inact$, where $\text{succ}^{m-2}$ indicates that the function $\text{succ}$ is repeated $m-2$ times. From the semantic definition of the delay transition in TA and given that the amount of time that can be spent in a certain location is described by means of invariants on a number of clock variables which is in this case the clocks in $Act_{delay}$ it is easy to see then that $\text{delay}(\pi, k) = \text{delay}(\pi, j) = \triangleright \lambda \land \lambda \land \land < \text{ since the corresponding generated zones at iterations } j,k \text{ will be identical w.r.t. } Act_{delay}.$

However, for finite cycles that have syntactic structure as described in Definition
7.3. ACCELERATING WCET CALCULATIONS OF FINITE CYCLES

7.3.1, where the duration of the execution of each iteration is constant, the acceleration is obviously possible: The maximal number of iterations for such cycles \( N \) can be computed on-the-fly during the analysis, and the total duration of the cycle corresponds to the sum of each iteration’s duration. Let us first define the WCET of a cycle \( \pi \) using the delay function.

\[
WCET(\pi) = \sum_{i=1}^{N}(\text{upperdelay}(\pi, i))
\]  

(1)

Formula 1 can be seen as a general formula for computing WCET of cycles which can be sometimes very slow in particular when the finite cycle can be repeated a large number of times. However, when we assume that the delay of each iteration is constant we can rewrite the above formula as follows.

\[
WCET(\pi) = N \times \text{upperdelay}(\pi)
\]  

(2)

Formula 2 can be considered as an acceleration formula for computing WCET in the sense that it allows us to compute the WCET of the cycle without going through the cycle the maximum allowed number of times.

It remains to mention how one can compute the exact value of \( N \). In fact, for finite cycles with constant delays the value of \( N \) can be computed on-the-fly during the search. That is, given the upper bound of the clock \( x_2 \) at the start location of the cycle at the second iteration and the delay of each iteration and the bound that \( x_2 \) is compared to in the cycle let us denote it \( C \), we can then compute \( N \) as follows.

\[
N = \lfloor \frac{C - \text{Init}(x_2, i = 2)}{\text{upperdelay}(\pi_r)} \rfloor
\]

The function \( \text{Init}(x_2, i = 2) \) returns the initial value of the clock \( x_2 \) at iteration 2 (i.e. the value of the clock at the start location of the cycle at iteration 2). Note that the value of \( N \) obtained from the above formula represents the number of full iterations of the cycle. A full iteration is an iteration that starts and ends at the start location of the cycle. On the other hand, the non-full iteration is an iteration in which the start location of the cycle can not be visited twice during that iteration.
Note that the last iteration of finite cycles can be a non-full iteration depending on the position of the edge at which the clock $x_2$ is tested. We explain this in details when discussing the acceleration process of finite cycles. Note also that we start the computations of $N$ from iteration two since the delay of the cycle at the first iteration can be an arbitrary delay as we explain in the following remark.

**Remark 7.3.4.** The delay of the first iteration of cycles can be an arbitrary delay since the starting zone at the first iteration can be an arbitrary zone and hence the delay of the cycle at the first iteration can be different than the delays at the later iterations even for cycles with constant delays.

**Remark 7.3.5.** The delay of the final iteration of finite cycles can be different than the delays at the past iterations of the cycle even for cycles with constant delays since the final iteration can be non-full iteration.

However, for this class of cycles it is easy to see that the WCET computations can be accelerated. The key question is then how the zones inside the cycle should be updated during the acceleration. In fact, the zones need to be updated very carefully so that the acceleration does not affect adversely the rest of the analysis of the automaton. That is, we need to ensure that the applied acceleration is exact. Recall that zones provide a representation of sets of clock interpretations as constraints on (lower and upper) bounds on individual clocks and clock differences. Recall also that the algorithm uses an extra clock $\delta$ which must not be influenced by extrapolation during the analysis. Hence, the constraints involving $\delta$ need to be updated differently than the constraints involving the automaton clocks since the extrapolation constant $M(A)$ should be taken into consideration when updating the constraints involving the automaton clocks.

In Figure 22 we give an example of a finite cycle that have a syntactic structure as described in Definition 7.3.1. The clock $x_1$ determining the delay of the cycle which will be in this example $(2 \times \text{delay})$ time units and the clock $x_2$ determining the termination of the cycle by identifying the maximum number of iterations the
cycle can be repeated, which will be in this example $\left\lfloor \frac{\text{Large}}{2 \times \text{delay}} \right\rfloor$. The locations are depicted as vertices labelled with the name of the location. Location \texttt{start} is the initial location of the automaton. The object \texttt{Large} that appears in a clock guard on the edge from location \texttt{loop} to location \texttt{L1} and in a clock guard on the edge from location \texttt{loop} to location \texttt{end} is a constant natural number. Similarly, the object \texttt{delay} that appears in the guards and invariants of the automaton is a constant natural number. It is clear that time and memory consumption of exploration of the automaton in Figure 22 is very dependent on the value of \texttt{large}. However, as we will see later this can be avoided while computing correctly the WCET of the automaton.

We now turn to discuss how one can accelerate the computations of WCET of such cycles. We call the proposed acceleration procedure as $\gamma$. However, in order to accelerate the WCET calculations one needs to know the value of the following three parameters: (1) the lower delay of the cycle, (2) the upper delay of the cycle, and (3) the maximum number of iterations the cycle can be repeated. These three parameters can be computed on-the-fly during the search. Note that although the cycle has constant lower and upper delays at each symbolic execution, accelerating its computations is still a non-trivial task. We give now the set of steps that are necessary for accelerating correctly the WCET calculations of finite cycles with constant delays.

1. **Updating the lower and upper bound of the extra clock $\delta$.** From the fact that the extra clock does not participate in the computations of the
automaton and does not influence by extrapolation we can then update its lower and upper bound as follows.

\[
D_{0,\delta}^N = D_{0,\delta}^3 - ((N - 2) \times \text{lowerdelay}(\pi))
\]

\[
D_{\delta,0}^N = D_{\delta,0}^3 + ((N - 2) \times \text{upperdelay}(\pi))
\]

where \(D_{0,\delta}^3\) represents the value of the lower bound of the extra clock \(\delta\) at the beginning of the third iteration of the cycle. Note that our proposed acceleration technique may require the cycle to be executed three times: two times before the acceleration is performed and one time after the acceleration is performed. Hence, we accelerate the execution of the cycle at least \((N - 3)\) iterations. These three executions of the cycle make the acceleration process and the necessary computations for the acceleration process much easier. The two iterations that come before the acceleration are necessary for determining the lower and upper bound delay of the cycle and the precise value of \(N\). Again, these parameters can be computed on-the-fly with the help of the extra clock \(\delta\). Recall that the delay of cycles at the first iteration can be an arbitrary value since the starting zone at that iteration can be an arbitrary zone, as we discussed in Remark 7.3.4. However, it perhaps looks somewhat surprising to the reader not including the last iteration of the cycle in the acceleration process so it may be worth providing some motivation why we do so. In fact, the iteration that comes after the acceleration (i.e. the last iteration of the cycle) is necessary for the correct termination of the cycle. Note that the last iteration of finite cycles can be non-full iteration so that the search may not visit the entire set of locations and edges of the cycle during that iteration. This mainly depends on the transition (edge) at which the clock \(x_2\) is tested in the cycle. Note also that since the last iteration of finite cycles can be non-full iteration the delay of the execution of that iteration can be less than the delay of the past iterations and hence careful consideration is needed when dealing with the final iteration of finite cycles.
However, to avoid any further syntactical analysis of the cycle we prefer to leave the last iteration of the cycle to the forward searching technique.

2. **Updating the lower and upper bound of inactive clocks.** Updating the inactive clocks during acceleration is also an easy task since these clocks do not participate in the computations of the cycle and hence they never reset inside the cycle. Hence, the lower and upper bound of an inactive clock \( y \) can be updated as follows.

\[
D^N_{y,0} = (D^3_{y,0} + ((N - 2) \times \text{upperdelay}(\pi)))
\]

\[
D^N_{0,y} = (D^3_{y,0} - ((N - 2) \times \text{lowerdelay}(\pi)))
\]

3. **Updating the lower and upper bound of the clocks \( x_1 \) and \( x_2 \) (i.e. the active clocks in the cycle).** Updating the two clocks \( x_1 \) and \( x_2 \) during the acceleration is also an easy task. Recall that the clock \( x_1 \) is reset at the last transition of the cycle and hence it always has the value zero at the start of each iteration.

\[
D^N_{x_1,0} = D^N_{0,x_1} = 0.
\]

Since we accelerate the execution of the cycle \((N - 3)\) iterations then the lower and upper bound of \( x_2 \) can be updated as follows.

\[
D^N_{0,x_2} = D^3_{0,x_2} - ((N - 2) \times \text{lowerdelay}(\pi))
\]

\[
D^N_{x_2,0} = D^3_{x_2,0} + ((N - 2) \times \text{upperdelay}(\pi))
\]

4. **Updating diagonal constraints involving \( \delta \).** These constraints need to be updated very carefully so that the clock \( \delta \) remains precise until the end of the analysis of the automaton. First note that the clock \( x_1 \) is reset at the last transition of the cycle and hence at the start of each iteration it has the value zero. Note also that the clock \( x_2 \) is not reset inside the cycle and that the inactive clocks are also not reset inside the cycle. Note that we will use the clock \( y \) as an example of inactive clocks in the cycle. Since that all
CHAPTER 7. ACCELERATING WCET ANALYSIS OF TA MODELS

clocks advance at the same rate we can then update the diagonal constraints involving $\delta$ straightforwardly as follows.

$$D_{\delta,x_1}^N = D_{\delta,0}^N;\quad D_{x_1,\delta}^N = D_{0,\delta}^N$$

$$D_{\delta,x_2}^N = D_{\delta,x_2}^1;\quad D_{x_2,\delta}^N = D_{x_2,\delta}^1$$

$$D_{\delta,y}^N = D_{\delta,y}^2;\quad D_{y,\delta}^N = D_{y,\delta}^2$$

5. **Updating diagonal constraints involving the automaton clocks.** In fact, updating the diagonal constraints involving the automaton clocks is very similar to the process of updating the diagonal constraints involving the clock $\delta$ for the same reasons mentioned at step 4.

$$D_{x_2,x_1}^N = D_{x_2,0}^N;\quad D_{x_1,x_2}^N = D_{0,x_2}^N$$

$$D_{y,x_1}^N = D_{y,0}^N;\quad D_{x_1,y}^N = D_{0,y}^N$$

$$D_{y,x_2}^N = D_{y,x_2}^2;\quad D_{x_2,y}^N = D_{x_2,y}^2$$

6. **Extrapolate the matrix that results from the above acceleration operations.** The extrapolation process should be performed while following condition C1 of the partial extrapolation process described in Chapter 5.

7. **Canonicalize the resulting matrix.** The canonicalization process should be performed while following conditions C2 and C3 of the partial extrapolation process described in Chapter 5.

We give now a formal description of the matrix (zone) that results from accelerating the cycle $(N - 3)$ iterations. Note that the following matrix represents the corresponding matrix of the start location of the cycle after accelerating it $(N - 3)$ iterations starting from iteration 3. However, to make the presentation more convenience we denote the term $((N - 2) \times \text{lowerdelay}(\pi_r))$ as $\beta_1$ and the term $((N - 2) \times \text{upperdelay}(\pi_r))$ as $\beta_2$. 
7.3. ACCELERATING WCET CALCULATIONS OF FINITE CYCLES

\[ D^{N-1} = \begin{pmatrix}
  x_0 & \delta & x_1 & x_2 & y \\
  (0, \leq) & ((D^1_{\beta, \delta}, \leq_{\alpha, \beta}) & (0, \leq) & ((D^1_{\gamma}, \leq_{\beta, \delta}) & ((D^1_{\zeta}, \leq_{\alpha, \beta}) \\
  \delta & ((D^2_{\beta, \delta}, \leq_{\alpha, \beta}) & (0, \leq) & ((D^2_{\gamma}, \leq_{\beta, \delta}) & ((D^2_{\zeta}, \leq_{\alpha, \beta}) \\
  x_1 & (0, \leq) & (D^3_{\delta, \leq_{\alpha, \beta}}) & (0, \leq) & (D^3_{\gamma}, \leq_{\beta, \delta}) \\
  x_2 & ((D^4_{\delta, \leq_{\alpha, \beta}}) & ((D^4_{\gamma}, \leq_{\beta, \delta}) & (0, \leq) & (D^4_{\zeta}, \leq_{\alpha, \beta}) \\
  y & ((D^5_{\delta, \leq_{\alpha, \beta}}) & (D^5_{\gamma}, \leq_{\beta, \delta}) & ((D^5_{\zeta}, \leq_{\alpha, \beta}) & (0, \leq)
\end{pmatrix}\]

Recall that \( x_0 \) is the reference clock, \( x_1 \) and \( x_2 \) are the active clocks inside the cycle, \( \delta \) is the extra clock, and \( y \) is used as an example of inactive clocks. The operator \( \leq_{\alpha, \beta} \) refers to the relational operator between the clocks \( \delta \) and \( y \) after executing the cycle two iterations. Note that the relational operators will not be changed between iterations while the lower and upper bound of the clocks and the bound differences between the clocks may be changed between iterations.

In fact, there are cases where the (lower and upper) delays of symbolic executions of finite cycles can be constant (i.e. do not vary between iterations) even that the number of clocks that determine the delay of the cycle is greater than one. This happens when all the clocks that determine the delay of the cycle are reset at the last transition of the cycle. We formalize this in the following definition.

**Definition 7.3.6.** Let \( \mathcal{A} = (\Sigma, L_0, L_F, X, I, E) \) be a timed automaton, let \( E_\pi = (e_0, \ldots, e_{n-1}) \) be the sequence of edges of a cycle \( \pi \) and let \( X_\pi = \{x_1, x_2, \ldots, x_n\} \subseteq X \).

We say that the tuple \((E_\pi, X_\pi)\) is an acceleratable finite cycle, if

- \( I(l) \) has the form \( x_i \sim c \) where \( \sim \in \{<, \leq\} \) and \( 1 \leq i \leq (n - 1) \) for all \( l \in L_\pi \),
- for all \( (l, l', a, \lambda, \phi) \in E_\pi \) then \( \phi \) has the form \( x_i \sim_1 c_1 \) where \( \sim_1 \in \{=, >, \geq\} \) and \( 1 \leq i \leq (n - 1) \) or it has the form \( x_n \sim_2 c_2 \) where \( \sim_2 \in \{=, <, \leq\} \) or the form \( x_i \sim_1 c_1 \wedge x_n \sim_2 c_2 \) where \( 1 \leq i \leq (n - 1) \) and \( \lambda \) is either empty or contains only \( x_i \),
- there exists an edge \( e = (l, l', a, \lambda, \phi) \in E_\pi \) such that \( \phi \) has the form \( x_n \sim_2 c_2 \) or \( x_i \sim_1 c_1 \wedge x_n \sim_2 c_2 \) where \( 1 \leq i \leq (n - 1) \), and
- for all \( l_i \in L_\pi \) and for all \( e \in out(l_i) \) we have either \( e \in in(l_{i+1}) \) and \( l_{i+1} \in L_\pi \) or \( e \in in(l_j) \) and \( l_j \notin L_\pi \) and the guard \( \phi \) in \( e \) has the form \( x_n > c_2 \), and
• for all $i = 1, ..., (n - 1)$ $x_i$ is reset on all in-going edges to $source(e_0)$.

Note that the same procedure described above can be used to accelerate the WCET computation of cycles that have syntactic structure as described in Definition 7.3.6. Note that the clocks $(x_1, x_2, ..., x_{n-1})$ should be treated similar to the way the clock $x_1$ is treated in the above procedure. However, the following two theorems show that the proposed acceleration is exact and effective with respect to the WCET analysis.

**Theorem 7.3.7.** (The proposed acceleration $\gamma$ is exact). For any finite cycle $\pi$ that has a syntactic structure as given in Definition 7.3.1 or 7.3.6 then the acceleration $\gamma$ is exact.

**Proof.** From Definitions 7.3.1 and 7.3.6 we note that the active clocks that affect the delay of the cycle are reset at the last edge of the cycle and by lemma 7.3.3 we know that in such cases the delay of the cycle will not vary between iterations. This is mainly because the starting zone at each iteration of the cycle will be the same with respect to the active clocks that affect the delay. It then suffices to execute the cycle two times in order to compute its WCET correctly since it is guaranteed that delays will not vary between iterations. Hence, compressing the execution of the cycle from $N$ iterations to 2 iterations will not lead to loss of timing information during the analysis. We therefore say that the acceleration $\gamma$ is exact.

**Theorem 7.3.8.** (Effectiveness of acceleration). Let $A = (\Sigma, L, L_0, L_F, X, I, E)$ be a timed automaton, let $E_\pi = (e_0, ..., e_{m-1})$ be the sequence of edges of a cycle $\pi$ that have a syntactic structure as described in Definition 7.3.1 or 7.3.6. Using the above acceleration technique one can compute the WCET of such cycles in only 2-3 iterations while each location of the cycle is visited at most 3 times.

Note that the space and time reduction that can be gained from the above acceleration is significant in particular when the value of $N$ is large. Since the cycle will be explored only three times and hence each location of the cycle will be visited
at most three times. Note that the above acceleration technique compresses the executions of the cycle from \( N \) iterations to 2 iterations and hence reducing the size of the state space that can be generated when analysing such cycles from \((N \times L)\) to \((2 \times L)\). Note that the number of times the cycle is explored during acceleration is independent of the maximum number of iterations and the value of the object Large appears in the given example. Therefore, the above acceleration can speed-up significantly the WCET verification of finite cycles that have syntactic structure as given in Definition 7.3.1 or 7.3.6.

### 7.3.2 Finite Cycles with Complex Structure

In the above section we consider cycles with constant delays where we assume that the active clocks that determine the delay of the cycle are reset at the last edge of the cycle. Such cycles can be detected by checking whether their syntactic structures are conformant to acceleration requirements described in Definitions 7.3.1 and 7.3.6. So the cycles that we studied in the previous section have simple structure. However, there are cycles that have constant delays between iterations but the active clocks that determine their delays are reset at arbitrary edges of the cycle depending on the behaviour of the system being analysed. Such cycles can be also accelerated since their delays are constant between iterations. However, for such cycles one can not detect whether their delays are constant from analysing only their syntactic structure. Hence, a new approach is needed to detect and accelerate such classes of cycles. In this section we propose a more general approach based on the notion of partial fixed points that can be used to accelerate cycles with constant delays where the active clocks that affect the delay can be reset at any edge of the cycle. We first define the notion of partial fixed point and then discuss how one can use it to detect finite cycles with constant delays in which the active clocks that determine their delays are reset at arbitrary edges. We then discuss how one can accelerate the WCET computations of such cycles.

**Definition 7.3.9. (Partial Fixed Point (PFP))** Let \( A = (\Sigma, L, L_0, L_F, X, I, E) \)
be a timed automaton, let $E_\pi = (e_0, ..., e_{m-1})$ be the sequence of edges of a cycle $\pi$ in $A$. Suppose that the set $Y$ represents the set of the constraints that do not influence the delay of $\pi$. The partial fixed point abstraction can be defined in terms of a partial matrix $(Z \setminus Y)$ and a function $f_\pi : (Z) \to (Z)$, where $Z$ is an $n \times n$ matrix. Suppose that we have two points or states $(l_s, Z^j_s)$ and $(l_s, Z^k_s)$ that result from two distinct iterations $j, k$ of a cycle $\pi$, where $l_s$ is the start location of $\pi$. We then say that $(l_s, (Z^j_s \setminus Y))$ is a partial fixed point of $\pi$ and a delay fixed point of $\pi$ if $(Z^j_s \setminus Y) = (Z^k_s \setminus Y)$ and hence $\text{delay}(\pi, j) = \text{delay}(\pi, k)$.

As we mentioned before any two consecutive visits to the start location of the cycle represents a full iteration of the cycle. However, since the extra clock $\delta$ remains precise during the analysis due to the application of the partial extrapolation algorithm (see Chapter 5) then subtracting the lower bound of $\delta$ at any two consecutive visits of the start location gives the lower delay of the cycle and subtracting the upper bound of $\delta$ at any two consecutive visits of the start location gives the upper delay of the cycle at iteration $i$. Note that this is only true if the lower and upper delay of the symbolic cycle are constant between iterations.

**Remark 7.3.10.** When $Z^j_s = Z^k_s$ then $\text{delay}(\pi, j) = \text{delay}(\pi, k)$. However, the opposite is not necessarily true. That is, when $\text{delay}(\pi, j) = \text{delay}(\pi, k)$ then it is not necessary that $Z^j_s = Z^k_s$ since it is possible for the non-identical zones to give the same delay.

In comparison with finite cycles, the infinite cycles have no clock for determining termination since all active clocks in the cycle contributing to the delay of the cycle and hence the cycle can be repeated infinitely often. On the other hand, for finite cycles there are some clocks that contribute to the delay of the cycle and some clocks that contribute to the termination of the cycle. It seems then natural to think about fixed point (FP) abstraction when analysing infinite cycles and about partial fixed point (PFP) abstraction when analysing finite cycles.

Let us explain precisely the reason behind using the FP abstraction and the reason
behind using the PFP abstraction during the analysis of WCET of cycles. The FP abstraction is used inside the cycles in order to detect whether the cycle is an infinite cycle and hence to accelerate the termination of the analysis. On the other hand, the PFP abstraction is used to accelerate the computation of WCET for finite cycles. It is interesting to note also that for infinite cycles it is always guaranteed to reach a fixed point using the extrapolation abstraction. However, it is not the case for finite cycles since it is not always guaranteed to reach a partial fixed point for such cycles and hence the acceleration of WCET computations for finite cycles is not always possible. Consider for example the case where the delay (execution time) of the cycle varies at each iteration. In this case it is necessary to go through the cycle the maximum number of times in order to compute its WCET and hence the PFP abstraction can not be used to accelerate the WCET computation in this case since a PFP can not be reached.

**Lemma 7.3.11.** Let \( A = (\Sigma, L, L_0, L_F, X, I, E) \) be a timed automaton and let \( E_\pi = (e_0, ..., e_{n-1}) \) be the sequence of edges of a cycle \( \pi \). Suppose that the set \( \text{Act}_{\text{delay}} \) represents the set of automaton clocks that affect the delay of \( \pi \). Suppose further that \( \pi \) reaches a fixed point w.r.t \( \text{Act}_{\text{delay}} \) at iteration \( j \) such that \( \text{delay}(\pi, j) = \triangleright \lambda_L, \lambda_U \triangleleft \) where \( 0 \leq \lambda_L \leq \lambda_U \leq \infty \), then for all \( k > j \) we have \( \text{delay}(\pi, k) = \triangleright \lambda_L, \lambda_U \triangleleft \).

**Proof.** From the assumption that \( \pi \) can reach a fixed point w.r.t \( \text{Act}_{\text{delay}} \) at iteration \( j \) then for all \( x \in \text{Act}_{\text{delay}} \) we have \( v(x)^{j-1} = v(x)^j \), where \( v(x)^{j-1} \) represents the value of \( x \) in the zone \( Z_s^{j-1} \) and \( v(x)^j \) represents the value of \( x \) in the zone \( Z_s^j \) and therefore \( ((Z_s^{j-1}) \setminus \text{inact}) = ((Z_s^j) \setminus \text{inact}) \), where \( \text{inact} \) represents the set of clocks that do not affect the delay of \( \pi \). From the semantic definition of the zone approach we know also that \( \text{delay}(\pi, j - 1) = \text{delay}(\pi, j) = \triangleright \lambda_L, \lambda_U \triangleleft \). Also from the assumption that \( \pi \) can reach a fixed point w.r.t \( \text{Act}_{\text{delay}} \) at iteration \( j \) we know that all the clocks in \( \text{Act}_{\text{delay}} \) are reset at some edges of \( \pi \) and hence their values do not accumulate between iterations. Again from the assumption that \( \pi \) can reach a fixed point w.r.t \( \text{Act}_{\text{delay}} \) at iteration \( j \) and by Definition 7.2.2
we know then that at any iteration $k$ such that $k > j$ the sequence of generated zones at locations $src(e_0), src(e_1), ..., src(e_{m-1})$ at iteration $k$ will be identical with those obtained at iteration $j$ w.r.t. $Act_{delay}$. That is, for any two distinct iterations $j, k$ such that $k > j$ then after executing the first edge of $\pi$ we have $(\text{succ}(Z_k^{e_0})) \setminus \text{inact} = (\text{succ}(Z_j^{e_0})) \setminus \text{inact}$, and after executing the second edge of $\pi$ we have $(\text{succ}(\text{succ}(Z_k^{e_i}, e_1))) \setminus \text{inact} = (\text{succ}(\text{succ}(Z_j^{e_1}, e_{i+1}))) \setminus \text{inact}$, and after executing the last edge of $\pi$ we have $(\text{succ}^{m-2}(\text{succ}(Z_k^{e_i}, ..., e_{m-1}))) \setminus \text{inact} = (\text{succ}^{m-2}(\text{succ}(Z_j^{e_i}, ..., e_{m-1}))) \setminus \text{inact}$, where $\text{succ}^{m-2}$ indicates that the function $\text{succ}$ is repeated $m - 2$ times. From the semantic definition of the delay transition in TA and given that the amount of time that can be spent in a certain location is described by means of invariants on a number of clock variables which is in this case the clocks in $Act_{delay}$ it is easy to see then that $\text{delay}(\pi, k) = \text{delay}(\pi, j) = \lambda_L, \lambda_U$ since the corresponding generated zones at iterations $j, k$ will be identical w.r.t. $Act_{delay}$ due to the reset operations.

We now turn to discuss how one can accelerate the WCET computations of cycles that have constant delays such that the active clocks that determine their delays are reset at arbitrary edges. Note that the same approach described at Section 7.3.1 can be used to accelerate the computations of such cycles but some modifications are needed. However, since we assume that the active clocks that affect the delays of the cycle can be reset at any transition of $\pi$ then some of the steps and formulas given in the above process need to be modified. In particular, the formulas that update the individual and diagonal constraints involving the active clocks that affect the delay of the cycle. The other formulas are still valid for such class of cycles since the only change that we make here with respect to the cycles that we studied in the previous section is that the clocks that affect the delay of the cycle can be reset at any arbitrary edge of $\pi$. However, the fact that the delays of the cycle are constant between iterations and that the clocks advance at the same rate make the acceleration process of this class of cycles a straightforward process. We discuss
7.3. ACCELERATING WCET CALCULATIONS OF FINITE CYCLES

here only the steps that need some modification during the acceleration of such cycles. Note that we use the clocks $x_1$ and $x_2$ as examples of the active clocks that determine the delay of the cycle. We use also the clock $y$ as an example of inactive clocks with respect to the cycle under analysis and the clock $x_3$ is the clock that determines the termination of the cycle.

- **Updating the lower and upper bound of the clocks $x_1$, $x_2$, and $x_3$ (i.e. the active clocks in the cycle).** Note that the values of the active clocks that are reset inside $\pi$ do not accumulate between iterations. Now since the delays of the cycle are constant between iterations then the value of the clocks $x_1$ and $x_2$ after accelerating the cycle will be the same as at iteration 2. Hence, the lower and upper bound of the clocks $x_1$ and $x_2$ will be updated as follows.

\[ D_{0,x_1}^N = D_{0,x_1}^3; \quad D_{x_1,0}^N = D_{x_1,0}^3 \]
\[ D_{0,x_2}^N = D_{0,x_2}^3; \quad D_{x_2,0}^N = D_{x_2,0}^3 \]

On the other hand, the value of the clock $x_3$ will accumulate between iterations since it is not reset inside the cycle and hence its value can be updated as follows.

\[ D_{0,x_3}^N = D_{0,x_3}^3 - (N - 3) \times \text{lowerdelay}(\pi) \]
\[ D_{x_3,0}^N = D_{x_3,0}^3 + (N - 3) \times \text{upperdelay}(\pi) \]

- **Updating diagonal constraints involving $\delta$.** Since the delays of the cycle are constant between iterations and that the value of the clock $\delta$ accumulates and increases by a fixed rate at each iteration and the fact that the clocks $x_1$ and $x_2$ are reset at each iterations then the constraints that describe the relationship of $\delta$ to $x_1$ and $x_2$ can be updated as follows.

\[ D_{\delta,x_1}^N = D_{\delta,x_1}^3 + ((D_{\delta,x_1}^3 - D_{\delta,x_1}^2) \times (N - 3)). \]
\[ D_{x_1,\delta}^N = D_{x_1,\delta}^3 + ((D_{x_1,\delta}^3 - D_{x_1,\delta}^2) \times (N - 3)). \]
\[ D_{\delta,x_2}^N = D_{\delta,x_2}^3 + ((D_{\delta,x_2}^3 - D_{\delta,x_2}^2) \times (N - 3)). \]
$D^N_{x_2, \delta} = D^3_{x_2, \delta} + ((D^3_{x_2, \delta} - D^2_{x_2, \delta}) \times (N - 3))$.

- **Updating diagonal constraints involving the automaton clocks.** In fact, updating diagonal constraints involving the automaton clocks is also straightforward since the clock $x_3$ that determines the termination of the cycle and the inactive clock $y$ are not reset inside the cycle and hence their values accumulate and increase by a constant rate at each iteration. Also the diagonal constraints involving the two clocks $x_1$ and $x_2$ can be updated straightforwardly during acceleration since both $x_1$ and $x_2$ are reset inside the cycle.

  \[ D^N_{x_1, y} = D^3_{x_1, y} + ((D^3_{x_1, y} - D^2_{x_1, y}) \times (N - 3)) \]

  \[ D^N_{y, x_1} = D^3_{y, x_1} + ((D^3_{y, x_1} - D^2_{y, x_1}) \times (N - 3)) \]

  \[ D^N_{x_1, x_3} = D^3_{x_1, x_3} + ((D^3_{x_1, x_3} - D^2_{x_1, x_3}) \times (N - 3)) \]

  \[ D^N_{x_3, x_1} = D^3_{x_3, x_1} + ((D^3_{x_3, x_1} - D^2_{x_3, x_1}) \times (N - 3)) \]

  \[ D^N_{x_1, x_2} = D^3_{x_1, x_2}; \quad D^N_{x_2, x_1} = D^3_{x_2, x_1} \]

- **Extrapolate and canonicalize the resulting accelerated zone afterwards.**

### 7.4 An Algorithm for Computing WCET with Accelerations

Algorithm 12 represents an algorithm for computing WCET of systems with accelerations. The algorithm performs on-the-fly checks to detect whether the cycle under exploration is conformant to acceleration requirements. Acceleration can be used then to fasten the state space exploration. The algorithm takes as input an automaton $A$ for the system to be analysed. Each node in the computed tree is of the form $(l_i, Z_i^7, sts)$ where $l_i$ is a location in the automaton, $Z_i^7$ is the corresponding abstracted zone, and $sts$ is an integer variable which is assigned to each state.
in order to detect whether there exists a cycle on locations in the behaviour of the automaton. The variable \( sts \) can take values from the set \( \{0, 1, 2\} \). When it is 0 it means that the location has not been visited before, when it is 1 it means the location has been visited before but not fully explored, and when it is 2 it means that everything reachable from that location have been explored. We assume that the reader is familiar with the classical DFS algorithm with the labelling process of nodes to unvisited (0), being explored (1), and finished (2) and hence we omit these details. The algorithm uses two data structures \( \text{WAIT} \) and \( \text{PASSED} \) to store symbolic states waiting to be examined, and the states that already examined, respectively. The \( \text{WAIT} \) set is instantiated with the initial symbolic state \( (l_0, Z_0, 0) \). The global variable \( \text{WCET} \) holds the currently best known longest execution time of reaching the final location; initially it is 0. The global clock \( \delta \) keeps track of the execution time of the system.

In each iteration of the while loop, the algorithm selects a symbolic state \( s \) from \( \text{WAIT} \), checking if the state is a final state. If the state does not evolve to any new state then we consider it as a final state of some branch in the graph. If the state \( s \) is a final state we update the best known \( \text{WCET} \) to the upper bound value of \( \delta \) at \( s \) if it is greater than the current value of \( \text{WCET} \). If the state is not a final state, we add all successors of \( s \) to \( \text{WAIT} \) and continue to the next iteration. During the search, if the algorithm encounters a reachable location that is not guarded by an invariant the search can stop immediately since the \( \text{WCET} \) will be infinity. Similarly, if the search discovers that there exists an infinite realizable cycle in the automaton then it stops immediately since the \( \text{WCET} \) will be infinity. The algorithm also detects on-the-fly whether there exists a selfcycle in the automaton which is a cycle of length one. That is, if there exists a transition that connects a location to itself. Such cycles can be detected easily even without searching the \( \text{PASSED} \) list by checking whether the successor of a particular state leads to a state that has the same control location. It is interesting to note that the extrapolation procedure used in the algorithm is the partial extrapolation that satisfies conditions
C1-C4 (see Chapter 5 for more details). Note also that we do not use the inclusion abstraction [BY04] in the WCET algorithm since inclusion abstraction may lead to stop the exploration when some zone is “included” inside an already explored zone. This way of cutting exploration based on inclusion is correct for reachability properties, but not when properties critically involve detection of cycles in the timed automaton.

Note that when the search detects that the cycle under analysis is acceleratable it calls some special operations in order to accelerate the execution of the cycle by compressing (if possible) its iterations. However, in order for these operations to accelerate the execution of the cycle they need to know the value of some certain parameters. For example, for finite cycles with constant delays the value of the lower bound delay of the cycle, the value of the upper bound delay of the cycle, and the number of iterations are necessary information for accelerating such cycles. The boolean operation checkAcceConformant() used in the algorithm to check whether the cycle under analysis can be accelerated. This can be performed by checking whether the cycle is conformant with the acceleration requirements described in Definitions 7.3.1 and 7.3.6. However, if the cycle is conformant to the acceleration requirements the operation accelerateCycleExec() will be called to accelerate the execution of the cycle as described in Section 7.3.1. Note that the search is parametrized by an abstraction \( \gamma \), which is a composition of the abstractions \( \gamma_{\text{extra}} \) (extrapolation), \( \gamma_{\text{act}} \) (activity), \( \gamma_{\text{FP}} \) (fixed point), and \( \gamma_{\text{PFP}} \) (partial fixed point). The correctness of the method comes from the fact that composition of abstractions respects property preservation [DT98]. Depending on \( \gamma \), the function \( post_e \) implements the chosen abstractions. We explain this in what follows.

**Extrapolation and activity.** These abstractions are implemented by modifying the successor function \( post_e \). When none of these abstractions is used, then \( post_e^\gamma \) is simply \( post_e \). When extrapolation is used, then \( post_e^\gamma(s) \) is \( \text{Extra}_M(post_e(s)) \), where \( M \) is computed as explained in Section 7.2.1. When activity abstraction is used, then \( post_e^\gamma(s) \) is \( post_e^\gamma(s) \setminus \text{inact} \). When both abstractions are used, both operators
are applied and the order does not matter.

**Fixed point and partial fixed point abstractions.** This abstraction is implemented by modifying the test $Z = Z'$ for the zones inside the cycles. We ignore both inactive clocks and the extra clock $\delta$ when applying fixed point and partial fixed point abstractions.

**Algorithm 12:** An efficient algorithm for computing WCET of diagonal-free TA

```
Input: (A)
Output: WCET := 0
clock \delta
PASSED := \emptyset; WAIT := \{(l_0, Z_0, 0)\}
while WAIT \neq \emptyset
    select s from WAIT
    // Check if s is a final node on some branch of the tree
    if for all $a \in \Sigma post\text{\_}^e(s) = \emptyset$ then if upperBound(s.$Z^\gamma$, $\delta$) > WCET
        then WCET := upperBound(s.$Z^\gamma$, $\delta$)
        add s to PASSED
    for all $s'$ such that $s \rightarrow s'$ do
        // if there exists a location that is not guarded with an invariant
        if upperBound(s'.$Z^\gamma$, $\delta$) = \infty then {WCET := \infty; WAIT := \emptyset; break}
        // Check if the cycle can be accelerated.
        else if s'.l = s''.l \land s''.sts = 1 for any s'' \in PASSED then
            if checkAcceConformant() then {accelerateCycleExec(); continue}
        // if there exists an infinite realizable self-cycle in the automaton
        else if s.l = s'.l \land s.$Z^\gamma$ = s'.$Z^\gamma$ \land (s'.$Z^\delta_0$ > s.$Z^\gamma_0$) then
            {WCET := \infty; WAIT := \emptyset; break}
        // if there exists an infinite non-realizable self-cycle in the automaton
        else if s.l = s'.l \land s.$Z^\gamma$ = s'.$Z^\gamma$ then continue
        // if there is an infinite realizable cycle in the automaton of length \geq 2
        else if s'.l = s''.l \land s.$Z^\gamma$ = s''.$Z^\gamma$ \land (s'.$Z^\delta_0$ > s''.$Z^\gamma_0$) \land s''.sts = 1
            for any s'' \in PASSED then {WCET := \infty; WAIT := \emptyset; break}
        // if there is an infinite non-realizable cycle in the automaton of length \geq 2
        else if s'.l = s''.l \land s.$Z^\gamma$ = s''.$Z^\gamma$ \land s''.sts = 1 for any s'' \in PASSED
            then continue
        else add s' to WAIT
return WCET
```
7.5 Implementation

In this section we briefly summarise our prototype implementation of the model checking algorithms given in Section 7.4. The prototype implementation has been developed using the opaal tool [DHJ+11] which has been designed to rapidly prototype new model checking algorithms. The opaal tool is implemented in Python and is a standalone model checking engine. Models are specified using the UPPAAL XML format. The main step in the implementation of the algorithms is the representation of sets of symbolic state and the operations required on them. We use the open source UPPAAL DBM library for the internal symbolic representation of time zones in the algorithms.

We give here a set of simple timed automata that we use to verify the correctness and the efficiency of our algorithm. The algorithm was able to give the correct answer of WCET for each automaton within a reasonable time. We compare also our algorithm with this implemented in the model checker UPPAAL. In UPPAAL, one can use a global clock GBL and check the upper bound for termination of system A: (sup { A.end } : GBL). The sup operator is documented in the Help menu of UPPAAL. The analysis shows that UPPAAL fails to terminate when verifying the automata in Figures 23 and 24 which have an infinite WCET. These automata can be verified almost instantly using our algorithm. On the other hand, the automata in Figures 25 and 26 represents automata with finite cycles which we use to verify the efficiency of our algorithm in comparison to UPPAAL’s algorithm. We vary the value of the object Loop during verification which can be any natural number until the analysis fails to terminate due to the state space explosion problem. Note that we report x when the analysis fails to terminate. As we can see the cycle in Figure 25 can be accelerated using our proposed algorithm and the little cycle in Figure 26 can be accelerated as well since they are conformant to the acceleration requirements discussed in the chapter. The verification results reported in Table 1 show that our algorithm can outperform UPPAAL’s algorithm by many orders of magnitude and it can handle largely repetitive cycle with constant delays very
7.6 Conclusions

In this chapter we presented a new algorithm for computing WCET of systems with cyclic behaviour. The proposed algorithm uses a set of accelerations that improve significantly the efficiency of WCET analysis of real-time systems. We compared also our algorithm with this implemented in the model checker UPPAAL which shows that the proposed algorithm can handle cases that UPPAAL fails to verify, where we show that when infinite cycles exist, UPPAAL’s algorithm may efficiently. On the other hand, when verifying finite cycles with large values for the object Loop UPPAAL’s algorithm suffers from the state space explosion, thus leading to a low efficiency or resource exhaustion.

Figure 23: A TA with infinite realizable cycle

Figure 24: Three intersecting finite cycles that lead to an infinite WCET
Figure 25: A TA with largely repetitive finite cycle

Figure 26: A TA containing a composite cycle
### 7.6. CONCLUSIONS

<table>
<thead>
<tr>
<th>Automaton</th>
<th>WCET</th>
<th>UPPAAL’s Algor.</th>
<th>Our Algor.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fig. 23</td>
<td>$\infty$</td>
<td>x</td>
<td>1.02s</td>
</tr>
<tr>
<td>Fig. 24</td>
<td>$\infty$</td>
<td>x</td>
<td>2.65s</td>
</tr>
<tr>
<td>Fig. 25 with Loop = $10^4$</td>
<td>$10^4$</td>
<td>4.73s</td>
<td>3.6s</td>
</tr>
<tr>
<td>Fig. 25 with Loop = $10^5$</td>
<td>$10^5$</td>
<td>520s</td>
<td>3.6s</td>
</tr>
<tr>
<td>Fig. 25 with Loop = $10^6$</td>
<td>$10^6$</td>
<td>50,918s</td>
<td>3.6s</td>
</tr>
<tr>
<td>Fig. 25 with Loop = $10^7$</td>
<td>$10^7$</td>
<td>x</td>
<td>3.6s</td>
</tr>
<tr>
<td>Fig. 26 with Loop = $10^4$</td>
<td>1104</td>
<td>364.65s</td>
<td>129s</td>
</tr>
<tr>
<td>Fig. 26 with Loop = $10^5$</td>
<td>10104</td>
<td>33,877s</td>
<td>493s</td>
</tr>
<tr>
<td>Fig. 26 with Loop = $10^6$</td>
<td>100104</td>
<td>x</td>
<td>1053s</td>
</tr>
<tr>
<td>Fig. 26 with Loop = $10^7$</td>
<td>1000104</td>
<td>x</td>
<td>4983s</td>
</tr>
</tbody>
</table>

Table 1: Comparing the performance of UPPAAL’s algorithm and our algorithm for different cases

not terminate, and when largely repetitive finite cycles exist, UPPAAL’s algorithm suffers from the state space explosion, thus leading to a low efficiency or resource exhaustion.
Chapter 8

Conclusions

8.1 Summary

The thesis presented novel zone-based algorithms for computing worst case execution time (WCET) or maximum termination time of real-time systems using the timed automata (TA) model checking technology. The algorithms can work on any arbitrary diagonal-free TA and can handle more cases than previously existing algorithms for WCET computations, as it can handle cycles in TA and decide whether they lead to an infinite WCET. In general, WCET analysis is undecidable (equivalent to halting problem), which states that it is undecidable to determine whether or not an execution of a system will eventually halt. However, for TA models one can use model-checking techniques to analyse the system and compute the WCET. We showed soundness of the proposed algorithms and studied their complexity. The solutions provided here are conceptually a marked improvement over some earlier work on the problem, in which repeated guesses (guided by binary search) and multiple model checking queries were effectively but inelegantly and less efficiently used; here only one run of the zone construction is sufficient to yield the answers. Thus, the proposed solution can be a significant break-through in computing WCET.
The thesis also proposed a set of acceleration techniques that improve the efficiency of WCET verification of cyclic real-time systems. We prove that the proposed accelerations are exact with respect to the WCET problem and demonstrate that model checking WCET with the proposed acceleration techniques can significantly speed-up the verification of WCET of real-time systems. We also compare our algorithms with this implemented in the state-of-the-art model checker UPPAAL which shows that the proposed algorithms can handle cases that UPPAAL fails to verify, where we showed that in certain circumstances, when infinite cycles exist, UPPAAL’s algorithm may not terminate, and when largely repetitive finite cycles exist, UPPAAL’s algorithm suffers from the state space explosion, thus leading to a low efficiency or resource exhaustion.

The thesis presented also a set of new operations for improving the reachability analysis of Timed Automata (TA) using the Difference Bound Matrices (DBMs), namely the partial canonicalization and the partial extrapolation of DBMs. The partial canonicalization allows one to fix the non-tightness introduced by extrapolation by updating only the clock constraints that have been changed during extrapolation, thus reducing the impact on the run time of the reachability algorithm of the canonicalization step. The proposed partial canonicalization algorithms are specializations of Floyd’s algorithm that have a time complexity of $O(c \times n)$ instead of $O(n^3)$ for the standard approach, where $n$ is the number of clocks in the automaton and $c$ is the number of variables that have been changed during extrapolation. We demonstrated that model checking TA with partial canonicalization can speed-up considerably the verification time of several interesting examples including the Philips audio control protocol and Fischer’s protocol. On the other hand, the partial extrapolation helps to perform a more precise analysis of the reachable states of the timed automaton and it is necessary for several applications of TA including the problem of computing minimum and maximum termination times.

The thesis also reported some lack of precision in previously published algorithmic of zones (sets of clock valuations) and difference bound matrices (a data structure
to represent and handle zones). In fact, these algorithms are rarely discussed in
details. In particular the extrapolation, canonicalization, and inclusion checking
operations and their role in forward reachability algorithm with respect to certain
problems such as the minimum termination time problem and the maximum termi-
nation time problem require extra care and non-trivial arguments for proving both
correctness and termination (see [Bou04]). With this in mind, any generalization
raises challenging questions and we believe that the partial canonicalization and
the partial extrapolation of difference bound matrices proposed in this thesis are of
great importance.

We believe that the algorithms, techniques, and optimisations proposed in this
thesis are of great importance and can be adopted by the state-of-the-art model
checking tool UPPAAL to improve the efficiency and the reliability of the imple-
mentation of the \textit{inf} and \textit{sup} operators. In fact, the majority of the dense time
model checking tools such as Kronos, RED, and Rabbit do not support features
for computing minimum and maximum termination times of systems and hence the
solutions proposed in this thesis can be valuable solutions since they can be used
to extend these tools.

\section{Future Work}

We intend to investigate new different techniques for analysing maximum termina-
tion time or worst-case execution time (WCET) of real-time systems other than TA
model and DBM data structure. For example we would like to investigate whether
the different variants of TA can be used efficiently to compute WCET of systems
such as the stop-watch TA [TLR08] . We would like also to investigate whether the
other data structures for representing zones such as Clock Restriction Diagrams
(CRD) and Clock Difference Diagrams (CDD) can be used efficiently to analyse
WCET of systems. Also TA with cycles (loops) that have non-constant delays be-
tween iterations have not been studied in this thesis so it might be interesting to
investigate techniques for accelerating WCET of such type of cycles in particular finite cycles with non-constant delays that can be iterated a large number of times.
Bibliography


[BFLS05] Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Philippe Schnoebelen. Flat acceleration in symbolic model checking. In *Automated


symposium on Operating systems principles, SOSP ’81, New York, NY, USA, 1981. ACM.


BIBLIOGRAPHY


[RPW08] Bernhard Rieder, Peter Puschner, and Ingomar Wenzel. Using model checking to derive loop bounds of general loops within ANSI-C applications for measurement based WCET analysis. In International


