FUNDAMENTAL STUDIES ON A DECOMPOSITIONAL AND HYBRID APPROACH TO AUTOMATIC VERIFICATION OF COMPONENT-BASED SYSTEMS

By
GAOYAN XIE
D.O.B.: 03/03/1975

A dissertation submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY
COMPUTER SCIENCE

Washington State University
School of Electrical Engineering and Computer Science

August 2005
To the Faculty of Washington State University:

The members of the Committee appointed to examine the dissertation/thesis of GAOYAN XIE find it satisfactory and recommend that it be accepted.

______________________________
Chair

______________________________
Publications

Refereed Journal Publications

Linear Reachability Problems and Minimal Solutions to Linear Diophantine Equation Systems.
In *Theoretical Computer Science*, Vol. 328(1-2): 203-219, 2004. This is the journal version of our CIAA’03 paper, which was invited to submit to TCS.

A Review of Some Rigorous Software Design and Analysis Tools.

Reachability Problems for Dense Counter Machines.
Submitted to *Information and Computation*. A short version of this paper appeared in CAV’03

Quadratic Diophantine Equations and Verification of Infinite State Systems with Parameterized Constants.
Submitted to *Information and Computation* and is currently under revision. A short version of this paper appeared in ICALP’03
Refereed Symposium Publications


Model checking of P systems.

CTL Model checking for Systems with Unspecified Components.

Real-Counter Automata and Applications to Verification.

An Automata-theoretic Approach for Model checking Systems with Unspecified Components.

Testability of Oracle Automata.

New Complexity Results for Some Linear Counting Problems Using Minimal Solutions to Linear Diophantine Equations.

Dense Counter Machines and Verification Problems.

A Solvable Class of Quadratic Diophantine Equations with Applications to Verification of Infinite State Systems.
Acknowledgments

I feel it really a luck to have Prof. Zhe Dang as my Ph.D advisor. Prof. Dang has done far more than being an exceptional advisor, he has also been a great friend. His strong support, constant encouragement, friendly criticism and generosity with his time for discussions have been the key factors in every progress I've made toward my degree. I especially thank Prof. Dang for showing me the right way and the right attitude to do research. Without all of these helps from him, I would never have reached this point in my life.

I am grateful to Prof. Dyreson and Prof. Andrews for participating in my doctoral committee and for the helpful discussions. I am also grateful to many colleagues who provided various feedbacks on various parts of this work. They are Prof. Oscar Ibarra, Prof. Pierluigi Pietro, and Mr. Cheng Li. Of course, I must thank all the good people who make the School of EECS at Washington State University and its facilities work. The school of EECS provides great support for graduate students and I consider myself privileged to have been part of it. I also acknowledge that this work has also been partially supported by the NSF Grant CCF-0430531.

I am greatly indebted to my parents for enduring the hardship in the past five years while I am far away from them and unable to help. I want to thank my daughter Cynthia who has brought me the unspeakable happiness and joy in the past two years. Last and above all, I want to thank my wife Yun for her confidence, support, patience and unconditional love, which in the end, is all that matters.
FUNDAMENTAL STUDIES ON A DECOMPOSITIONAL AND HYBRID APPROACH TO AUTOMATIC VERIFICATION OF COMPONENT-BASED SYSTEMS

Abstract

by GAOYAN XIE, Ph.D
Washington State University
August 2005

Chair: Zhe Dang

This work introduces a decompositional and hybrid approach for the automatic verification of component-based systems and studies some fundamental issues on the approach. By using a formal verification technique like model checking or automata operations, the approach first derives from the system specification and some system property a verification condition for each individual component in the system, such that the system satisfies the property if and only if all the components satisfy their verification conditions. Then the approach checks the validity of an individual component’s verification condition either by model checking or automata operations when the component’s specification is available, or by traditional black-box testing when the component’s specification is not available. We first study the possibility of verifying a system with black-box components through testing by introducing a theoretic tool called Oracle Automata. Then we study specific model checking (both LTL and CTL) algorithms for systems with only one finite-state black-box. Next, we show a decompositional technique on testing a system with multiple black-boxes. We also consider the problem of verifying a system with only one component but against non-temporal properties, as well as verifying a system with only one infinite-state component.
Contents

Publications iii
Acknowledgments vi
Abstract vii

1 Introduction 1
  1.1 Problem Statement and Analysis .............................. 2
  1.2 A Decompositional and Hybrid Approach ........................ 4
  1.3 Scope of This Work ............................................. 5
    1.3.1 Testability of Oracle Automata ............................ 5
    1.3.2 Model Checking Systems with One Black-box Component .......... 6
    1.3.3 Decompositional Testing .................................... 7
    1.3.4 The Linear Reachability Problem ............................ 8
    1.3.5 A Solvable Class of Quadratic Diophantine Equations .......... 9

2 Preparations 11
  2.1 The Component Model .......................................... 11
  2.2 The System Model .............................................. 14
  2.3 Model Checking ................................................ 20
    2.3.1 CTL Model Checking ....................................... 20
    2.3.2 LTL Model Checking ....................................... 21
  2.4 Semi-linear Languages and Presburger Formulas .................. 23
3 Testability of Oracle Automata

3.1 Definitions .......................................................... 26
3.2 Oracle Finite Automata .............................................. 27
3.3 Testing Emptiness for Oracle Finite Automata .................. 30
  3.3.1 The Testability of OFA With Regular Oracles ............... 32
  3.3.2 The Testability of OFA With Context-free Oracles .......... 33
  3.3.3 The Testability of OFA With Semi-linear Oracles .......... 33
3.4 Testing Emptiness for Oracle Buchi Automata ................... 34
  3.4.1 Testability of \(\omega\)-OFA With Regular Oracles ........... 36
  3.4.2 Testability of \(\omega\)-OFA With Context-free Oracles ....... 36
  3.4.3 Testability of \(\omega\)-OFA With Semi-linear Oracles ....... 36
3.5 A Dynamic Testing Algorithm ..................................... 37
3.6 Some Verification Problems ...................................... 40
  3.6.1 The Reachability Problem .................................. 40
  3.6.2 The Safety Problem ....................................... 41
  3.6.3 The LTL Model Checking problem .......................... 42
3.7 Summary ............................................................... 44

4 Model Checking Systems With One Black-box

4.1 Definitions .......................................................... 46
4.2 LTL Model Checking .................................................. 47
  4.2.1 Liveness Analysis .......................................... 47
  4.2.2 Liveness Testing .......................................... 48
  4.2.3 LTL Model Checking Driven Testing ......................... 51
4.3 CTL Model Checking .................................................. 52
  4.3.1 The Ideas .................................................. 52
  4.3.2 Processing a CTL formula ................................. 55
  4.3.3 Checking an EX Sub-Formula ............................... 57
  4.3.4 Checking an EU Sub-Formula ............................... 59
  4.3.5 Checking an EG Sub-Formula ............................... 60
  4.3.6 Testing a Witness Graph ................................. 62
4.3.7 Testing an EX Graph ........................................... 63
4.3.8 Testing an EU Graph ........................................... 64
4.3.9 Testing an EG Graph ........................................... 66
4.4 Examples ............................................................ 69
4.5 Discussions .......................................................... 76
  4.5.1 Practical Efficiency ............................................. 77
  4.5.2 Coverage Metrics .............................................. 78
  4.5.3 More Complex System Models ............................... 78

5 Decompositional Testing .............................................. 79
  5.1 Introduction ....................................................... 80
  5.2 The Push-in Technique ........................................... 82
    5.2.1 Theory Foundation of the Push-in Technique ............ 83
    5.2.2 Automata Generation in Step $i$ ............................ 85
    5.2.3 Surviving Set Generation in Step $i$ ......................... 86
    5.2.4 Correctness and Bad Behavior Generation ................ 87
  5.3 Experiments ....................................................... 89
  5.4 Summary .......................................................... 92

6 The Linear Reachability Problem .................................... 95
  6.1 Introduction ....................................................... 96
  6.2 Definitions ....................................................... 98
  6.3 A Bounding Box for the Linear Reachability Problem ....... 100
  6.4 The Linear Liveness Problem .................................... 106
  6.5 Summary .......................................................... 107

7 Quadratic Diophantine Equation Systems .......................... 108
  7.1 Introduction ....................................................... 109
  7.2 Preliminaries ..................................................... 111
  7.3 Semi-linear Languages with Weights ........................... 119
  7.4 Applications ..................................................... 126
    7.4.1 Finite-State Systems and Their Extensions ............... 128
7.5 Summary .................................. 134

8 Conclusions ................................ 135
  8.1 Related Work .............................. 136

A Proofs Omitted In The Dissertation ........ 140
  A.1 Proofs Omitted From Chapter 3 ........... 140
      A.1.1 Proof of Theorem 3.1 .................. 140
      A.1.2 Proof of Theorem 3.2 .................. 141
      A.1.3 Proof of Theorem 3.3 .................. 141
      A.1.4 Proof of Theorem 3.4 ................. 144
      A.1.5 Proof of Theorem 3.6 ................. 150
      A.1.6 Proof of Theorem 3.7 ................. 151
      A.1.7 Proof of Theorem 3.8 ................. 153
  A.2 Proofs Omitted From Chapter 7 ........... 153
      A.2.1 Proof of Theorem 7.8 .................. 153

Bibliography ................................. 158
List of Tables

5.1 Experiment Results: Counts of Test Sequences . . . . . . . . . . . 89
5.2 Experiment Results: Time Efficiency . . . . . . . . . . . . . . . 92
List of Figures

2.1 A Data Acquisition System ................................. 15
2.2 The Environment of the Data Acquisition System ........ 15
2.3 The Gluer .................................................. 18
2.4 Internal Implementation of Timer .......................... 18
2.5 Internal Implementation of Sensor ......................... 19
2.6 Internal Implementation of Comm ........................ 19

3.1 A Simple Communication System .......................... 41
3.2 An Oracle Finite Automaton ................................ 42

4.1 Example 4.3 ................................................. 69
4.2 Communication Graph of Example 4.3 ...................... 69
4.3 Example 4.4 ................................................. 70
4.4 Communication Graph of Example 4.4 ...................... 70
4.5 Witness Graph for Example 4.5 ............................ 71
4.6 Example 4.6 ................................................. 71
4.7 Witness Graph for Example 4.6 ............................ 72
4.8 Witness Graph 2 for Example 4.7.1 ....................... 73
4.9 Witness Graph 3 for Example 4.7.1 ....................... 74
4.10 Witness Graph 2 for Example 4.7.2 ...................... 75
4.11 Witness Graph 3 for Example 4.7.2 ...................... 75
4.12 Witness Graph 4 for Example 4.7.2 ...................... 75
4.13 Witness Graph 5 for Example 4.7.2 ...................... 76
6.1 An Example of A Scheduler .......................... 97
6.2 A Layered Structure ................................. 103

7.1 A Simplified Packet-Based Network Switch ................. 129
7.2 A Design of The Scheduling Unit ......................... 130
Chapter 1

Introduction

Component-based software development [62, 17] (sometimes also called component-based software engineering—CBSE) is an engineering practice to build large software systems by assembling software components, which are either developed specifically for the system, reused from a previous project in the same organization, contracted from third party software vendors, or even purchased as commercial-off-the-shelf (COTS) products. This development method promises the great benefits of reusing valuable software assets, reducing development and maintenance costs, improving productivity, etc. In the past decade, it has been increasingly adopted by the software industry, and at least in late 1990s, the software components market was already at the billion dollar scale and was also projected to increase substantially in the following years [9].

However, this development method has also posed serious challenges to the quality assurance issue of component-based systems—system developers may lose the quality control of externally obtained software components, which could therefore lead to serious system failures. This issue is of vital importance to safety-critical and mission-critical systems. For instance, in June 1996, during the maiden voyage of the Ariane 5 launch vehicle, the launcher veered off course and exploded less than one minute after taking off. The report [69] of the Inquiry Board indicates that the disaster resulted from insufficiently tested software reused from the Ariane 4. The developers had reused certain Ariane 4 software component in the Ariane 5 without substantially
testing it in the new system, having assumed that there were no significant differences in these portions of the two systems.

As agreed by many researchers in this area, the long-term success of component-based development really depends to a great extent on an established science and technology foundation for achieving predictable quality in component-based systems. In recent years, there have been lots of research efforts on building that foundation and as part of these efforts, this dissertation deals with some fundamental issues related to the automatic verification of component-based systems.

1.1 Problem Statement and Analysis

Most of the current work addresses the quality assurance issue of component-based systems from the viewpoint of component developers; i.e., they are mainly concerned with how to ensure the quality of a component before it is released. However, this view is fundamentally insufficient since a component may be deployed in a wide variety of environments, which theoretically can not be tried out by its developers or third party laboratories. Therefore, a component may still not perform as well as expected in a particular deployment system, even after extensive testing/verification by its developers or third party laboratories.

We, however, look at the issue from a system developer’s point of view; i.e., we are concerned with how to ensure that components function correctly in a particular deployment environment. More precisely, we are interested in the following problem:

(*) Given: a component-based system $Sys$ that consists of a set of components $C_1, \cdots, C_k$ for some $1 \leq k$ and a desired property $P$ about the observable behaviors of $Sys$

Question: will the observable behaviors of the system $Sys$ satisfy $P$?

Notice that, essentially this is a verification problem since we are seeking a definite answer to the problem.

In practice, testing is the most widely used technique for answering such questions
CHAPTER 1. INTRODUCTION

about the observable behaviors of software. Actually, the extreme end of testing (exhaustive testing) is akin to verification, but this extreme end is generally unachievable. To answer the above question, the straightforward approach for most system developers is to hook up all the components together and conduct integration/system testing, trying to expose all possible observable behaviors of the system. Unfortunately, there are serious challenges to this straightforward approach. First, when some component’s source code or specification is not available (which is very common for externally obtained components), it is hard to choose test cases for the system and it is hard to know when the system has been adequately tested. Second, integration/system testing over a component-based system could be extremely difficult since components in the system may run concurrently and concurrency is notoriously hard to test. Third, integration/system testing is not always applicable for a component-based system. This is because, in many applications, software components could be applied for upgrading or extending a running system [87] that is costly or not supposed to shut down for testing at all.

Besides the above challenges, the impossibility of being exhaustive also makes purely testing-based approaches insufficient to establish a solid confidence in the quality of a component-based system that requires high reliability, for which formal verification techniques like model checking are highly desirable. However, one fundamental obstacle of using model checking to address the problem in (*) is that, existing model checking techniques require a complete and formal specification of the system to be verified, while the design detail or source code of an externally obtained software component is generally not available to system developers (in this work, we call such a component a black-box component or just black-box in short). Additionally, even if we have all the specifications of the components in a system, to model check the system against some property, we will have to first compose all the components’ specification together to obtain a global specification, which causes the “state explosion” problem. Last, existing model checking techniques only work for finite-state systems, which is not always the case for software components.

In summary, neither purely testing-based approaches nor purely formal verification techniques are sufficient to address the difficulties of the quality assurance of
component-based systems brought by their unique characteristics, especially the difficulties caused by the concurrency among multiple components and the existence of black-box components. Nevertheless, combining testing with formal verification techniques is considered to be a viable approach that may avoid the pitfalls of either purely software testing techniques or purely formal verification techniques while taking advantages of both.

In this work, we propose a decompositional and hybrid approach to address the problem in (*) (i.e., how to check that a component-based system $Sys$ with $k$ number of components $C_1, \ldots, C_k$ satisfies a given property $P$).

1.2 A Decompositional and Hybrid Approach

Specifically, we model each component as a labeled transition system (LTS), which is not necessarily of finite state space, and model a component-based system as a collection of LTSs that synchronously communicate with each other through output and input actions. This abstract LTS model seems to be simple but is actually flexible enough to model a wide variety of systems. We use temporal logic or regular expressions to specify the desired properties about the observable behaviors of a component-based system.

The idea of our approach can be outlined as follows. From the system $Sys$ and the system property $P$, we first automatically derive some verification condition $P_i$ for each component $C_i$, for all $1 \leq i \leq k$ such that the system satisfies the property $P$ if and only if component $C_i$ satisfies its verification condition $P_i$, for all $1 \leq i \leq k$. Thus, a verification problem for a component-based system can be reduced to verification problems for individual components in the system. This is why we call our approach a decompositional approach. Deriving the verification conditions can be done through some formal verification techniques like model checking or automata operations. Then we check the validity of an individual component’s verification condition either by model checking or automata operations when the component’s specification is available, or by traditional black-box testing when the component is a black-box. This is why we call our approach a hybrid approach.
CHAPTER 1. INTRODUCTION

Compared to the straightforward testing-based approach, the advantages of our approach are

- a stronger confidence in the reliability of a component-based system can be established through both formal verification and adequate black-box testing;

- the challenges to integration/system testing of a component-based system can be avoided, and especially testing an individual component can be customized with respect to a specific system property; and

- the whole process can be carried out in an automatic way.

1.3 Scope of This Work

Certainly, our decompositional and hybrid approach is not aimed at completely solving the problem in (*). This is because the LTS model used in the approach is essentially universal; i.e., it has the computation power of Turing Machines such that no algorithm exists to completely solve any non-trivial problem over the model. Hence, we approach the problem by studying some fundamental issues of the problem over some restricted forms of the system model. To be exact, we first address the possibility of verifying a system with black-boxes through testing by introducing Oracle Automata. Then we study model checking (both LTL and CTL) systems with only one finite-state black-box. Next, we show a new technique on testing a system with multiple black-boxes. We also consider the problem of verifying a system with only one component but against non-temporal properties, as well as verifying a system with only one infinite-state component. These studies are outlined in the subsequent subsections in the above order respectively.

1.3.1 Testability of Oracle Automata

As we mentioned in Section 1.2, eventually the verification problem in (*) will be reduced to verification problems (checking the validity of a verification condition \( P_i \)) for each component \( C_i \). When \( C_i \) is a black-box component (i.e., no available
specification), we use traditional black-box testing techniques to check whether the verification condition $P_i$ is valid over $C_i$. But this is not always doable, since the black-box component $C_i$ may have an infinite state space and have very complex behaviors such that the validity of $P_i$ over $C_i$ may not be answered through testing. So, the first question we are interested in is:

**Question:** under what conditions can we solve a verification problem for a black-box component through testing?

To answer this question, we introduce oracle (finite) automata that are finite / Buchi automata augmented with oracles in some classes of formal languages, and study the testability of some important verification problems (such as reachability, safety, LTL model checking, etc.) for oracle (finite) automata. In this study, oracles are intended to capture the observable behaviors of a system’s black-box components, so an oracle is defined as a language. On each transition of an oracle (finite) automaton, the automaton may query some oracle and the query result decides whether the transition can be fired. Here, queries to an oracle are used to characterize communications between a system and its black-box components.

We show that the verification problems for an oracle (finite) automaton can be reduced to the emptiness problem of oracle (finite) automata, which can be solved by testing (querying) the oracles with *bounded-length* test cases. Then we give some results on the testability of the emptiness problem of oracle (finite) automata with various classes of oracles. We also give a symbolic algorithm to perform bounded-testing for the emptiness problem.

With these testability results, we next study specific (CTL/LTL) model checking algorithms for systems with only one, finite-state, black-box component.

### 1.3.2 Model Checking Systems with One Black-box Component

Specifically, we study the following model checking problem:

- **Given:** a component-based system $Sys$ that consists of two components $M$ and $X$ where $M$ is well specified, $X$ is a black-box component with finite state
space, and a temporal formula $f$ that specifies some desired property about the system

- **Check**: whether the system $Sys$ satisfies $f$

We present a set of new algorithms for both LTL and CTL formulas, called *model checking driven black-box testing*, which combine model checking techniques and black-box testing techniques to solve the problem.

As mentioned in Section 1.2, the idea of our algorithms is to use a model checking based technique that automatically derives from the rest of the system (i.e., $M$) a verification condition over the black-box component $X$. This verification condition guarantees that the system $Sys$ satisfies the requirement $f$ iff the condition is satisfied by the black-box component $X$. In the algorithms, the verification condition is represented as communication graphs (for LTL) or witness graphs (for CTL) and the validity of the condition is checked through testing the black-box component $X$. Test-cases are generated by a bounded and nested depth-first search procedure over the communication graphs (for LTL) or witness graphs (for CTL). Since the black-box is assumed to have a finite state space, our algorithms are both sound and complete, provided there is an upper bound for the number of states in the black-box.

However, for systems with multiple finite-state black-boxes, the model checking algorithms (especially the CTL algorithms) are very difficult to apply. Also, the assumption of a black-box component having only finite state space might be too restrictive for some applications. So, we also consider verifying a system with multiple (and not necessarily finite-state) black-boxes against a simple form of properties, namely, a set of bounded-length behaviors.

### 1.3.3 Decompositional Testing

The global testing problem we study in this work is to seek a definite answer (i.e., essentially this is a verification problem) to whether a system of multiple black-boxes has an observable behavior in a given finite (but could be huge) set $Bad$. We introduce a novel approach to solve the problem that does not require integration testing.
CHAPTER 1. INTRODUCTION

Instead, by exploiting the synchronous nature of the communications among the components, our approach reduces the global testing problem to testing individual black-boxes in the system one by one in some given order. Using an automata-theoretic approach, test sequences for each individual black-box are generated from the system's description as well as the test results of black-boxes prior to this black-box in the given order. Since the number of test sequences for a component is finite, we do not require the component to be of finite state space to achieve the completeness of the approach; i.e., the components could be of infinite state space. In contrast to the conventional compositional/modular verification/testing approaches, our approach is essentially decompositional. Also, our technique is complete, sound, and can be carried out automatically. Our experiments results show that the total number of tests needed to solve the global testing problem is very small even for an extremely large \textit{Bad}.

As mentioned earlier, most difficulties of verifying component-based systems are caused by the existence of black-boxes. However, even when a system that has only one specified component, verification could still be very hard, especially for properties that are out of the expressiveness scope of temporal logic. For instance, event counting is a fundamental concept to specify some important fairness properties about a system, yet it can not be expressed in either CTL or LTL. In the next subsections, we study the linear reachability problem for systems with only one, finite-state, and specified component (i.e., a FLTS). The solution to this problem is a novel application of a known result in linear Diophantine equation systems.

1.3.4 The Linear Reachability Problem

The linear reachability problem for finite-state transition systems is to decide whether there is an execution path in a given finite-state transition system such that the counts of labels on the path satisfy a given linear constraint. Using some known results on minimal solutions (in nonnegative integers) for linear Diophantine equation systems, we obtained new time complexity bounds for the problem. In contrast to the previously known results, the bounds obtained in this paper are polynomial in
the size of the transition system in consideration, when the linear constraint is fixed. The bounds are also used to establish a worst-case time complexity result for the linear reachability problem for timed automata. With this (small) bound, solving the linear reachability problem could be efficiently done by exhaustively checking all the bounded-length paths in the system.

Traditional model checking techniques work only on finite-state systems. However, in real world, a software component is often of infinite state space. The successes of finite-state model checking have greatly inspired researchers to develop automatic techniques for analyzing infinite-state systems (such as systems that contain integer variables and parameters). However, in general, this is an undecidable problem. Therefore, one focus of the research in this area is to identify what kinds of practically useful infinite-state models are decidable with respect to a particular form of properties (e.g., reachability).

1.3.5 A Solvable Class of Quadratic Diophantine Equations

In this work, we study a class of infinite-state systems that contain parameterized or unspecified constants. Significantly different from existing techniques for analyzing infinite-state systems (e.g., automata-theoretic techniques in [64, 15, 32], computing transitive closures for Presburger transition systems [30, 19], etc.), we use a technique closely related to nonlinear Diophantine equation systems. We show that various verification problems over this class of infinite-state systems can be reduced to the satisfiability problem of a special class of quadratic Diophantine equation systems. Then we show that the satisfiability problem is actually solvable.

Specifically, a $k$-system consists of $k$ quadratic Diophantine equations over nonnegative integer variables $s_1, \ldots, s_m, t_1, \ldots, t_n$ of the form:

$$\sum_{1 \leq i \leq l} B_{ij}(t_1, \ldots, t_n)A_{ij}(s_1, \ldots, s_m) = C_i(s_1, \ldots, s_m)$$

$$\vdots$$

$$\sum_{1 \leq i \leq l} B_{kj}(t_1, \ldots, t_n)A_{kj}(s_1, \ldots, s_m) = C_k(s_1, \ldots, s_m)$$

where $l, n, m$ are positive integers, the $B$'s are nonnegative linear polynomials over $t_1, \ldots, t_n$ (i.e., they are of the form $b_0 + b_1 t_1 + \ldots + b_n t_n$, where each $b_i$ is a nonnegative
CHAPTER 1. INTRODUCTION 10

integer), and the A’s and C’s are nonnegative linear polynomials over \( s_1, \ldots, s_m \). We show that it is decidable to determine, given any 2-system, whether it has a solution in \( s_1, \ldots, s_m, t_1, \ldots, t_n \), and give applications of this result to some interesting problems in verification of infinite-state systems. The general problem is undecidable; in fact, there is a fixed \( k > 2 \) for which the \( k \)-system problem is undecidable. However, certain special cases are decidable and these, too, have applications to verification.

In the next chapter, we present the system model used in this work, introduce the basics of model checking, and define some concepts used in this work. Then, we address in detail the above issues in the subsequent chapters in the same order. The last chapter is a brief conclusion and a comparison with related work.
Chapter 2

Preparations

2.1 The Component Model

In this work, we model a component as a labeled transition system (LTS) that moves from one state to another state by performing an action.

Definition 2.1 A labeled transition system is a quadruple \( T = \langle S, \text{Init}, \nabla, R \rangle \), where

- \( S \) is a (potentially infinite but countable) set of states\(^1\),
- \( \text{Init} \subseteq S \) is a set of initial states,
- \( \nabla = \) is a finite set of labels, and
- \( R \subseteq S \times \nabla \times S \) defines the transition relation.

\[ \]

In the above definition, the set \( \nabla \) can be further partitioned into three pair-wise disjoint subsets:

- \( \Lambda \), where each label denotes an internal action,
- \( \Pi \), where each label denotes an input action, and

\(^1\)When \( S \) is a finite set, \( T \) is called a finite-state labeled transition system (FLTS).
• $\Gamma$, where each label denotes an output action.\(^2\)

Especially, the set $\Sigma = \Pi \cup \Gamma$, i.e., the set of observable actions of $T$, is called the interface of $T$. When the interface is the only known part in its definition, $T$ is considered to be a black-box.

**Example 2.2** Consider the LTS in Figure 2.4, which has two states, two input actions (pause and resume) and one output action (fire).\(^3\)

For each transition $t = (s, a, s') \in R$, if $a \in \Lambda$ (resp. $a \in \Pi$, $a \in \Gamma$) then $t$ is called an internal (resp. input, output) transition of $T$. At a state $s$, $T$ can always move to another state $s'$ by performing an action $a$ when $t$ is an internal transition of $T$. However, only by synchronizing (which will be elaborated in the next subsection) over an action $a$ with another component, can $T$ move from a state $s$ to another state $s'$ when $t$ is either an input or an output transition of $T$.

An execution of $T$ is an alternating sequence of states in $S$ and actions in $\nabla$: $s_1a_1 \cdots s_{h-1}a_{h-1}s_h$ (for some $h$) such that $s_1 \in \text{Init}$ and $(s_j, a_j, s_{j+1}) \in R$ for each $1 \leq j \leq h - 1$. A behavior of $T$ is a sequence $\tau$ of actions in $\nabla$ such that there is an execution $\epsilon$ of $T$ and $\tau$ is the result of dropping all the states from $\epsilon$. An observable behavior of $T$ is the result of dropping all the internal actions from a behavior. Trivially, the empty string is an observable behavior for any unit $T$.

An action $a$ is enabled at state $s \in S$ if there exists a transition $(s, a, s') \in R$ for some $s' \in S$. We say $T$ is deterministic if at any state, $T$ has only one choice about where to go after performing an action; i.e., $\forall s \in S, a \in \nabla, s_1, s_2 \in S$, if $(s, a, s_1) \in R$ and $(s, a, s_2) \in R$ then $s_1 = s_2$. We further say $T$ is strongly deterministic if $T$ is deterministic and at any state, if more than one action is enabled then all of these actions are input actions; i.e., $\forall s \in S, a_1, a_2 \in \nabla, s_1, s_2 \in S$, if $(s, a_1, s_1) \in R$ and $(s, a_2, s_2) \in R$ then $a_1 \in \Pi$ and $a_2 \in \Pi$.

**Example 2.3** Still consider the LTS in Figure 2.4. Obviously, it is deterministic but not strongly deterministic.\(^3\)

\(^2\)In the rest of this work, labels and actions will be used interchangeably without further explanation.

\(^3\)Throughout the figures in this work, suffixes ? and ! will be used to distinguish input and output actions respectively.
CHAPTER 2. PREPARATIONS

A test sequence $\alpha$ for $T$ is just a sequence of observable actions of $T$. When $T$ is a black-box, we say $T$ is testable if

- There is a special action $\text{reset}$ that makes $T$ return to an initial state from any state; i.e., $\text{reset} \in \Pi$ and $\forall s \in S, (s, \text{reset}, s') \in R$ for some $s' \in \text{Init}$.

- There is a black-box testing procedure $\text{BBtest}(T, \cdot)$ such that, for any test sequence $\alpha$ for $T$,
  - $\text{BBtest}(T, \alpha)$ returns “yes” (i.e., $\alpha$ is successful) if $\alpha$ is an observable behavior of $T$, and
  - $\text{BBtest}(T, \alpha)$ returns “no” (i.e., $\alpha$ is unsuccessful) otherwise.

Example 2.4 Consider the black-box component $\text{Comm}$ in Figure 2.1, which has seven observable actions. If we assume that the black-box was implemented as shown in Figure 2.6. Clearly, $\text{send msg ack}$ would be a successful test sequence for $\text{Comm}$ while $\text{send msg fail}$ would not.

It’s easy to see that if one further assumes that the black-box is strongly deterministic, then an input action sequence decides an unique output sequence. Thus, a test sequence for the black-box can be simply represented as a sequence of input actions. However, there are testable components that are not necessarily output deterministic (e.g., [74, 89, 80]). Therefore, to make this work more general, in the definition, a test sequence is always a sequence of both input actions and output actions.

Remark. Certainly, it should be pointed out that the LTS model is not sufficient to cover all aspects of a complex software component, which may include unbounded data values, dynamic data structures, and recursive procedure calls, etc. For these computations, this work was not intended. However, this LTS model is indeed flexible enough to model some very import logic of a real software component, like the procedure call graphs, synchronizations in concurrent programs, and the stimulus-response relations in reactive systems, etc. For instance, output actions can be used

---

4The black-box testing procedure can be implemented in practice for a variety of transition systems [16].
CHAPTER 2. PREPARATIONS

...to model calling a procedure, sending a message, and returning to its caller upon the termination of a procedure, etc. Input actions can be used to model entering a procedure, receiving a message, and the end of calling a procedure, etc. At present, automatic verification of software is only practical over such abstract models.

2.2 The System Model

Intuitively, a component-based system can be understood as a collection of concurrently running components. The components communicate with each other either synchronously or asynchronously. Without loss of generality, this work considers only systems of synchronously communicating components.

Since in this work a component is modeled as a LTS, a component-based system $Sys$ consisting of $1 \leq k$ number of communicating components can thus be written as

$$Sys = (C_1, \ldots, C_k),$$

where each $C_i = \langle S_i, Init_i, \nabla_i, R_i \rangle$, $1 \leq i \leq k$ is a LTS model for one of the components. The LTSs' state sets $S_1, \ldots, S_k$ are pair-wise disjoint. Their internal action sets $A_1, \ldots, A_k$ are also pair-wise disjoint. Let $\Lambda = A_1 \cup \cdots \cup A_k$ and $\Sigma = \Sigma_1 \cup \cdots \cup \Sigma_k$, then $\Lambda \cap \Sigma = \emptyset$; i.e., in the system $Sys$, no action can be both an internal action of some component and an observable action of another component. However, the interfaces $\Sigma_1, \ldots, \Sigma_k$ may not be pair-wise disjoint; i.e., some components may share some common observable actions and especially, an action can be both an input action of some component and an output action of another component.

In practice, a system always resides in some particular environment and constantly interacts with the environment. When designing a system, we usually do not care about the dynamics of its environment (assume that the environment can act according to any logics), but we do often assume a set of possible actions that the environment may perform. These actions may model human operations over the

---

5 Asynchronous communications with lossless and bounded channels can always be emulated by synchronous communications with an added environment.
system or the system’s request for the environment’s attention, etc. Formally,

**Definition 2.5** the environment of a component-based system $Sys = (C_1, \cdots, C_k)$ is also LTS, $ENV = \langle S_0, Init_0, \nabla_0, R_0 \rangle$, where

- $|S_0| = |Init_0| = 1$; i.e., the environment has only one state, say $s_0$.
- $\nabla_0 = \Sigma_0$; i.e., the environment has no internal actions, and
- $R_0 = S_0 \times \nabla_0 \times S_0$; i.e., the environment can perform any action at any moment.

Obviously, to specify the environment of a system, it is sufficient to just specify the environment’s interface. Sometimes in this work, even the interface is omitted when it is clear from the context.

![Figure 2.1: A Data Acquisition System](image)

![Figure 2.2: The Environment of the Data Acquisition System](image)

**Example 2.6** Consider a data acquisition system shown in Figure 2.1, which consists of four components: **Gluer Timer**, **Sensor** and **Comm** (note that Timer, Sensor and Comm are black-boxes). In this system, action $fire$ is both an output action of Timer and an input action of Gluer; action $msg$ is both an output action of Comm and an input action of the environment while action $ack$ is both an input action of Comm and an output action of the environment. Also look at Figure 2.2 for its environment. ■
CHAPTER 2. PREPARATIONS

Let \( I = \{0, 1, \cdots k\} \). A \textit{synchronization} in the system \( \text{Sys} \) is a triple \( \text{syn} = (o, a, IN) \), where \( o \in I \), \( a \in \Sigma \), and \( IN \subseteq I \setminus \{o\} \). At any moment in the system (suppose that each component \( C_i \) is in state \( s_i \) for all \( 1 \leq i \leq k \)), the synchronization \( \text{syn} \) is \textit{enabled} if the following two conditions are satisfied:

- there is an output transition \((s_o, a, s'_o) \in R_o\) in component \( C_o \) (or the environment when \( o = 0 \)),
- there is an input transition \((s_j, a, s'_j) \in R_j\) for all \( j \in IN \).

Essentially, a synchronization of an output action and multiple input actions represents a communication among the components or between the environment and the components in the system. Obviously, this synchronization model allows both one-to-one communications and multi-cast communications.

\textbf{Example 2.7} Consider the data acquisition system shown in Figure 2.1, an implementation of the \textit{Timer} component in Figure 2.4, and an implementation of the \textit{Sensor} component in Figure 2.5, a synchronization over \textit{fire} is enabled whenever the \textit{Timer} is at state \( s_0 \) and the \textit{Sensor} is at state \( s_0 \) or \( s_1 \). ■

\textbf{Remark.} We assume that the behavior of the environment of a system is unconstrained. Actually, the design of a system can always enforce an assumption about the environment’s behaviors using synchronous communications. For instance, the design of an object-oriented software component may assume that the calls to its methods occur in a specific order and it can enforce this assumption by blocking the out-of-order calls.

As mentioned earlier, in a component-based system \( \text{Sys} \), all its components run concurrently but in an \textit{interleaving} fashion. That is, at any moment, either only one internal action is performed in some component or only one synchronization occurs; i.e., if two internal actions or two synchronizations are enabled at the same moment, the system nondeterministically picks only one of them to carry out, and no internal actions and synchronizations can be carried out at the same time. So, a component-based system can also be considered as an LTS by interleaving its components’ internal actions as well as all possible synchronizations.
CHAPTER 2. PREPARATIONS

Definition 2.8 A component-based system $Sys = (C_1, \cdots, C_k)$ is also a LTS $Sys = \langle S, Init, \nabla, R \rangle$, where

- $S = S_1 \times \cdots \times S_k$ is the (global) state set,
- $Init = Init_1 \times \cdots \times Init_k$ is the (global) initial state set; i.e., the system $Sys$ is in an (global) initial state if all of its components are in their (local) initial states,
- $\nabla = \nabla_1 \cup \cdots \cup \nabla_k$ is the action set, and
- $R \subseteq S \times \nabla \times S$ is the (global) transition relation.

For each $a \in \Sigma$, let $In(a)$ denote the set of all the components that share the same input action $a$; i.e., $In(a) = \{i| 1 \leq i \leq k \wedge a \in \Pi_i\}$. Similarly, let $Out(a)$ denote the set of all the components that share the same output action $a$; i.e., $Out(a) = \{i| 1 \leq i \leq k \wedge a \in \Gamma_i\}$. Then, a global transition that makes the system move from a global state $(s_1, \cdots, s_k)$ to another global state $(s'_1, \cdots, s'_k)$ while performing an action $a \in \nabla$ is in $R$ iff one of the following conditions is satisfied:

- $a$ is an internal action of some component and by this transition, only that component performs this action while the other components do not move; i.e., $\exists 1 \leq i \leq k(a \in \Lambda_i \wedge (s_i, a, s'_i) \in R_i \wedge \forall j \leq 1 \neq i \leq k(s_j = s'_j))$,
- $a$ is an observable action and by this transition, only one synchronization over this action occurs; i.e., $\exists 0 \leq o \leq k(o \in Out(a) \wedge (s_o, a, s'_o) \in R_o \wedge \forall 0 \leq i \neq o \leq k(i \in In(a) \wedge (s_i, a, s'_i) \in R_i \wedge i \not\in In(a) \wedge s_i = s'_i))$.

Remark. It shall also be pointed out that in the system $Sys$: if a global transition is through an internal action of some component, this action is still internal to the system; if a global transition is through a synchronization of an output action and multiple input actions among some components (or the environment), these actions are considered to be one single action, and we do not discriminate whether it is output or input but just treat it as an observable action to the environment.
CHAPTER 2. PREPARATIONS

As defined earlier, a sequence $\tau \in \mathcal{V}^*$ is a behavior of the system $Sys$ if the system has an execution from an initial global state to some global state and $\alpha$ is the result of dropping all the states along the execution. And an observable behavior of $T$ is the result of dropping all the internal actions from a behavior.

![Diagram of the Gluer]

Figure 2.3: The Gluer

![Diagram of the Timer]

Figure 2.4: Internal Implementation of Timer

**Example 2.9** Still consider the data acquisition system shown in Figure 2.1. The system works as follows. Once started, the Timer keeps sending (an output action) a fire message when the time interval set runs out; the Timer can also be paused (resp. resumed) by receiving (an input action) a pause (resp. resume) command. The Sensor is supposed to respond to the fire message by sending a data message when the sensor’s reading is ready; it also sends a serr message when something is wrong inside the Sensor. The Comm component responds to a send command to send some data by sending a msg message to the environment (the underlying physical network); it responds to an ack (resp. nack) message by sending an ok (resp. fail) message to indicate that the data associated with a previous send command has been transmitted successfully (resp. unsuccessfully) by the underlying network; it sends an cerr message when something is wrong inside Comm. The Gluer (whose transition graph is depicted in Figure 2.3) simply relays data from Sensor to Comm; it pauses the
Timer when something is wrong with the Sensor or Comm, and after that, it resumes the Timer when either an ok or fail is received from Comm. Together, they constitute a data acquisition system, which periodically transmits a reading of the Sensor through Comm via some underlying communication network. In this system, the Gluer and the three components run concurrently and interact with each other through synchronizations of output and input actions (here, all synchronizations are between a pair of components (or the environment)). Assume that the internal implementations of the three components are shown in Figure 2.4, Figure 2.5, and Figure 2.6, respectively. It can be seen (though not obviously) that the following sequence is an observable behavior of the system: fire fire serr pause data send msg ack ok resume fire, while sequence fire fire serr data pause send is not.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{sensor_diagram.png}
\caption{Internal Implementation of Sensor}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{comm_diagram.png}
\caption{Internal Implementation of Comm}
\end{figure}

**Remark.** When none of the components in a system is a black-box, our system model is roughly equivalent to the IOTS studied in [80]. Our model is also closely related to I/O automata [70] (but ours is not input-enabled) and to interface-automata [35] (but ours, similar to the IOTS, makes synchronizations among components (or the environment) observable at the system level). These observable synchronizations are the key to testing the behavior of a system of black-boxes components, where an abstract model (such as design or source code) of each black-box component is unavailable.
2.3 Model Checking

Model checking is an automatic technique for verifying a finite-state system against some temporal specification. The system is usually represented by a Kripke structure \( K = \langle S, R, L \rangle \) over a set of atomic propositions \( AP \), where

- \( S \) is a finite set of states;
- \( R \subseteq S \times S \) is the (total) transition relation;
- \( L : S \rightarrow 2^{AP} \) is a function that labels each state with the set of atomic propositions that are true in the state.

The temporal specification can be expressed in, among others, a computation-tree logic (CTL), in which there can be many possible futures at one time, or a linear-time logic (LTL), in which there is only one future at one time. Both CTL and LTL formulas are composed of path quantifiers \( A \) and \( E \), which denote “for all paths” and “there exists a path”, respectively, and temporal operators \( X, F, U \) and \( G \), which means “next state”, “eventually”, “until”, and “always”, respectively.

2.3.1 CTL Model Checking

More specifically, CTL formulas are defined as follows:

- Constants \( true \) and \( false \), and every atomic proposition in \( AP \) are CTL formulas.

- If \( f_1 \) and \( f_2 \) are CTL formulas, then so are \( \neg f_1, f_1 \land f_2, f_1 \lor f_2, EX\ f_1, AX\ f_1, EF\ f_1, AF\ f_1, E[f_1 \ U \ f_2], A[f_1 \ U \ f_2], EG\ f_1, AG\ f_1 \).

A CTL model checking problem, formulated as \( K, s_0 \models f \), is to check whether the CTL formula \( f \) is true at state \( s \). For example, \( AF\ f \) is true at state \( s \) if \( f \) will be eventually true on all paths from \( s \); \( E[f \ U \ g] \) is true at state \( s \) if there exists a path from \( s \) on which \( f \) is true at each step until \( g \) becomes true.

The explicit algorithm [28] for solving this problem operates by searching the structure and, during the search, labeling each state \( s \) with the set of sub-formulas
of $f$ that are true at $s$. Initially, labels of $s$ are just $L(s)$. Then, the algorithm goes through a series of stages—during the $i$-th stage, sub-formulas with the $(i - 1)$-nested CTL operators are processed. When a sub-formula is processed, it is added to the labels for each state where the sub-formula is true. When all the stages are completed, the algorithm returns true when $s_0$ is labeled with $f$, or false otherwise.

Due to duality, any CTL formula can be expressed in terms of $\neg$, $\lor$, $EX$, $EU$, and $EG$. Thus, each intermediate state of the algorithm only handles six cases, depending on whether the sub-formula is atomic or has one of the following forms: $\neg f$, $f_1 \lor f_2$, $EX f_1$, $E[f_1 U f_2]$, or $EG$. The details of algorithms that handle these six cases can be found in the textbook [28].

A symbolic model checking algorithm based on BDDs [18] was proposed by McMillan [59] and has been implemented into a model checker SMV [2], which has been successfully used for verifying many industrial-level applications [27, 22, 23, 49].

### 2.3.2 LTL Model Checking

LTL formulas, on the other hand, are all in the form of $A f$ where $f$ is a *path formula* defined as follows:

- Constants true and false, and every atomic proposition in AP are path formulas.
- If $f_1$ and $f_2$ are path formulas, then so are $\neg f_1$, $f_1 \land f_2$, $f_1 \lor f_2$, $X f_1$, $F f_1$, $[f_1 U f_2]$, $G f_1$.

An LTL model checking problem, formulated as $K, s_0 \models A f$, is to check whether the path formula $f$ is true on all paths from $s$. For example, $AG f$ is true at state $s$ if on all paths from $s$, after a future point $f$ will be always true; $AGF f$ is true at state $s$ if on all paths from $s$, $f$ will be true infinitely often.

A classic LTL model checking technique is an automata-theoretic approach by Vardi and Wolper [90] that first translate the negation of an LTL formula into a Buchi automaton and then check the emptiness of the production of the Buchi automaton and the system.
CHAPTER 2. PREPARATIONS

Simply, a Buchi automaton is a finite automaton over infinite words. More precisely, a Buchi automaton $\mathbb{A}$ is a five tuple

$$\mathbb{A} = \langle S, \text{Init}, \text{Final}, \Sigma, R, \rangle$$

where

- $S$ is the finite set of states,
- $\text{Init} \subseteq S$ is the set of initial states,
- $\text{Final} \subseteq S$ is the set of accepting states,
- $\Sigma$ is a finite set of alphabeta, and
- $R \subseteq S \times \Sigma \times S$ is the transition relation.

A run $p$ of a Buchi automaton is an infinite sequence $s_0a_0s_1a_1\ldots$ such that $s_0 \in \text{Init}$ and $(s_i, a_i, s_{i+1}) \in R$ for every natural number $0 \leq i$. Let $inf(p)$ be the set of states that appear infinitely often in the run $p$. Then $p$ is accepting if and only if $F \cup inf(p) \neq \emptyset$. Let $p_\Sigma$ be the results of dropping all states from $p$. Then the language accepted by an Buchi automaton $\mathbb{A}$ is the set of words $L(\mathbb{A}) = \{ p_\Sigma | p$ is an accepting run of $\mathbb{A} \}$.

Given any LTL formula $f$, it can be translated into an Buchi automaton $\mathbb{A}_f$ such that $f$ is true along an infinite path $p$ if and only if $p \in L(\mathbb{A}_f)$. Also, a Kripke structure $M$ can be simply extended into a Buchi automaton where every state is an accepting state. Then the original LTL model checking problem can be solved as follows

- obtain $\neg f$ by complementing the LTL formula $f$,
- translate $\neg f$ into a Buchi automaton $\mathbb{A}_{\neg f}$,
- extend the Kripke structure $M$ into a Buchi automaton $\mathbb{A}_M$,
- compute the product $\mathbb{A} = \mathbb{A}_M \times \mathbb{A}_{\neg f}$, and


- check the emptiness of $A$: $L(A) = \emptyset \iff M, s \models A f$.

The successful LTL model checker SPIN [52] was built upon the above automata-
theoretic approach.

More detailed background in model checking and temporal logic can be found in
the textbook [28].

2.4 Semi-linear Languages and Presburger Formulas

Let $\mathbb{N}$ be the set of nonnegative integers and let $\Sigma$ be an alphabeta with $\Sigma = 
\{a_1, a_2, \ldots, a_k\}$ for some positive $k$. A subset $S$ of $\mathbb{N}^k$ is a linear set if there
exist vectors $v_0, v_1, \ldots, v_t$ in $\mathbb{N}^k$ such that $S = \{v \mid v = v_0 + b_1 v_1 + \cdots + b_t v_t, b_i \in \mathbb{N}\}$. The set $S \subseteq \mathbb{N}^k$ is semi-
linear if it is a finite union of linear sets. For each word $w$ in $\Sigma^*$, define the Parikh map of $w$ to be $\#(w) = (|w|_{a_1}, |w|_{a_2}, \ldots, |w|_{a_k})$, where $|w|_{a_i}$
denotes the number of symbol $a_i$'s in word $w$, $1 \leq i \leq k$. For a language $L \subseteq \Sigma^*$,
the Parikh map of $L$ is $\#(L) = \{\#(w) \mid w \in L\}$. The language $L$ is semi-linear
if $\#(L)$ is a semi-linear set [76]. $L$ is a semi-linear commutative language if $L$ is
semi-linear and, for all $w_1, w_2$ with $\#(w_1) = \#(w_2)$, $w_1 \in L$ iff $w_2 \in L$. That is, only
the counts information $(|w|_{a_1}, |w|_{a_2}, \ldots, |w|_{a_k})$ is sufficient to decide whether $w \in L$. For instance, 
$\{w : |w|_a - |w|_b > 2|w|_c \land |w|_a < 5|w|_c\}$ is a commutative semi-linear
language over alphabet $\{a, b, c\}$.

Let $x_1, \ldots, x_n$ be $k$ variables over $\mathbb{N}$. An atomic linear constraint is defined as
$a_1 x_1 + \cdots + a_k x_k \sim b$, where $\sim \in \{=, >\}$, $a_1, \cdots, a_k$ and $b$ are integers. The constraint
is called an equation (resp. inequality), if $\sim$ is $=$ (resp. $\geq$). The constraint is made homogeneou
if one makes $b = 0$ in the constraint. A linear constraint is a Boolean
combination of atomic linear constraints (using $\land, \lor, \neg, \rightarrow$). A congruence is in the
form of $x_i \equiv_b c$, where $1 \leq i \leq k$, and $b \neq 0, 0 \leq c < b$. A Presburger formula is a
Boolean combination of atomic linear constraints and congruences using $\lor$ and $\land$. It
is well known that Presburger formulas are closed under negation and quantification
($\neg, \lor$ and $\exists$). A subset $S$ of $\mathbb{N}^k$ is semi-linear iff $S$ is Presburger definable (i.e., there
is a Presburger formula $P$ such that $P(v)$ iff $v \in S$) [46].
Chapter 3

Testability of Oracle Automata\textsuperscript{1}

As we mentioned in Section 1.3.1, eventually the verification problem in (*) will be reduced to a verification problem (checking the validity of a verification condition $P_i$) for each component $C_i$, particularly a testing problem when $C_i$ is a black-box. Thus the first issue we would like to study is “under what conditions we can solve a verification problem for a black-box component through testing”.

In this chapter, we introduce oracle (finite) automata (OFA) as a theoretic tool to study this issue. We show that various verification problems for an oracle (finite) automaton can be reduced to its emptiness problem, which can be solved by testing (querying) the oracles with bounded-length test cases. Then we give some results on the testability of the emptiness problem of oracle (finite) automata with various classes of oracles.

The rest of the chapter is organized as follows. Section 3.1 introduces some definitions used in this chapter. Section 3.2 starts with a formal definition of oracle finite automata and a naive algorithm for testing its emptiness. Then in section 3.3, we present some results on the testability of the emptiness problem for various classes of oracle finite automata. These results are further extended to oracle Buchi automata in Subsection 3.4. We provide a symbolic algorithm for testing the emptiness of oracle finite/Buchi automata in section 3.5. In section 3.6, we establish the connections between the testability results and some important verification problems for oracle

\textsuperscript{1}The content of this chapter is based upon the joint work with C. Li and Z. Dang in [101].
finite/Buchi automata. Then we summarize this chapter in Section 3.7.

3.1 Definitions

Throughout this chapter, Σ is any fixed alphabet. An oracle $O$, is a language in a class $\Theta$ of languages. The name of “oracle” comes from the fact that we only know that the oracle is an element in the class but we do not know which one it is. However, one may obtain a truth value from a query “$w \in O$?” to the oracle $O$ for a word $w$, where $w$ is called a query string.

A finite automaton (FA) $A$ consists of finitely many transitions, each of which makes the automaton move from one state to another while reading an input symbol (in $\Sigma$). In the description of $A$, we also designate an initial state and a number of accepting states. A sequence of input symbols or an input word $w \in \Sigma^*$ is accepted by $A$ if, from the initial state of $A$, $A$ reaches an accepting state after reading the entire word $w$. As usual, $L(A)$ stands for the language accepted by $A$. In general, a FA is nondeterministic; so we use DFA to denote a deterministic FA. A pushdown automaton (PDA) can be obtained by augmenting an FA with a pushdown stack (without loss of generality, we assume that the stack alphabet is the same as the input alphabet $\Sigma$ and each time, the PDA pushes/pops at most one symbol). Similarly, DPDA is used to denote a deterministic PDA. We further use FA($n$) (resp. DFA($n$), PDA($n$), DPDA($n$)) to denote an FA (resp. DFA, PDA, DPDA) with at most $n \geq 1$ states. In this chapter, these notations of automatata are also abused to represent languages accepted by the automata. For instance, FA($n$) is the class of regular languages (on alphabet $\Sigma$) accepted by finite automata with at most $n$ states.

Next, we recall the definitions of (semi-)linear sets and their connection to counter machines. Let $c$ be a nonnegative integer. A $c$-counter machine is an FA augmented with $c$ counters, each of which can be incremented by 1, decremented by 1, and tested for zero. We assume, w.l.o.g., that each counter can only store a nonnegative integer (since the sign can be stored in the states). Let $r$ be a nonnegative integer and let NCM($c$, $r$) denote the class of $c$-counter machines where each counter is $r$ reversal-bounded [54]; i.e., each counter makes at most $r$ alternations between nondecreasing
and non-increasing modes in any computation. For instance, a counter whose values change according to the pattern 0 1 1 2 3 4 4 3 2 1 0 1 1 0 is 3-reversal, where the reversals are underlined. We use DCM(c,r) to denote the deterministic machines in NCM(c,r). From a result in [54], a semi-linear commutative language $L$ can be recognized by a DCM(c,r) $M$ for some $c$ and $r$ if $M$’s input is equipped with an end marker. In particular, it can be shown from [56] that there is a constant $d$ such that, $L \neq \emptyset$ iff there is a word $|w| \leq d^{crm}$ in $L$, where $m$ is the number of states in $M$. This result remains even when $M$ is nondeterministic. From now on, we use $M$ to characterize $L$ and LIN($n$) to denote those semi-linear commutative languages that can be accepted by a DCM(c,r) with $m$ states, where $n = d^{crm}$. With this definition, when $L \in$ LIN($n$) with $n \geq 1$, we say that $L$, as well as the $M$, has characteristic $n$. We use LIN to denote the class of all semi-linear commutative languages.

### 3.2 Oracle Finite Automata

Recall that $\mathcal{O}$ is a class of languages over alphabet $\Sigma$ and $O$, called an oracle, is a language in $\mathcal{O}$. Formally, $M$, an oracle finite automaton (OFA) with $t$ oracles is a tuple

$$\langle t, \Sigma, S, R, s_{\text{init}}, F \rangle,$$

where $\Sigma$ is the given (input/query tape) alphabet, $S$ is a finite set of states with $s_{\text{init}}$ being the initial state and $F \subseteq S$ being a set of accepting states. $R$ is a (finite) set of transitions, each of which is in one of the following five forms:

1) (a read-input transition) $s \xrightarrow{a} s'$, which makes $M$ move from state $s$ to state $s'$ after reading an input symbol $a$;

2) (a write transition) $s \xrightarrow{\text{write}(i,a)} s'$, which makes $M$ move from state $s$ to state $s'$ after appending a symbol $a$ to the end of the $i$-th query tape;

3) (a positive query transition) $s \xrightarrow{\text{query}(i)} s'$, which makes $M$ move from state $s$ to state $s'$ when querying the $i$-th oracle (with the $i$-th query tape content as the query string) returns a “yes” answer;
4) (a negative query transition) $s \xrightarrow{\neg \text{query}(i)} s'$, which makes $M$ move from state $s$ to state $s'$ when $\text{query}(i)$ returns a “no” answer;

5) (a reset transition) $s \xrightarrow{\text{reset}(i)} s'$, which makes $M$ move from state $s$ to state $s'$ and resets the $i$-th query tape content to be empty;

where $s, s' \in S$, $a \in \Sigma$, and $1 \leq i \leq t$. When $t = 1$, $M$ is called a single OFA. Notice that the syntactical definition of $M$ involves neither any description of $O$ nor $O$. When $M$ is associated with an array $O_1, \cdots, O_t$ of $t$ oracles in $O$, we use $M(O_1, \cdots, O_t)$ to denote the association. The semantics of $M$ is defined as follows. Let $M(O_1, \cdots, O_t)$ be an association. A configuration is a tuple $(s, w_1, \cdots, w_t)$ of a state $s$ and $t$ query tape contents $w_1, \cdots, w_t \in \Sigma^*$. The configuration is initial if the state is the initial state and the query tape contents are all empty. The configuration is accepting if $s$ is an accepting state. A one-step transition between two configurations is written as $(s, w_1, \cdots, w_t) \xrightarrow{\alpha} (s', w'_1, \cdots, w'_t)$ when one of the following conditions is satisfied:

- $\alpha$ is $a$, $s \xrightarrow{a} s'$ is a read-input transition in $R$, and each $w'_j = w_j$;
- $\alpha$ is $\text{write}(i, a)$, $s \xrightarrow{\text{write}(i, a)} s'$ is a write transition in $R$, and for each $j \neq i$, $w'_j = w_j$ and $w'_i = wa$;
- $\alpha$ is $\text{query}(i)$, $s \xrightarrow{\text{query}(i)} s'$ is a positive query transition in $R$, query string $w_i$ is in $O_i$, and each $w'_j = w_j$;
- $\alpha$ is $\neg \text{query}(i)$, $s \xrightarrow{\neg \text{query}(i)} s'$ is a negative query transition in $R$, query string $w_i$ is not in $O_i$, or each $w'_j = w_j$;
- $\alpha$ is $\text{reset}(i)$, $s \xrightarrow{\text{reset}(i)} s'$ is a reset transition in $R$ and, for each $j \neq i$ $w'_j = w_j$, and $w'_i = \Lambda$ (the empty string).

A run of $M(O_1, \cdots, O_t)$ is a sequence

$$C_0 \xrightarrow{\alpha_1} C_1 \cdots C_{n-1} \xrightarrow{\alpha_n} C_n,$$

such that
CHAPTER 3. TESTABILITY OF ORACLE AUTOMATA

- for each $j < n$, $C_{j-1} \stackrel{\alpha_j}{\rightarrow} C_j$ is a one-step transition, and

- $C_0$ is the initial configuration.

The run is an accepting run if $C_n$ is an accepting configuration. Let $w$ be the result of deleting elements not in $\Sigma$ from the sequence $\alpha_1 \cdots \alpha_n$. Then we say that the run in (3.2) is a run on input word $w$. A word $w$ is accepted by $M(O_1, \ldots, O_t)$ if there is an accepting run on $w$. The language accepted by $M(O_1, \ldots, O_t)$, written $L(M(O_1, \ldots, O_t))$, is the set of all words accepted by $M(O_1, \ldots, O_t)$. Obviously, when associated with a different array of oracles, a query may return a different result and hence $M$ may behave differently. Therefore, $M$ can be thought of a template with $t$ places to be filled in with oracles. To emphasize the fact that oracles are drawn from $\Omega$, we sometimes use $M^\Omega$ to denote the oracle finite automaton $M$ and further use $M^\Omega(O_1, \ldots, O_t)$ to denote the specific association of the oracles $O_1, \ldots, O_t \in \Omega$ with $M$.

Various restrictions can be placed on query behaviors of an oracle finite automaton $M$. In this chapter, we will focus on the following four forms of restrictions. $M$ is a prefix-closed OFA if $M$ is only associated with prefix-closed oracles\footnote{A language on the alphabet is prefix-closed if the following condition is satisfied: for any word $w$, if $w$ is in the language, then so is every prefix of $w$.}. $M$ is a $k$-query OFA if, during any run, the oracles are queried for at most $k$ times. $M$ is a positive OFA if, in $M$, each query must return a “yes” answer (i.e., $M$ does not have negative query transitions). $M$ is a memoryless OFA if for each $i$, the $i$-th query tape content is erased (by a reset$(i)$ transition) immediately after each query query$(i)$. Therefore, during any run of a memoryless OFA, each query string sent to an oracle was “freshly written” since the previous query to the same oracle.

A $B$-bounded testing script $T$ (with $t$ oracles) is a deterministic Turing machine equipped with two tapes:

- the first tape, called the query tape, is a two-way readable and writable Turing tape whose length is $B$, and

- the second tape, called the working tape, is an ordinary unbounded Turing tape,
CHAPTER 3. TESTABILITY OF ORACLE AUTOMATA

and is further augmented with query instructions. Each query instruction allows the script to query an oracle with a query string that is the content of the portion of the query tape between the first cell and the current cell under the query tape head. A state transition is made upon the query result. We assume that $T$ starts with both tapes blank and always halts, when associated with any array of $t$ oracles. $T$ is successful (resp. unsuccessful) on $O_1, \ldots, O_t$ if, when associated with $O_1, \ldots, O_t$, $T$ halts with an accepting (resp. rejecting) state. The name of a “testing script” comes from the fact that, when $T$ runs, the oracles are tested (queried) with query strings not longer than $B$. When $T$ halts, the testing is finished and an answer of either “successful” or “unsuccessful” is given. Of course, the answer may be different when $T$ is associated with another array of oracles.

A testing script is used to solve problems concerning an OFA. Let $X$ be one of the language classes FA, DFA, PDA, DPDA, and LIN defined earlier. We use $OFA^X$ to denote the class of OFAs whose oracles are drawn only from $X$. Let $M^{X(n)}$ be one such automaton in $OFA^X$ with $n \geq 1$. Then a problem of $M$ is a predicate over some oracles $O_1, \ldots, O_t$ in $X(n)$. For instance, the emptiness problem of $M^{X(n)}$ is to decide whether $M^{X(n)}(O_1, \ldots, O_t)$ accepts an empty language. This problem can be characterized by the predicate $\mathbb{P}_{M^{X(n)}}(O_1, \ldots, O_t)$, which is true iff $L(M^{X(n)}(O_1, \ldots, O_t)) = \emptyset$. A problem $\mathbb{P}$ of $M^{X(n)}$ is testable if there is an algorithm such that from the description of $M$ and $n$, one can compute a number $B(M, n)$ and a $B(M, n)$-bounded testing script $T$ satisfying the following condition: for each $O_1, \ldots, O_t \in X(n)$, $\mathbb{P}(O_1, \ldots, O_t)$ is true (resp. false) iff $T$ is successful (resp. unsuccessful) on $O_1, \ldots, O_t$. In this case, we also say that the $\mathbb{P}$ problem of the oracle finite automaton $M^{X(n)}$ is $B(M, n)$-testable. That is, the $\mathbb{P}$ problem of $M^{X(n)}$ can be decided by running the test script which queries the oracle with query strings not longer than $B(M, n)$.

### 3.3 Testing Emptiness for Oracle Finite Automata

In automata theory, the emptiness problem is to decide whether an automaton accepts the empty language. Algorithmic solutions to the emptiness problem for various classes of automata have become a cornerstone in many areas of computer science,
especially in automata-theoretic based model checking technique. In traditional automata theory, the automaton studied in the emptiness problem must be fully specified. However, the oracle finite automata studied in this chapter are only partially specified (due to the existence of the oracles). So, testing the oracles seems an inevitable way to answer the problem. But we first need to know whether the problem is testable.

Notice that an oracle finite automaton $M_X^{(n)}$, when associated with oracles $O_1, \cdots, O_t$, runs on some input, during which the oracles are queried. On a specific run, one may record the maximal length of all query strings sent to the oracles. Assume that the maximal length is uniformly bounded by a number $B$, called a query bound, among all the possible input words, runs, and associations of oracles from $X(n)$. Under this assumption, checking whether $L(M_X^{(n)}(O_1, \cdots, O_t) = \emptyset$ becomes easier. This is because, for the purpose of emptiness, one can make each oracle to be finite (the number of elements is bounded by $|\Sigma|^B$) by dropping any word longer than $B$ from the oracle. In this way, the finite oracle can be “recovered” through a finite number of queries. Even though the assumption in general does not hold, one can effectively build an approximated version $M_X^{(n)}_B$ from $M_X^{(n)}$ that satisfies the assumption for any number $B$, by forcing $M_X^{(n)}$ to crash whenever it tries to query the oracle with a query string longer than $B$. Then we refine the definition of query bound: $B$ is a query bound of $M_X^{(n)}$ if, for any $O_1, \cdots, O_t \in X(n)$,

\[
L(M_X^{(n)}(O_1, \cdots, O_t) = \emptyset \text{ iff } L(M_X^{(n)}_B(O_1, \cdots, O_t) = \emptyset.
\]

Once the query bound is identified, a $B$-bounded testing script $T$ can be easily constructed to answer the emptiness of $M_X^{(n)}$.

1) For each oracle, $T$ enumerates each string not longer than $B$, queries the oracle with the string, and stores the result on the working tape.

2) On the working tape, $T$ also constructs a finite automaton (without any oracles) that simulates $M_X^{(n)}_B$ where each query is answered by retrieving the stored results.
3) $T$ returns “successful” or “unsuccessful” depending on whether the finite automaton accepts an empty language (this can be decided by running a standard algorithm on the finite automaton).

Later in the chapter, we will show a more efficient construction. Hence, in proving that the class $\text{OFA}^X$ is testable, we only need to demonstrate that a computable query bound exists for every $M^X$ in $\text{OFA}^X$. This is the fundamental approach we will use to study some testable classes of oracle finite automata.

Studies on black-box testing [67] have shown that the structure of a finite automaton with $n$ states can be completed recovered by test sequences with length not longer that a bound $BT(n)$. This result can be immediately used to establish the testability of OFAs with regular oracles in $\text{FA}(n)$. However, there are reasons that new techniques are needed. First, for emptiness testing, one does not need to recover the complete information of the oracle, and a smaller query bound than $BT(n)$ may exist. Second, as shown below, complete information is not recoverable for some practically useful but irregular oracles, e.g., $\text{PDA}(n)$. That is, $BT(n)$ is not computable (from $n$) for oracles in $\text{PDA}(n)$. In other words, context-free languages are not black-box testable.

**Theorem 3.1** Context-free languages are not black-box testable.

See Appendix A.1.1 for the proof of this theorem.

In the rest of this section, we will present some results concerning the testability of the emptiness problem for various classes of OFAs.

### 3.3.1 The Testability of OFA With Regular Oracles

Recall that an oracle finite automaton $M$ is associated with an array of $t$ oracles. Let $|M|$ denote the number of states in $M$. We start with the case when $M$’s oracles are regular.

**Theorem 3.2** (a) The emptiness problem for oracle finite automata $M^\text{DFA}(n)$ is $O(n^t \cdot |M|)$-testable. (b) The emptiness problem for oracle finite automata $M^\text{FA}(n)$ is $O(2^{nt} \cdot |M|)$-testable.
CHAPTER 3. TESTABILITY OF ORACLE AUTOMATA

See Appendix A.1.2 for the proof of this theorem.

3.3.2 The Testability of OFA With Context-free Oracles

We now study the case when an OFA’s oracles are drawn from context-free languages. As shown in Theorem 3.1, context-free languages are not testable. But there do exist some special conditions under which the emptiness problem for $M$ is testable. The proof of the following Theorem 3.3(a) uses a reduction to the halting problem of two-counter machines. For the testable cases, Theorem 3.3(b) involves PDA constructions.

**Theorem 3.3** (a) The emptiness problem for oracle finite automata in $\text{OFA}^{\text{PDA}}$ is not testable. The result remains in each of the following restricted cases:

(a.1) the automata are 1-query and single,

(a.2) the automata are 2-query, positive, and single,

(a.3) the automata are 2-query, positive, and in $\text{OFA}^{\text{DPDA}}$.

(b) The emptiness problem for oracle finite automata $M^{\text{PDA}(n)}$ is $2^{O(|M|^2n^2|\Sigma|)}$-testable, under each of the following conditions:

(b.1) $M^{\text{PDA}(n)}$ is positive, single, and prefix-closed,

(b.2) $M^{\text{PDA}(n)}$ is positive, single, and 1-query,

(b.3) $M^{\text{PDA}(n)}$ is positive, single, and memoryless,

(b.4) $M^{\text{PDA}(n)}$ is $M^{\text{DPDA}(n)}$ and single.

See Appendix A.1.3 for the proof of this theorem.

3.3.3 The Testability of OFA With Semi-linear Oracles

Finally, we consider the case when an OFA’s oracles are drawn from semi-linear commutative languages. As we are going to show, even though, in general, OFAs with such oracles are in general not testable for emptiness, under some restrictions, the
problem becomes testable. The proof of the following Theorem 3.4(a) is a complex reduction to the halting problem of two-counter machines. In showing Theorem 3.4(b), properties over reversal-bounded NCMS are used.

Theorem 3.4 (a) The emptiness problem for oracle finite automata in OFA$$^{\text{LIN}}$$ is not testable. The result remains in each of the following restricted cases:

(a.1) the automata are single and positive,

(a.2) the automata are memoryless, positive, and have two oracles (i.e., $$t = 2$$).

(b) The emptiness problem for oracle finite automata $$M^{\text{LIN}(n)}$$ is testable, under each of the following conditions:

(b.1) $$M^{\text{LIN}(n)}$$ is $$k$$-query. In this case, it is $$O(n^{k|M|^k})$$-testable,

(b.2) $$M^{\text{LIN}(n)}$$ is prefix-closed,

(b.3) $$M^{\text{LIN}(n)}$$ is memoryless and single. In this case, it is $$O(n^{|M|})$$-testable.

See Appendix A.1.4 for the proof of this theorem.

3.4 Testing Emptiness for Oracle Buchi Automata

Syntactically, an oracle Buchi automaton ($$\omega$$-OFA) $$M_\omega$$ is an oracle finite automaton $$M$$ in (3.1). The difference is that, $$M_\omega$$ accepts only $$\omega$$-runs (i.e., infinite runs). We write $$M_\omega^O$$ for $$M_\omega$$ when its oracles are drawn from $$\mathcal{O}$$. Let $$M_\omega^O(O_1, \ldots, O_t)$$ be an association of $$M_\omega$$ with oracles $$O_1, \ldots, O_t$$ in $$\mathcal{O}$$. An $$\omega$$-run of $$M_\omega^O(O_1, \ldots, O_t)$$ is an infinite sequence

$$C_0 \xrightarrow{\alpha_1} C_1 \cdots C_{n-1} \xrightarrow{\alpha_n} C_n \cdots, \quad (3.3)$$

such that each prefix $$C_0 \xrightarrow{\alpha_1} C_1 \cdots C_{n-1} \xrightarrow{\alpha_n} C_n$$ is a run of $$M_\omega^O(O_1, \ldots, O_t)$$, and for all $$m$$ there is an $$n > m$$ with $$\alpha_n \in \Sigma$$. This latter requirement ensures that an $$\omega$$-run reads an infinite number of input symbols. The $$\omega$$-run is accepting if some accepting state in $$F$$ appears infinitely often on the run. An $$\omega$$-word $$\tau$$ is accepted by $$M_\omega^O(O_1, \ldots, O_t)$$ if there is an accepting run in the form of (3.3) such that the word $$w_n$$ (the result of
CHAPTER 3. TESTABILITY OF ORACLE AUTOMATA

deleting elements not in $\Sigma$ from the sequence $\alpha_1 \cdots \alpha_n$ is a prefix of $\tau$, for each $n$. We use $L^\omega(M^\omega_\omega(O_1, \cdots, O_i))$ to denote the $\omega$-language accepted by $M^\omega_\omega(O_1, \cdots, O_i)$.

Completely analogous to oracle finite automata, we use $\omega$-OFA$^X$ to denote the set of all $\omega$-OFA $M^X_\omega(n)$, for $X \in \{\text{FA, DFA, PDA, DPDA, LIN}\}$. We also follow a similar definition for prefix-closed, $k$-query, memoryless, and positive $\omega$-OFAs.

The emptiness problem (for oracle Buchi automata) is to decide whether $M^X_\omega(n)(O_1, \cdots, O_i)$ accepts an empty $\omega$-language. For each class $C$ of oracle finite automata considered in Theorems 3.3(a) and 3.4(a) whose emptiness is not testable, one can easily conclude that the emptiness problem for its corresponding class of oracle Buchi automata is not testable either.

**Theorem 3.5** (a) The emptiness problem for oracle Buchi automata in $\omega$-OFA$^{\text{PDA}}$ is not testable. (b) The emptiness problem for oracle Buchi automata in $\omega$-OFA$^{\text{LIN}}$ is not testable. The two results remain even when each of the restrictions stated in Theorem 3.3(a) is applied.

**Proof.** This is because from an oracle finite automaton $M$, one can build an oracle Buchi automaton $M'$ as follows. $M'$ behaves in the exactly same way as $M$, except that when $M$ enters an accepting state, $M'$ nondeterministically enters a special state (that is the only accepting state of $M'$) and keeps staying in the state forever. Clearly, on associating with any oracles, $M$ accepts an empty language iff $M'$ accepts an empty $\omega$-language. Notice that if $M$ belongs to the class $C$ of oracle finite automata mentioned earlier, $M'$ belongs to the same class of oracle Buchi automata too.

However, the emptiness problem for some restricted $M^X_\omega(n)$ is testable; i.e., one can compute a $B(M, n)$-bounded testing script and, after running the script, the emptiness can be decided. The basic technique in showing testability is to reduce the emptiness problem of an oracle Buchi automaton in a class to the emptiness problem of an oracle finite automaton in the same class through loop analysis. Although loop analysis is a general technique, such a reduction does not always exist. For instance, currently, we do not know whether a positive, single and prefix-closed $M^\text{PDA}(n)$ is testable or not for emptiness. However, according to Theorem 3.3 (b.1), a positive, single and prefix-closed $M^\text{PDA}(n)$ is testable for emptiness. Next, let’s look at some
“testable” results for regular, context-free, and semi-linear oracle Buchi automata respectively.

### 3.4.1 Testability of ω-OFAWith Regular Oracles

**Theorem 3.6** The emptiness problem for oracle Buchi automata $M_{\omega}^{\text{DFA}(n)}$ is $O(n^{2l} \cdot |M|)$-testable. The emptiness problem for oracle Buchi automata $M_{\omega}^{\text{FA}(n)}$ is $O(2^{2nt} \cdot |M|)$-testable.

See Appendix A.1.5 for the proof of this theorem.

### 3.4.2 Testability of ω-OFAWith Context-free Oracles

**Theorem 3.7** The emptiness problem for oracle Buchi automata $M_{\omega}^{\text{PDA}(n)}$ is $2^{O(|M|^2 \cdot n^2 \cdot |\Sigma|)}$-testable, under each of the following conditions:

1. $M_{\omega}^{\text{PDA}(n)}$ is positive, single and 1-query.
2. $M_{\omega}^{\text{PDA}(n)}$ is positive, single and memoryless.
3. $M_{\omega}^{\text{PDA}(n)}$ is $M_{\omega}^{\text{DPDA}(n)}$ and single.

See Appendix A.1.6 for the proof of this theorem.

### 3.4.3 Testability of ω-OFA With Semi-linear Oracles

**Theorem 3.8** The emptiness problem for oracle Buchi automata $M_{\omega}^{\text{LIN}(n)}$ is testable, under each of the following conditions:

1. $M_{\omega}^{\text{LIN}(n)}$ is $k$-query. In this case, it is $O(n^{k \cdot |M|^k})$-testable.
2. $M_{\omega}^{\text{LIN}(n)}$ is prefix-closed.
3. $M_{\omega}^{\text{LIN}(n)}$ is memoryless and single. In this case, it is $O(n^{|M|})$-testable.

See Appendix A.1.4 for the proof of this theorem.
3.5 A Dynamic Testing Algorithm

The solution to the emptiness problem for $\text{OFA}_{1}^{\text{DFA}(m)}$ and $\text{OFA}_{\omega}^{\text{DFA}(m)}$ in Section 3.3 involves pre-querying the oracles indiscriminately with all possible strings with length shorter than $2m - 1$. This would be extremely inefficient in practice, considering the fact that there are an exponential number $(|\Sigma|^{2m-1})$ of such strings.

In this subsection, we introduce a more efficient algorithm to solve the emptiness problem for $\text{OFA}_{1}^{\text{DFA}(m)}$ and $\text{OFA}_{\omega}^{\text{DFA}(m)}$. The new algorithm only queries the oracles with query strings that could be “generated” by the OFAs. Since each query to an oracle can also be viewed as a test over the oracle where the query string is a test-case, this algorithm can also be viewed as a dynamic testing process where test-cases are generated on-the-fly.

Suppose that $M^{\text{DFA}(m)}$ is an OFA as defined in (3.1). Without loss of generality, we assume that $M$ is associated with only one oracle (i.e., $k = 1$); generalization to multiple oracles is straightforward. Consequently, there will be only one query tape in $M$. Then we write instructions $\text{reset}(i)$, $\text{write}(i,a)$, $\text{query}(i)$, and $\neg \text{query}(i)$ as $\text{reset}$, $\text{write}(a)$, $\text{query}$, and $\neg \text{query}$, respectively. A transition relation $r$ is a subset of $S \times S$, where $S$ is the state set of $M$. We use $r_1 \circ r_2$ to denote the relation obtained from composing relation $r_1$ with relation $r_2$, $\text{Intersect}$ to denote the intersection operator, and $\text{TransClosure}(r)$ to denote the transitive closure of a relation $r$, respectively. We also use $\text{Empty}(r)$ to test whether a relation $r$ is empty.

Then, from the definition of $M$, we define the following transition relations:

- $r_{\text{input}} = \{ (s, s') : \exists a, s \xrightarrow{a} s' \in R \}$,
- $r_{\text{reset}} = \{ (s, s') : s \xrightarrow{\text{reset}} s' \in R \}$,
- $r_{\text{write}(a)} = \{ (s, s') : s \xrightarrow{\text{write}(a)} s' \in R \}$,
- $r_{\text{query}} = \{ (s, s') : s \xrightarrow{\text{query}} s' \in R \}$,
- $r_{\neg \text{query}} = \{ (s, s') : s \xrightarrow{\neg \text{query}} s' \in R \}$.

We first present the algorithm, $\text{TestEmptiness}(B)$, for testing the emptiness of $M^{\text{DFA}(m)}$, where the query strings are not longer than $B$. Later, we will describe an algorithm for testing the emptiness of $M_\omega^{\text{DFA}(m)}$.

Algorithm $\text{TestEmptiness}(B)$
1: \( l := 0; \)
2: \( \Theta := \{(s, s) : s \in S, \lambda\}; \)
3: \( E = \{s_{\text{init}}, s_{\text{init}}\}; \)
4: \( \Theta' := \Theta; \)
5: \textbf{for each} \( (r, w) \) in \( \Theta \) with \( |w| = l \)
6: \( r' := r \circ \text{TransClosure}(r_{\text{input}}); \)
7: \textbf{if NotEmpty}(\( r' \circ r_{\text{query}} \)) or \textbf{NotEmpty}(\( r' \circ r_{\text{query}} \))
8: \text{query the oracle with query string} \( w; \)
9: \textbf{if} the query returns yes
10: \( r' := r \circ \text{TransClosure}(r_{\text{input}} \cup r_{\text{query}}); \)
11: \textbf{if} the query returns no
12: \( r' := r \circ \text{TransClosure}(r_{\text{input}} \cup r_{\text{query}}); \)
13: \text{replace the entry} \( (r, w) \) in \( \Theta \) with \( (r', w); \)
14: \( r'' := r' \circ r_{\text{reset}}; \)
15: \textbf{if NotEmpty}(\( r'' \))
16: \( E := \text{TransClosure}(E \cup r'' \cup r_{\text{input}}); \)
17: \textbf{for each} \( a \in \Sigma \)
18: \( r'' := r' \circ r_{\text{write}}(a); \)
19: \textbf{if NotEmpty}(\( r'' \))
20: \text{add} \( (r'', wa) \) to \( \Theta; \)
21: \( l := l + 1; \)
22: \textbf{for each} \( (r, w) \) in \( \Theta \)
23: \( r' := \text{Intersect}(E \circ r, \{s_{\text{init}}\} \times F); \)
24: \textbf{if NotEmpty}(\( r' \))
25: \textbf{return} \text{“unsuccessful”;}
26: \textbf{if} \( \Theta' \) and \( \Theta \) are equal or \( l > B \)
27: \textbf{return} \text{“successful”;}
28: \textbf{goto} 4;

The \textbf{TestEmptiness} algorithm works as follows. We maintain a finite set \( \Theta \) of pairs of a relation \( r \) and a word \( w \). For two states \( s \) and \( s' \), \( \langle s, s' \rangle \) is in \( r \) iff, starting from state \( s \) and with empty query tape, there is some input word such that state
s' is reached (after running $M$ on the input) with the query tape content $w$, during which no reset occurs. The algorithm also maintains a relation $E$: for two states $s$ and $s'$, $\langle s, s' \rangle$ is in $E$ iff, starting from state $s$ and with empty query tape, there is a run of $M$ that brings to state $s'$ and also with empty query tape. After initializing $\Theta$ and $E$, the entire algorithm works as a loop from statement 4 to statement 28 and back. In the $l$-th round ($l$ starts with 0), the algorithm updates an element $(r, w)$ in $\Theta$ with $|w| = l$, realized by changing $w$ into $wa$ (i.e., $\text{write}(a)$ on the query tape). However, transitions like reading input symbols and querying the oracle can happen before this write, and obviously, the query result matters. This is shown in statements 6–13 where an updated version $(r', w)$ of $(r, w)$ is replaced in $\Theta$ (i.e., statement 13). Notice that, a query is performed when necessary shown in statement 8. Then, $\text{write}(a)$ is implemented in statements 17–20 to add longer query strings $wa$ into $\Theta$. Clearly, $w$ can also be changed into an empty string through a reset, which causes an update on $E$ (recalling the meaning of $E$ mentioned earlier) shown in statements 14–16. Finally in the round, statements 22–27 are used to check whether $M$ accepts an empty language. Clearly, according to the semantics of $\Theta$, if it has a $(r, w)$ where $r$ contains the pair of the initial state and an accepting state, then obviously $M$ accepts a nonempty language — an “unsuccessful” is returned as the result of statements 23–25. If the set $\Theta$ does not change in the round (so further rounds are not necessary) or the level $l$ is higher than the given bound $B$, then $M$ must accept an empty language (i.e., returns “successful” as in statement 27).

It’s not hard to show that the above algorithm is both sound and complete, if one chooses a bound $B \geq m \cdot |M|$. It shall also be noted that the algorithm can be implemented symbolically. This is because a relation can be represented symbolically as a Boolean formula whose satisfying assignments can be further encoded with a BDD [18]. Operations like $\text{TransClosure}$, $\text{Intersect}$, $\circ$, $\text{Empty}$ are all standard operations in existing BDD libraries [85].

We can construct another algorithm $\omega$-$\text{TestEmptiness}$ for testing the emptiness problem of $M^\omega_{\text{DFA}(m)}$, using $\text{TestEmptiness}$. This algorithm works as follows. It first constructs an OFA $M'$ from the $\omega$-OFA $M$ that works as follows. $M'$ first guesses an accepting state in $F$ (the set of accepting states in the $\omega$-OFA $M$) and faithfully
simulates $M$. $M'$ accepts an input word if $M$ enters the guessed accepting state for $m$ times. Clearly, $M'$ is an OFA (instead of an $\omega$-OFA), and it is not hard to show that $M'$ accepts an empty language iff the $\omega$-OFA $M$ does. Then $\omega$-TestEmptiness calls algorithm TestEmptiness$(B)$ running on $M'$ with $B \geq |F| \cdot |M| \cdot m^2$. One can also show that the algorithm $\omega$-TestEmptiness is both sound and complete.

### 3.6 Some Verification Problems

As mentioned at the beginning of this chapter, the oracle automata proposed in this work are intended as a theoretic tool for studying how far we can go on verifying systems with black-box components. In this section, we show that the testability result of oracle automata can be immediately used to solve the reachability, safety, and LTL model checking problems for systems with black-box components.

Suppose that $Sys = \langle M, X_1, \ldots, X_k \rangle$ is defined in (2.1) where each $X_i$ for $1 \leq i \leq k$ is a testable, black-box component. Let $m = \max_{1 \leq i \leq k} m_i$ where each $m_i$ is an upper bound for the number of states in $X_i$ for all $1 \leq i \leq k$.

#### 3.6.1 The Reachability Problem

The reachability problem is to decide: starting from its initial state, whether $Sys$ can reach some state in a given set $Bad$ of states; i.e., whether $Bad$ is reachable in $Sys$ (in practice, $Bad$ may specify some “bad” states that are not supposed to be reached).

To solve this problem, we first construct an OFA, $M_{OFA}^{DFA(m)}(O_1, \ldots, O_k)$ in (3.1) from the definition of $Sys$ as follows.

1. For each $1 \leq i \leq k$, let oracle $O_i$ denote the set of behaviors of the unspecified component $X_i$ (remember that an oracle is a language without detailed definition).

2. Let $M_{OFA}$ have the same set of states and same initial state as $M$.

3. Let $M_{OFA}$’s $\Sigma$ be the union of all $\nabla_i$’s in $Sys$.

4. Let $Bad$ be $M_{OFA}$’s accepting states.
5. For each transition \((s, a, s')\) in \(M\) with \(a \in \Lambda_M \cup \Sigma_{ENV}\) (i.e., \(a\) is either an internal action of \(M\) or it's in the interface of the environment), add a transition \((s \xrightarrow{a} s')\) to \(M_{OFA}\).

6. For each output transition \((s, a, s')\) in \(M\) with \(a \in \Gamma_i\) for some \(1 \leq i \leq k\) (i.e., \(a\) is an input action of \(X_i\)), add a transition \((s \xrightarrow{\text{write}(a, i)} s')\) to \(M_{OFA}\).

7. For each input transition \((s, a, s')\) in \(M\) with \(a \in \Pi_i\) for some \(1 \leq i \leq k\) (i.e., \(a\) is an output action of \(X_i\)), add a new state \(s''\), as well as two transitions \((s \xrightarrow{\text{write}(a, i)} s'')\) and \((s'' \xrightarrow{\text{query}(i)} s')\) to \(M_{OFA}\).

![Figure 3.1: A Simple Communication System](image)

For instance, from the system depicted in Figure 3.6.1, we can construct an OFA as shown in Figure 3.6.1 (since this OFA has only one query tape, in Figure 3.6.1, we write instructions \text{write}(i, a) and \text{query}(i) as \text{write}(a) and \text{query} respectively).

Now it is easy to see that \text{Bad} is not reachable in the system \(Sys\) iff the constructed \(M_{OFA}\) accepts a nonempty language. Then we have,

**Theorem 3.9** The reachability problem for the system \(Sys\) is testable.

### 3.6.2 The Safety Problem

The safety problem is to decide whether every behavior of the system \(Sys\) is contained in a given regular language \(R\). Assume that the complement of \(R\) can be accepted by a finite automaton \(M_R\) and let \(\tilde{M}\) be the Cartesian product of \(M_R\) and \(M\). Notice that each state in \(\tilde{M}\) is a pair of states in \(M_R\) and \(M\) respectively and \(\tilde{M}\) totally
CHAPTER 3. TESTABILITY OF ORACLE AUTOMATA

![Diagram of an Oracle Finite Automaton]

Figure 3.2: An Oracle Finite Automaton

has \(|M_R| \cdot |M|\) number of states; i.e. \(|\bar{M}| = |M_R| \cdot |M|\). Let \(F\) denote the set of states in \(\bar{M}\), each of which contains a final state of \(M_R\). Similar to the construction in the reachability problem (except that \(F\) would be the OFA’s accepting states), we can construct an OFA, \(\bar{M}_{OFA}^{DFA(m)}(O_1, \ldots, O_k)\) from this \(\bar{M}\) as well as the unspecified components \(X_i, 1 \leq i \leq k\). Then it shall be noticed that the safety problem is true iff the constructed \(\bar{M}_{OFA}^{DFA(m)}\) accepts an empty language. Hence we have,

**Theorem 3.10** The safety problem for the system \(Sys\) is testable.

### 3.6.3 The LTL Model Checking problem

Next, we consider a verification problem concerning \(\omega\)-behaviors of the system \(Sys\); i.e., the LTL model checking problem.

The **LTL model checking** problem is to decide whether every \(\omega\)-behavior of the system \(Sys\) satisfies a given LTL formula \(f\). Similar to the standard LTL model checking approach [90], we define \(\bar{M}\) to be the Cartesian product of \(M\) and the Buchi automaton that accepts \([-f]\). Similar as before, we construct an \(\omega\)-OFA, \(\bar{M}_{\omega}^{DFA(m)}(O_1, \ldots, O_k)\) from this \(\bar{M}\) as well as the unspecified components \(X_i\) s. Observe that the LTL model checking problem is equivalent to checking the emptiness of \(\bar{M}_{\omega}^{DFA(m)}(O_1, \ldots, O_i)\). Hence, we have the following result:
Theorem 3.11 The LTL model checking problem for the system Sys is testable.

Note that in the above constructions, the oracles actually characterize the behaviors of the unspecified components. Therefore, when we apply the TestEmptiness algorithm to the constructed OFAs (OFA\_w's), line 8 of TestEmptiness, i.e., “query the oracle with string w” should be replaced with BBTest(X, w). That is, the model checking problem for the systems Sys are finally reduced to testing the black-box components in Sys.

Remark. As we have seen from the above reductions from the model checking problems on Sys to the emptiness problems for the constructed M\_OFA's, there are no reset and negative query transitions in M\_OFA. This implies that the reduction and the algorithms still work when we understand each \( m_i \), instead of being the number of states in component \( X_i \), to be the number of states in a nondeterministic finite automaton that accepts the behaviors of \( X_i \). This will greatly bring down the bound \( B \) for query strings for the algorithms TestEmptiness and \( \omega\text{-TestEmptiness} \). Also, the above argument still applies, if we further allow “reset” to be an ordinary input symbol of \( X_i \), i.e., “reset” can appear on a transition in Sys. Clearly, the transition containing a “reset” in \( sys \) corresponds to a reset transition in the OFA to which \( Sys \) is reduced.

Example 3.12 For instance, still consider the data acquisition system depicted in Figure 2.1, which periodically transmits the reading of a photo sensor via some underlying communication network. We want to verify that the following property is satisfied by the system \( Sys \):

- Along all observable behaviors of \( Sys \), no two data actions can occur without a send action occurring in between; i.e., \( \neg E F \) data \( X(\neg\text{send U data}). \)

It is easy to see that our models in (2.1) and (2.1) are very suitable to specify the design of such TinyOS applications. Thus, this model checking problems can be readily reduced to the emptiness problem of an \( \omega \)-OFA constructed from the LTS model. And the emptiness problem of the \( \omega \)-OFA can be solved by querying (testing) the oracles (unspecified components) with strings of bounded length.
3.7 Summary

In this chapter, we mainly addressed the general problem of verifying systems interacting with an unknown environment or some black-box components. We introduced oracle automata (which are finite/Buchi automata augmented with oracles in some classes of formal languages) as a formalism to model such systems and showed that some important verification problems (such as reachability, safety, and LTL model checking problems etc.) can be reduced to testing the emptiness of oracle automata. Then, we devoted the main part of this chapter to establishing the testability results of the verification problems for various classes of oracle automata. We also gave a symbolic algorithm to perform bounded-testing for the emptiness problem.

Oracle (Turing) machines are a classic concept in the theory of computation, and have been quite useful in studying, e.g., relativized complexity classes [8]. However, as far as we know, studying oracle finite automata in the context of model checking is new. In the future, there are several possible directions for extending this study. For instance, one may investigate oracle infinite-state automata instead of oracle finite automata. That is to study testability of designs with unbounded variables that interact with partially specified environments. One other possible work is to find ways to obtain a smaller query bound using, e.g., structural information of the transition graph of an oracle finite automaton. We will also look at the possibility of hooking up a real-world tester with a component-based design and performing model checking through testing, using some of the algorithms in this chapter. The (worst-case) query bounds obtained in the chapter are large. However, the worst-cases may not show up in a specific application. In particular, even an internally complex environment may only have a very simple pattern of observable (by the design or the OFA) behavior, which will significantly bring down the query bounds.
Chapter 4

Model Checking Systems With One Black-box\textsuperscript{1}

As mentioned earlier in Chapter 1, one of the most difficult challenges to the the quality assurance of component-based systems is the existence of black-box components. Having shown under what conditions we can solve a verification problem for a black-box component through testing in the previous chapter. In this chapter, we study the specific model checking problems for systems with only one black-box component:

- given a component-based system $Sys = \langle M, X \rangle$ where $X$ is a black-box component, a state $s \in \text{Init}_M$, and a temporal formula $f$;
- check whether $\langle M, X \rangle, s \models f$; i.e., whether $f$ is true when $M$ is in the state $s$.

We present a set of new algorithms, called model checking driven black-box testing, which combine model checking techniques and black-box testing techniques to solve the problem.

The idea of our algorithms is to use a model checking based technique that automatically derives from the rest of the system (i.e., $M$) a verification condition over the black-box component $X$. This verification condition guarantees that the system $Sys$ satisfies the requirement $f$ iff the condition is satisfied by the black-box component $X$.

\textsuperscript{1}The content of this chapter is based upon the work in [97] and the joint work with Z. Dang in [99].
CHAPTER 4. MODEL CHECKING SYSTEMS WITH ONE BLACK-BOX

In the algorithms, the verification condition is represented as communication graphs (for LTL) or witness graphs (for CTL) and the validity of the condition is checked through testing the black-box component. Test-cases are generated by a bounded and nested depth-first search procedure over the communication graphs (for LTL) or witness graphs (for CTL). Our algorithms are both sound and complete.

This chapter is organized as follows. Section 4.1 contains some definitions. In Section 4.2 and Section 4.3, we present algorithms for LTL and CTL model checking driven black-box testing, respectively. Section 4.4 illustrates the algorithms through some examples and Section 4.5 ends this chapter with discussions on issues to be addressed in the future.

4.1 Definitions

The system studied in this chapter is defined as in (2.1), but contains only one black-box component; i.e., $Sys = \langle M, X \rangle$ where $M = \langle S_M, Init_M, \nabla_M, R_M \rangle$ with $|S| = n$ and $X$ is a black-box component with $\nabla_X$ as its interface and $m$ being an upper bound for its state space. We further partition the transition relation $R_M$ of $M$ into two parts:

- $R_{env} = \{(s, a, s')|(s, a, s') \in R_M \land (a \in \Lambda_M \lor a \in \Sigma_{ENV})\}$; i.e., set of transitions involving internal actions or communications with the environment;

- $R_{comm} = R_M \setminus R_{env}$.

We also define some convenience relations:

- $R_{env}^s := \{(s, s') | \exists a \in \nabla : (s, a, s') \in R_{env}\}$,

- $R_M^s := \{(s, s') | \exists a \in \nabla : (s, a, s') \in R_M\}$,

- $R_{env}^T := TransitiveClosure(R_{env}^s)$,

- $R_M^T := TransitiveClosure(R_M^s)$,

where $TransitiveClosure$ is used to compute the transitive closure of a given relation.
4.2 LTL Model Checking Driven Black-box Testing

In this section, we introduce algorithms for LTL model checking driven black-box testing for systems with one black-box component: $Sys = \langle M, X \rangle$. We first show how to solve a liveness analysis problem. Then, we discuss the general LTL model checking problem.

4.2.1 Liveness Analysis

The liveness analysis problem (also called the infinite-often problem) is to check: starting from some initial state $s_0 \in Init_M$, whether $M$ can reach a given state $s_f$ for infinitely many times.

When $M$ has no communications with the black-box component $X$, solving the problem is equivalent to finding a path $p$ that runs from $s_0$ to $s_f$ and a loop $C$ that passes $s_f$. However, as far as communications are involved, the problem gets more complicated. The existence of the path $p$ does not ensure that $M$ can indeed reach $s_f$ from $s_0$ (e.g., communications with $X$ may never allow the system to take the necessary transitions to reach $s_f$). Moreover, the existence of the loop $C$ does not guarantee that the system can run along $C$ forever either (e.g., after running along $C$ for three rounds, the system may be forced to leave $C$ by the communications with $X$).

We solve this infinite-often problem in three steps. First, we look at whether a definite answer to the problem is possible. If we can find a path from $s_0$ to $s_f$ and a loop from $s_f$ to $s_f$ that involve only environment transitions, then the original problem (i.e., the infinite-often problem) is definitely true. If such a path and a loop, no matter what transitions they may involve, do not exist at all, then the original problem is definitely false. If no definite answer is possible, we construct a directed graph $G$ and use it to generate test-cases for the black-box component $X$. The graph $G$, called a communication graph, is a subgraph of $M$, represents all paths and loops in $M$ that could witness the truth of the problem (i.e., paths that run from $s_0$ to $s_f$
and loops that pass \( s_f \). The graph \( G \) is defined as a pair \( \langle N, E \rangle \), where \( N \) is a set of nodes and \( E \subseteq N \times \nabla_M \cup \{\epsilon\} \times N \) is a set of edges. Each edge of \( G \) is annotated either by an action \( a \in \Sigma_X \) that denotes a communication of \( M \) with \( X \), or has no annotation (i.e., annotated by \( \epsilon \)). We construct \( G \) as follows.

- Add one node to \( G \) for each state in \( M \) that is involved in some path between \( s_0 \) and \( s_f \) or in a loop that passes \( s_f \);
- Add one edge between two nodes in \( N \) if \( M \) has a transition between two states corresponding to the two nodes respectively. If the transition involves a communication with \( X \), then annotate the edge with the action.

It is easy to see that the liveness analysis problem is true if and only if the truth is witnessed by a path in \( G \).

Therefore, the last step is to check whether \( G \) has a path along which the system can reach \( s_f \) from \( s_0 \) first and then reach \( s_f \) for infinitely many times. More details of this step are addressed in the next subsection.

**procedure** CheckIO(\( \langle M, X \rangle, s_0, s_f \))

\[
N := \emptyset; \quad E := \emptyset;
\]

if \((s_0, s_f) \in R^T_{\text{env}} \land (s_f, s_f) \in R^T_{\text{env}}\) then

return “Yes”;

else if \((s_0, s_f) \notin R^T_M \land (s_f, s_f) \notin R^T_M\) then

return “No”;

end if

\[
N := \{s|(s_0, s) \in R^T_M \land (s, s_f) \in R^T_M\};
\]

\[
E := \{(s, \epsilon, s')|s, s' \in N \land \exists a \in \nabla_M : (s, a, s') \in R_{\text{env}}\}
\]

\[
\cup\{(s, a, s')|s, s' \in N \land (s, a, s') \in R_{\text{comm}}\};
\]

return TestIO(\( X, \text{reset}, s_0, s_f, \text{level} = 0, \text{count} = 0 \));

**end procedure**

### 4.2.2 Liveness Testing

To check whether the constructed communication graph \( G \) has a path that witnesses the truth of the original problem, the straightforward way is to try out all paths in
CHAPTER 4. MODEL CHECKING SYSTEMS WITH ONE BLACK-BOX

$G$ and then check, whether along some path, the system can reach $s_f$ from $s_0$ first and then reach $s_f$ for infinitely many times. The check is done by testing $X$ with the communication trace of the path to see whether it is an observable behavior of $X$. However, one difficulty is that $G$ may contain loops, and certainly we can only test $X$ with a finite communication trace. Fortunately, the following observations are straightforward

- To check whether the system can reach $s_f$ from $s_0$, we only need to consider paths with length less than $mn_1$ where $n_1$ is the maximal number of communications on all simple paths (i.e., no loops on the path) between $s_0$ and $s_f$ in $G$, and $m$ is the upper bound for the number of states in the black-box component $X$.

- To check whether the system can reach from $s_f$ to $s_f$ for infinitely many times, we only need to make sure that the system can reach $s_f$ for $m - 1$ times, and between $s_f$ and $s_f$, the system goes through a path no longer than $n_2$ that is the maximal number of communications on all simple loops (i.e., no nested loops along the loop) in $G$ that pass $s_f$.

Let $n = max(n_1, n_2)$. The following procedure TestIO uses a bounded and nested depth-first search to traverse the graph $G$ while testing $X$. The algorithm maintains a sequence $\pi$ of action symbols that has been successfully accepted by $X$, an integer variable level that records how many communications have been gone through without reaching $s_f$, and an integer variable count that indicates how many times $s_f$ has been reached.

It first tests whether the system can reach $s_f$ from $s_0$ along a path with length less than $mn$, then it tests whether the system can further reach $s_f$ to $s_f$ for $m - 1$ more times. At each step, it first tests whether the original (infinite-often) property is true at any state $s'$ such that the system can reach $s'$ from $s_0$ through an environment/internal transition. It returns true if that’s the case. Otherwise, it chooses one candidate from the set of all possible action symbols at a node, and feeds $\text{reset}\pi$ concatenated with that symbol to $X$ ($\text{reset}$ is used to brings $X$ to its initial state). If $X$ accepts the sequence, the procedure moves forward to try the destination node
of the edge with level increased by 1. If X does not accept, then the procedure keeps trying the next candidate. The procedure returns false when all candidates are tried without an acceptance, or when more than mn communications have been gone through without reaching sf. After sf is reached, the procedure increases count by 1 and resets level to 0. The procedure returns true when it has already encountered sf for m times.

procedure TestIO(X, π, s₀, sf, level, count)
    if level > mn then
        return false;
    else if s₀ = sf then
        if count >= m then
            return true;
        else
            count := count + 1; level := 0;
        end if
    end if
    for each (s₀, ε, s') ∈ E do
        BBTest(X, resetπ);
        if TestIO(X, π, s', sf, level, count) Then
            return true;
        end if
    end for
    for each (s₀, a, s') ∈ E² do
        if BBTest(X, resetπ a) == “yes” then
            if TestIO(X, π a, s', sf, level + 1, count)
                then return true;
            end if
        end if
    end for each;

²This excludes those (s, ε, s') ∈ E
return false.
end procedure

In summary, our liveness testing algorithm to solve the liveness analysis problem has two steps: (1) build the communication graph $G$; (2) return the truth of

$$TestIO(X, \text{reset}, s_0, s_f, level = 0, count = 0).$$

### 4.2.3 LTL Model Checking Driven Testing

Recall that the LTL model checking problem is, for a Kripke structure $K = \langle S, R, L \rangle$ with a state $s \in S$ and a path formula $f$, to determine if $K, s \models A \phi$. Notice that $K, s \models A \phi$ if and only if $K, s \models \neg E \neg f$. Therefore it is sufficient to only consider formulas in the form $E \phi$. The standard LTL model checking algorithm [28] first constructs a tableau $T$ for the path formula $f$. $T$ is also a Kripke structure and includes every path that satisfies $f$. Then the algorithm composes $T$ with $K$ and obtains another Kripke structure $P$ which includes exactly the set of paths that are in both $T$ and $K$. Thus, a state in $K$ satisfies $E \phi$ if and only if it is the start of a path (in the composition $P$) that satisfies $\phi$.

Define $\text{sat} (\phi)$ to be the set of states in $T$ that satisfy $\phi$ and use the convention that $(s, s') \in \text{sat}(\phi)$ if and only if $s' \in \text{sat}(\phi)$. The LTL model checking problem can be summarized by the following theorem [28].

**Theorem 4.1** $K, s \models E \phi$ if and only if there is a state $s'$ in $T$ such that $(s, s') \in \text{sat}(\phi)$ and $P, (s, s') \models EG$ true under fairness constraints $\{ \text{sat}(\neg(g U h) \vee h) \mid g U h \}$ occurs in $f$.

Note that the standard LTL model checking algorithm still applies to the system $Sys = \langle M, X \rangle$, although it contains an black-box component $X$. To see this, the construction of the tableau $T$ from $f$ and the definition of $\text{sat}$ are not affected by the black-box component $X$. The composition of $Sys$ and $T$ is a new system $Sys' = \langle P, X \rangle$ where $P$ is the composition of $M$ and $T$. Then one can show
CHAPTER 4. MODEL CHECKING SYSTEMS WITH ONE BLACK-BOX

Corollary 4.2 \( \langle M, X \rangle, s \models E f \) if and only if there is a state \( s' \) in \( T \) such that \( (s, s') \in \text{sat}(f) \) and \( \langle P, X \rangle, (s, s') \models EG \text{ true} \) under fairness constraints \( \{ \text{sat}(\neg(g U h) \lor h) \mid g U h \text{ occurs in } f \} \).

Checking whether there is a state \( s' \) in \( T \) such that \( (s, s') \in \text{sat}(f) \) is trivial. To check whether \( \langle P, X \rangle, (s, s') \models EG \text{ true} \) under the fairness constraints is equivalent to checking whether there is computation in \( \langle P, X \rangle \) that starts from \( (s, s') \) and on which the fairness constraints are true infinitely often. One can show that this is equivalent to the liveness analysis problem we studied in the previous subsection, and thus, the LTL model checking problem can be solved by extending our algorithms for the liveness analysis problem. Moreover, the algorithms are both complete and sound.

4.3 CTL Model Checking Driven Black-box Testing

In this section, we introduce algorithms for CTL model checking driven black-box testing for the system \( Sys = \langle M, X \rangle \).

4.3.1 The Ideas

Recall that the CTL model checking problem is, for a Kripke structure \( K = (S, R, L) \), a state \( s_0 \in S \), and a CTL formula \( f \), to check whether \( K, s_0 \models f \) holds. The standard algorithm [28] for this problem operates by searching the structure and, during the search, labeling each state \( s \) with the set of sub-formulas of \( f \) that are true at \( s \). Initially, labels of \( s \) are just \( L(s) \). Then, the algorithm goes through a series of stages—during the \( i \)-th stage, sub-formulas with the \( (i - 1) \)-nested CTL operators are processed. When a sub-formula is processed, it is added to the labels for each state where the sub-formula is true. When all the stages are completed, the algorithm returns \text{true} when \( s_0 \) is labeled with \( f \), or \text{false} otherwise.

However, if a system is not completely specified, the standard algorithm does not work. This is because, in the system \( Sys = \langle M, X \rangle \), transitions of \( M \) may depend
on communications with the black-box component $X$. In this section, we adapt the standard CTL model checking algorithm [28] to handle the system $Sys$ (i.e., to check whether

$$
\langle M, X \rangle, s_0 \models f
$$

holds where $s_0$ is an initial state in $M$ and $f$ is a CTL formula).

The new algorithm follows a structure similar to the standard one. It also goes through a series of stages to search $M$’s state space and label each state during the search. However, during a stage, processing the sub-formulas is rather involved, since the truth of a sub-formula $h$ at a state $s$ can not be simply decided (it may depend on communications). Similar to the algorithm for the liveness analysis problem, our idea here is to construct a graph representing all the paths that witness the truth of $h$ at $s$. But, the new algorithm is far more complicated than the liveness testing algorithm for LTL, since the truth of a CTL formula is usually witnessed by a tree instead of a single path. In the new algorithm, processing each sub-formula $h$ is sketched as follows.

When $h$ takes the form of $EX g$, $E[g_1 U g_2]$, or $EG g$, we construct a graph that represents exactly all the paths that witness the truth of $h$ at some state. We call such a graph the sub-formula’s witness graph (WG), written as $[h]$. We also call $[h]$ an $EX$ graph, an $EU$ graph, or an $EG$ graph if $h$ takes the form of $EX g$, $E[g_1 U g_2]$, or $EG g$, respectively.

Let $k$ be the total number of CTL operators in $f$. In the algorithm, we construct $k$ WGs, and for each WG, we assign it with a unique ID number that ranges between 2 and $k + 1$. (The ID number 1 is reserved for constant true.) Let $\mathbb{I}$ be the mapping from the WGs to their IDs; i.e., $\mathbb{I}([h])$ denotes the ID number of $h$’s witness graph, and $\mathbb{I}^{-1}(i)$ denotes the witness graph with $i$ as its ID number, $1 < i \leq k + 1$. We label a state $s$ with ID number 1 if $h$ is true at $s$ and the truth does not depend on communications between $M$ and $X$. Otherwise, we label $s$ with $2 \leq i \leq k + 1$ if $h$ could be true at $s$ and the truth would be witnessed only by some paths which start from $s$ in $\mathbb{I}^{-1}(i)$ and, on which, communications are involved.

When $h$ takes the form of a Boolean combination of sub-formulas using $\neg$ and $\lor$,
the truth of $h$ at state $s$ is also a logic combination of the truths of the component sub-formulas at the same state. To this end, we label the state with an \textit{ID expression} $\psi$ defined as follows:

- $ID := 1 \mid 2 \mid \ldots \mid k + 1$;
- $\psi := ID \mid \neg \psi \mid \psi \lor \psi$.

Let $\Psi$ denote the set of all ID expressions. For each sub-formula $h$, we construct a labeling (partial) function $L_h : S \rightarrow \Psi$ to record the ID expression labeled to each state during the processing of the sub-formula $h$, and the labeling function is returned when the sub-formula is processed.

The detailed procedure, called $ProcessCTL$, will be presented in subsection 4.3.2. After all sub-formulas are processed, a labeling function $L_f$ for the outer-most sub-formula (i.e., $f$ itself) is returned. The algorithm returns \textit{true} when $s$ is labeled with 1 by $L_f$. It returns \textit{false} when $s$ is not labeled at all. In other cases, a testing procedure over $X$ is applied to check whether the ID expression labeled in $L_f(s)$ could be evaluated true. The procedure, called $TestWG$, will be given in Section 4.3.6. In summary, the algorithm (to solve the CTL model checking problem $\langle M, X \rangle, s_0 \models f$) is sketched as follows:

\begin{verbatim}
Global id := 2;
procedure CheckCTL(M, X, s_0, f)
    L_f := ProcessCTL(M, f)
    if s_0 is labeled by L_f then
        if L_f(s_0) = 1 then
            return true;
        else
            return TestWG(X, reset, s_0, L_f(s_0));
        end if
    else (i.e., s_0 is not labeled at all)
        return false.
    end if
\end{verbatim}
end procedure

Note that the id is a global variable to be used to give each witness graph constructed in the process an ID number. The next subsections introduce the main algorithm for checking a CTL formula as well as algorithms for handling each CTL operator.

4.3.2 Processing a CTL formula

Processing a CTL formula $h$ is implemented through a recursive procedure $ProcessCTL$. Recall that any CTL formula can be expressed in terms of $\lor$, $\neg$, $EX$, $EU$, and $EG$. Thus, at each intermediate step of the procedure, depending on whether the formula $h$ is atomic or takes one of the following forms: $g_1 \lor g_2$, $\neg g$, $EX \ g$, $E[g_1 U g_2]$, or $EG \ g$, the procedure has only six cases to consider and when it finishes, a labeling function $L_h$ is returned for formula $h$.

procedure $ProcessCTL(M, h)$

Case

$h$ is atomic: Let $L_h$ label every state with 1 whenever $h$ is true on the state;

$h = g_1 \lor g_2$:

$L_{g_1} := ProcessCTL(M, g_1)$;
$L_{g_2} := ProcessCTL(M, g_2)$;
$L_h := HandleUnion(L_{g_1}, L_{g_2})$;

$h = \neg g$:

$L_g := ProcessCTL(M, g)$;
$L_h := HandleNegation(M, L_g)$;

$h = EX \ g$:

$L_g := ProcessCTL(M, g)$;
$L_h := HandleEX(M, L_g)$;

$h = E[g_1 U g_2]$:

$L_{g_1} := ProcessCTL(M, g_1)$;
$L_{g_2} := ProcessCTL(M, g_2)$;
CHAPTER 4. MODEL CHECKING SYSTEMS WITH ONE BLACK-BOX  56

\[ L_h := \text{HandleEU}(M, L_{g_1}, L_{g_2}); \]
\[ h = \text{EG } g; \]
\[ L_g := \text{ProcessCTL}(M, g); \]
\[ L_h := \text{HandleEG}(M, L_g); \]

end case

return \( L_h \).

end procedure

In the above procedure, when \( h = g_1 \lor g_2 \), we first process \( g_1 \) and \( g_2 \) respectively by calling \( \text{ProcessCTL} \), then construct a labeling function \( L_h \) for \( h \) by calling \( \text{HandleUnion} \), which merges \( g_1 \) and \( g_2 \)'s labeling functions \( L_{g_1} \) and \( L_{g_2} \) as follows:

- For each state \( s \) that is in both \( L_{g_1} \)'s domain and \( L_{g_2} \)'s domain, let \( L_h \) label \( s \) with 1 if either \( L_{g_1} \) or \( L_{g_2} \) labels \( s \) with 1 and label \( s \) with ID expression \( L_{g_1}(s) \lor L_{g_2}(s) \) otherwise;

- For each state \( s \) that is in \( L_{g_1} \)'s domain (resp. \( L_{g_2} \)'s domain) but not in \( L_{g_2} \)'s domain (resp. \( L_{g_1} \)'s domain), let \( L \) label \( s \) with \( L_{g_1}(s) \) (resp. \( L_{g_2}(s) \)).

The above handling can be summarized as the following procedure:

procedure \( \text{HandleUnion}(L_1, L_2) \)

\[ L := \emptyset; \]

for each \( s \in \text{dom}(L_1) \cup \text{dom}(L_2) \) do

if \( s \in \text{dom}(L_1) \cap \text{dom}(L_2) \) then

if \( L_1(s) = 1 \lor L_2(s) = 1 \) then

\[ L := L \cup \{(s, 1)\}; \]

else

\[ L := L \cup \{(s, L_1(s) \lor L_2(s))\}; \]

end if

else if \( s \in \text{dom}(L_1) \) then

\[ L := L \cup \{(s, L_1(s))\}; \]

else

\[ L := L \cup \{(s, L_2(s))\}; \]

end if

end for
CHAPTER 4. MODEL CHECKING SYSTEMS WITH ONE BLACK-BOX

end if
end for
return \( L \);
end procedure

When \( h = \neg g \), we first process \( g \) by calling \( \text{ProcessCTL} \), then construct a labeling function \( L_h \) for \( h \) by “negating” (\( \text{HandleNegation} \)) \( g \)'s labeling function \( L_g \) as follows:

- For every state \( s \) that is not in the domain of \( L_g \), let \( L_h \) label \( s \) with 1;
- For each state \( s \) that is in the domain of \( L_g \) but not labeled with 1 by \( L_g \), let \( L_h \) label \( s \) with ID expression \( \neg L_g(s) \).

The above handling can be summarized as the following procedure:

\textbf{procedure} \( \text{HandleNegation}(M, L_1) \)
\begin{align*}
    &L := \emptyset; \\
    &\text{for each } s \in S \text{ do} \\
    &\quad \text{if } s \notin \text{dom}(L_1) \text{ then} \\
    &\quad \quad L := L \cup \{(s, 1)\}; \\
    &\quad \text{else if } f(s) \neq 1 \text{ then} \\
    &\quad \quad L := L \cup \{(s, \neg L_1(s))\}; \\
    &\quad \text{end if} \\
    &\text{return } L; \\
&\text{end procedure}
\end{align*}

The remaining three cases (i.e., for \( EX, EU, \) and \( EG \)) in the above procedure are more complicated and are handled in the following three subsections respectively.

4.3.3 Checking an EX Sub-Formula

When \( h = EXg \), \( g \) is processed first by \( \text{ProcessCTL} \), then \( \text{HandleEX} \) is called with \( g \)'s labeling function \( L_g \) to construct a labeling function \( L_h \) and create a witness graph for \( h \) (we assume that, whenever a witness graph is created, the current value of a global variable \( id \), which initially is 2, is assigned as the ID number of the graph, and \( id \) is incremented by 1 after it is assigned to the graph).
CHAPTER 4. MODEL CHECKING SYSTEMS WITH ONE BLACK-BOX

The labeling function \( L_h \) is constructed as follows. For each state \( s \) that has a successor \( s' \) in the domain of \( L_g \), if \( s \) can reach \( s' \) through an environment transition and \( s' \) is labeled with 1 by \( L_g \) then let \( L_h \) also label \( s \) with 1, otherwise let \( L_h \) label \( s \) with the current value of the global variable \( id \).

The witness graph for \( h = EXg \), called an \( EX \) graph, is created as a triple:

\[
[[h]] = \langle N, E, L_g \rangle,
\]

where \( N \) is a set of nodes and \( E \) is a set of annotated edges. It is created as follows.

- Add one node to \( N \) for each state that is in the domain of \( L_g \).
- Add one node to \( N \) for each state that has a successor in the domain of \( L_g \).
- Add one edge between two nodes in \( N \) to \( E \) when \( M \) has a transition between two states corresponding to the two nodes respectively; if the transition involves a communication with \( X \) then annotate the edge with the communication symbols.

When \( HandleEX \) finishes, it increases the global variable \( id \) by 1 (since one new witness graph has been created).

**procedure** \( HandleEX(M, L_1) \)

\[ N := \text{dom}(L_1); L := \emptyset; \]

for each \( t \in \text{dom}(L_1) \) do

for each \( s : R^*(s, t) \) do

\[ N := N \cup \{s\} \]

if \( L_1(t) = 1 \land R^*_{env}(s, t) \) Then

if \( s \not\in \text{dom}(L) \) then

\[ L := L \cup \{(s, 1)\}; \]

else if \( L(s) \neq 1 \) then

\[ L := L|_{s+1}; \]

end if

else if \( s \not\in \text{dom}(L) \) then


\[ L := L \cup \{(s, id)\}; \]

end for
end for
end if
\[ E := \{(s, \epsilon, s')|s' \in \text{dom}(f) \land \exists a : (s, a, s') \in R_{env}\} \]
\[ \cup \{(s, a, s')|s' \in \text{dom}(f) \land (s, a, s') \in R_{comm}\}; \]
Associate \( id \) with \( G = \langle N, E, L_1 \rangle; \) \( id := id + 1; \)
return \( L; \)
end procedure

4.3.4 Checking an EU Sub-Formula

The case when \( h = E [g_1 U g_2] \) is more complicated. We first process \( g_1 \) and \( g_2 \) respectively by calling \textit{ProcessCTL}, then call procedure \textit{HandleEU} with \( g_1 \) and \( g_2 \)'s labeling functions \( L_{g_1} \) and \( L_{g_2} \) to construct a labeling function \( L_h \) and create a witness graph for \( h. \)

We construct the labeling function \( L_h \) recursively. First, let \( L_h \) label each state \( s \) in the domain of \( L_{g_2} \) with \( L_{g_2}(s) \). Then, for state \( s \) that has a successor \( s' \) in the domain of \( L_h \), if both \( s \) and \( s' \) is labeled with 1 by \( L_{g_1} \) and \( L_h \) respectively and \( s \) can reach \( s' \) through an environment transition then let \( L_h \) also label \( s \) with 1, otherwise let \( L_h \) label \( s \) with the current value of the global variable \( id \). Notice that, in the second step, if a state \( s \) can be labeled with both 1 and the current value of \( id \), let \( L_h \) label \( s \) with 1. Thus, we can ensure that the constructed \( L_h \) is indeed a function.

The witness graph for \( h \), called an \( EU \) graph, is created as a 4-tuple:

\[ \llbracket h \rrbracket := \langle N, E, L_{g_1}, L_{g_2} \rangle, \]

where \( N \) is a set of nodes and \( E \) is a set of edges. \( N \) is constructed by adding one node for each state that is in the domain of \( L_h \), while \( E \) is constructed in the same way as that of \textit{HandleEX}. When \textit{HandleEU} finishes, it increases the global variable \( id \) by 1.
procedure HandleEU (M, L1, L2)
L := L2;
T1 := dom(L1); T2 := dom(L);
while T2 ≠ {} do
    Choose t ∈ T2; T2 := T2 \ {t};
    for each s ∈ T1 \ R*(s, t) do
        if L1(s) = 1 ∧ L(t) = 1 ∧ R^s_{env}(s, t) then
            if s ∉ dom(L) then
                T2 := T2 ∪ {s}; L := L ∪ {(s, 1)};
            else if L(s) ≠ 1 then
                T2 := T2 ∪ {s}; L := L|_{s<1};
            end if
        end if
    end for
end while
N := dom(L);
E := \{(s, ε, s')|s, s' ∈ N ∧ ∃a : (s, a, s') ∈ R_{env}\}
∪ \{(s, a, s')|s, s' ∈ N ∧ (s, a, s') ∈ R_{comm}\};
Associate id with G = ⟨N, E, L1, L2⟩; id := id + 1;
return L;
end procedure

4.3.5 Checking an EG Sub-Formula

To handle formula h = EGg, we first process g by calling ProcessCTL, then call procedure HandleEG with g’s labeling function L_g to construct a labeling function L_h and create a witness graph for h.

The labeling function L_h is constructed as follows. For each state s that can reach a loop C through a path p such that every state (including s) on p and C is in the
domain of \( L_g \), if every state (including \( s \)) on \( p \) and \( C \) is labeled with 1 by \( L_g \) and no communications are involved on the path and the loop, then let \( L_h \) also label \( s \) with 1, otherwise let \( L_h \) label \( s \) with the current value of the global variable \( id \).

The witness graph for \( h \), called an \( EG \) graph, is created as a triple:

\[
[h] := \langle N, E, L_g \rangle,
\]

where \( N \) is a set of nodes and \( E \) is a set of annotated edges. The graph is constructed in a same way as that of \( \text{HandleEU} \). When \( \text{HandleEG} \) finishes, it also increases the global variable \( id \) by 1.

**procedure** \( \text{HandleEG}(X, \pi, s_0, G = \langle N, E, L_g \rangle) \)

\[
\begin{align*}
SCC_{env} := \{C|C \text{ is a nontrivial SCC of } M \text{ and } C \text{ contains no transitions that involve a communication with } X\}; \\
SCC_{comm} := \{C|C \text{ is a nontrivial SCC of } M \text{ and every transition in } C \text{ involves a communication with } X\}; \\
L := \{(s, 1)|\exists C \in SCC_{env} : s \in C\} \\
\cup\{(s, id)|\exists C \in SCC_{comm} : s \in C\} \\
T := \text{dom}(L); \\
\text{while } T \neq \emptyset \text{ do} \\
\text{Choose } t \in T; T := T \setminus \{t\}; \\
\text{for each } s \in \text{dom}(L_1) \land R^s(s, t) \text{ do} \\
\quad \text{if } L(t) = 1 \land L_1(s) = 1 \land R^s_{env}(s, t) \text{ then} \\
\quad \quad \text{if } s \notin \text{dom}(L) \text{ then} \\
\quad \quad \quad T := T \cup \{s\}; L := L \cup \{(s, 1)\}; \\
\quad \quad \text{else if } L(s) \neq 1 \text{ then} \\
\quad \quad \quad T := T \cup \{s\}; L := L|_{s-1}; \\
\quad \quad \text{end if} \\
\quad \text{else if } s \notin \text{dom}(L) \text{ then} \\
\quad \quad T := T \cup \{s\}; L := L \cup \{(s, id)\}; \\
\quad \text{end if} \\
\text{end for}
\end{align*}
\]
end while

\[ N := \text{dom}(L); \]
\[ E := \{ (s, \epsilon, s') | s, s' \in N \land \exists a : (s, a, s') \in R_{\text{env}} \} \]
\[ \cup \{ (s, a, s') | s, s' \in N \land (s, a, s') \in R_{\text{comm}} \}; \]

\textbf{Associate} \textit{id with} \ G = \langle N, E, L_1 \rangle; \ \textit{id} := \textit{id} + 1;

\textbf{return} \ L;

\textbf{end procedure}

### 4.3.6 Testing a Witness Graph

As mentioned in subsection 4.3.1, the procedure for CTL model checking driven black-box testing, \textit{CheckCTL}, consists of two parts. The first part, which was discussed in Section 4.3.2, includes \textit{ProcessCTL} that processes CTL formulas and creates witness graphs. The second part is to evaluate the created witness graphs through testing \( X \). We will elaborate on this second part in this section.

In processing the CTL formula \( f \), a witness graph is constructed for each CTL operator in \( f \) and a labeling function is constructed for each sub-formula of \( f \). As seen from the algorithm \textit{CheckCTL} (at the end of Section 4.3.1), the algorithm either gives a definite “yes” or “no” answer to the CTL model checking problem, i.e., \( \langle M, X \rangle, s_0 \models f \), or it reduces the problem to checking whether the ID expression \( \psi \) labeled to \( s_0 \) can be evaluated true at the state. The evaluation procedure is carried out by the following recursive procedure \textit{TestWG}, after an input sequence \( \pi \) has been accepted by the black-box component \( X \).

\begin{verbatim}
procedure TestWG(X, \pi, s_0, \psi)

Case
\[ \psi = \psi_1 \lor \psi_2; \]
if \textit{TestWG}(X, \pi, s_0, \psi_1) then
\textbf{return} \true;
else
\textbf{return} \textit{TestWG}(X, \pi, s_0, \psi_2)
end procedure
\end{verbatim}
end if
\( \psi = \neg \psi_1 \):
\textbf{return} \( \neg \text{TestWG}(X, \pi, s_0, \psi_1) \)
\( \psi = 1 \):
\textbf{return} \( \text{true} \);
\( \psi = i \) with \( 2 \leq i \leq k + 1 \):
\textbf{if} \( \llbracket i \rrbracket^{-1}(i) \) is an \( EX \) graph \textbf{Then}
\textbf{return} \( \text{TestEX}(X, \pi, s_0, \llbracket i \rrbracket^{-1}(i)) \);
\textbf{end if}
\textbf{if} \( \llbracket i \rrbracket^{-1}(i) \) is an \( EU \) graph \textbf{Then}
\textbf{return} \( \text{TestEU}(X, \pi, s_0, \llbracket i \rrbracket^{-1}(i), \text{level} = 0) \);
\textbf{end if}
\textbf{if} \( \llbracket i \rrbracket^{-1}(i) \) is an \( EG \) graph \textbf{Then}
\textbf{return} \( \text{TestEG}(X, \pi, s_0, \llbracket i \rrbracket^{-1}(i)) \).
\textbf{end if}
\textbf{end case}
\textbf{end procedure}

In \( \text{TestWG} \), the first three cases are straightforward, which are consistent with the intended meaning of ID expressions. The cases \( \text{TestEX}, \text{TestEU}, \text{TestEG} \) for evaluating \( EX, EU, EG \) graphs are discussed in the following three subsections.

### 4.3.7 Testing an EX Graph

The case for checking whether an \( EX \) graph \( G = \langle N, E, L_g \rangle \) can be evaluated true at a state \( s_0 \) is simple. We just test whether the system \( M \) can reach from \( s_0 \) to another state \( s' \in \text{dom}(L_g) \) through a transition in \( G \) such that the ID expression \( L_g(s') \) can be evaluated true at \( s' \).

The algorithm for testing an \( EX \) graph is simple. It first checks whether \( L_1(s') \) can be evaluated true at any state \( s' \) such that the system can reach \( s' \) from \( s_0 \) through an environment/interaction transition. It returns true if it is the case. Otherwise, it chooses one candidate from the set of all possible action symbols from \( s_0 \), and feeds
the sequence \( \text{reset} \pi \) concatenated with that symbol to \( X \) (\( \text{reset} \) is used to brings \( X \) to its initial state). If \( X \) accepts this action sequence, it moves forward to try the destination node of the edge. If \( X \) does not accept, then it keeps trying the next candidate. The algorithm returns \( \text{false} \) when all candidates are tried without an acceptance.

\[
\text{procedure } \text{TestEX}(X, \pi, s_0, G = \langle N, E, L_1 \rangle) \\
\text{for each } (s_0, \epsilon, s') \in E : s' \in \text{dom}(L_1) \text{ do} \\
\quad \text{BBTest}(X, \text{reset} \pi); \\
\quad \text{if } \text{TestWG}(X, \pi, s', L_1(s')) \text{ Then} \\
\quad \quad \text{return } \text{true}; \\
\quad \text{end if} \\
\text{end for} \\
\text{for each } (s_0, a, s') \in E^2 \text{ do} \\
\quad \text{if } \text{BBTest}(X, \text{reset} \pi, a) = \text{"yes"} \text{ then} \\
\quad \quad \text{if } \text{TestWG}(X, \pi \alpha, s', L_1(s')) \text{ then} \\
\quad \quad \quad \text{return } \text{true}; \\
\quad \quad \text{end if} \\
\quad \text{end if} \\
\quad \text{return } \text{false}; \\
\text{end procedure}
\]

### 4.3.8 Testing an EU Graph

To check whether an EU graph \( G = \langle N, E, L_{g_1}, L_{g_2} \rangle \) can be evaluated true at a state \( s_0 \), we need to traverse all paths \( p \) in \( G \) with length less than \( mn \) and test the black-box component \( X \) to see whether the system can reach some state \( s' \in \text{dom}(L_{g_2}) \) through one of those paths. In here, \( m \) is an upper bound for the number of states in the black-box component \( X \) and \( n \) is the maximal number of communications on all simple paths between \( s_0 \) and \( s' \). In the meantime, we should also check whether \( L_{g_2}(s') \) can be evaluated true at \( s' \) and whether \( L_{g_1}(s_i) \) can be evaluated true at \( s_i \).
for each $s_i$ on $p$ (excluding $s'$) by calling $TestWG$.

The procedure $TestEU$ keeps a sequence of action symbols $\pi$ that has been successfully accepted by $X$ and an integer $level$ that records how many communications have been gone through without reaching a destination state. And the algorithm works as follows. At first, it checks whether it has gone through more than $mn$ communications without success, it returns false if it is the case. Then, it checks whether it has reached a destination state (i.e., $s_0 \in \text{dom}(L_2)$). If it is the case, it returns $true$ when $L_2(s_0)$ can be evaluated true $s_0$. Next, it checks whether $L_1(s_0)$ can be evaluated true at $s_0$, it returns false if it is not the case. After that, it checks whether $L_1(s')$ can be evaluated true at any state $s'$ such that the system can reach $s'$ from $s_0$ through an environment/internal transition. It returns true if it is the case. Otherwise, it chooses one candidate from the set of all possible action symbols from $s_0$, and feeds the sequence $\text{reset}\pi$ concatenated with that symbol to $X$ ($\text{reset}$ is used to brings $X$ to its initial state). If $X$ accepts this action sequence, it moves forward to try the destination node of the edge. If $X$ does not accept, then it keeps trying the next candidate. The algorithm returns $false$ when all candidates are tried without an acceptance.

\begin{verbatim}
procedure TestEU(X, \pi, s_0, G = (N, E, L_1, L_2), level)
if level > mn then \footnote{Here, $n$ always denotes the maximal number of communications on any simple paths in $G$.}
    return false;
else if s_0 \in \text{dom}(L_2) then
    if TestWG(X, \pi, s_0, L_2(s_0)) then
        return true;
    end if
else if not TestWG(X, \pi, s_0, L_1(s_0)) then
    return false;
end if
for \exists s' : (s_0, \epsilon, s') \in E do
    BBTest(X, reset\pi);
    if TestEU(X, \pi, s', G, level) then
\end{verbatim}
CHAPTER 4. MODEL CHECKING SYSTEMS WITH ONE BLACK-BOX

\[
\text{return } true; \\
\text{end if} \\
\text{end for} \\
\text{for each } (s, a, s') \in E^{-1} \text{ do} \\
\quad \text{if } BBTest(X, reset_\pi a) = \text{"yes" then} \\
\quad \quad \text{if } TestEU(X, \pi_\alpha, s', G, level + 1) \text{ Then} \\
\quad \quad \quad \text{return } true; \\
\quad \quad \text{end if} \\
\quad \text{end if} \\
\text{end for each;} \\
\text{return } false; \\
\text{end procedure}
\]

4.3.9 Testing an EG Graph

For the case to check whether an EG graph \( G = \langle N, E, L_g \rangle \) can be evaluated true at a state \( s_0 \), we need to find an infinite path in \( G \) along which the system can run forever.

The following procedure \( TestEG \) first decomposes \( G \) into a set of SCCs. Then, for each state \( s_f \) in the SCCs, it calls another procedure \( SubTestEG \) to test whether the system can reach \( s_f \) from \( s_0 \) along a path not longer than \( mn \), as well as whether the system can further reach \( s_f \) from \( s_f \) for \( m - 1 \) times. The basic idea of \( SubTestEG \) is similar to that of the \( TestIO \) algorithm in Section 4.2.2, except that we need also check whether \( L_g(s_i) \) can be evaluated true at \( s_i \) for each state \( s_i \) that has been reached so far by calling \( TestWG \). Here, \( m \) is the same as before while \( n \) is the maximal number of communications on all simple paths between \( s_0 \) and \( s_f \).

\[
\text{procedure } TestEG(X, \pi, s_0, G = \langle N, E, L_g \rangle) \\
\text{SCC} := \{C|C \text{ is a nontrivial SCC of } G\}; \\
T := \bigcup_{C \in SCC} \{s|s \in C\}; \\
\text{for each } s \in T \text{ do} \\
\quad BBTest(X, reset_\pi); \\
\]

\[
\text{return } \\
\text{end procedure}
\]
if $SubTestEG(X, \pi, s_0, s, G, level = 0, count = 0)$;
    return true;
end if
end for
return false.
end procedure

The procedure $SubTestEG$ keeps a sequence of action symbols that has been successfully accepted by $X$, an integer $level$ that records how many communications have been gone through without reaching $s_f$, and an integer $count$ that indicates how many times $s_f$ has been reached. It first checks whether it has gone through more than $mn$ communications without reaching $s_f$, it returns false if it is the case. Then, it checks whether it has reached the given state $s_f$. If it is the case, it returns $true$ when it has already reached $s_f$ for $m$ times, it increases $count$ by 1 and resets $level$ to 0 when otherwise. The next, it tests whether $L_1(s_0)$ can be evaluated true at $s_0$, and it returns false if it is not the case. After that it checks whether $L_1(s')$ can be evaluated true at any state $s'$ such that the system can reach $s'$ from $s_0$ through an environment/internal transition. It returns true if it is the case. Otherwise, it chooses one candidate from the set of all possible action symbols from $s_0$, and feeds the sequence $reset\pi$ concatenated with that symbol to $X$ ($reset$ is used to brings $X$ to its initial state). If $X$ accepts this action sequence, it moves forward to try the destination node of the edge. If $X$ does not accept, then it keeps trying the next candidate. The algorithm returns $false$ when all candidates are tried without an acceptance.

procedure $SubTestEG(X, \pi, s_0, s_f, G = \langle N, E, L_1 \rangle, level, count)$
    if $level > mn$ then
        return false;
    else
        if $s_0 = s_f$
            if $count >= m$
                return true;
            else

        end if
    end if

end procedure
\[
\text{count} := \text{count} + 1; \ \text{level} := 0;
\]
end if
else if not \text{TestWG}(X, \pi, s_0, L_1(s_0)) \text{ then}
\text{return false;}
end if
for each \((s_0, \epsilon, s') \in E\) do
\text{BBTest}(X, \text{reset} \pi);
if \text{SubTestEG}(X, \pi, s', s_f, G, \text{level, count}) \text{ Then}
\text{return true;}
end if
end for
for each \((s_0, a, s') \in E\) do
if \text{BBTest}(X, \text{reset} \pi) a = \text{“yes”} \text{ then}
if \text{SubTestEG}(X, \pi a, s', s_f, G, \text{level + 1, count}) \text{ then}
\text{return true;}
end if
end if
end for;
\text{return false;}
end procedure

\textbf{Remark.} In summary, to solve the CTL model checking problem

\[(M, X), s_0 \models f,\]

our algorithm \text{CheckCTL} in Section 4.3.1 either gives a definite yes/no answer or gives a sufficient and necessary condition in the form of ID expressions and witness graphs. The condition is evaluated through black-box testing over the black-box component \(X\). The evaluation process will terminate with a yes/no answer to the model checking problem. One can show that our algorithm is both complete and sound.
4.4 Examples

In this section, to better understand our algorithms, we look at some examples\(^4\).

Consider a system \( Sys = \langle M, X \rangle \) where \( M \) keeps receiving messages from the outside environment and then transmits the message through the black-box component \( X \). The environment has one output action \( msg \), and \( X \) has one input actions \( send \), and three output actions \( yes \) and \( no \), while \( M \) share all these actions with the environment and \( X \) (i.e., \( msg, ack, yes, \) and \( no \) are \( M \)'s input action while \( send \) is \( M \)'s output action). The transition graph of \( M \) is depicted in Figure 4.1.

**Figure 4.1: Example 4.3**

![Diagram](image)

**Figure 4.2: Communication Graph of Example 4.3**

**Example 4.3** Assume that we want to solve the following LTL model checking problem

\[(M, X), s_0 \models EGFs_2\]

i.e., starting from the initial state \( s_0 \), the system can reach state \( s_1 \) infinitely often. Applying our liveness analysis algorithms, we can obtain the (minimized) communication graph in Figure 4.2. From this graph and our liveness testing algorithms, the system satisfies the liveness property iff the communication trace

\[
send \ yes(send \ yes \ ack)^{n-1} \]

\(^4\)The transition graphs in the figures in this section are not made total for the sake of readability.
is an observable behavior of $X$, where $m$ is an upper bound for number of states in $X$. 

![Diagram](image)

**Figure 4.3: Example 4.4**

**Example 4.4** Now, we slightly modify the transition graph of $M$ into Figure 4.3 such that when a send fails, the system shall return to the initial state. For this modified system, its (minimized) communication graph with respect to the liveness property would be as shown in Figure 4.4. From Figure 4.4 and the liveness testing algorithms, we know that the system satisfies the liveness property iff there exist $0 \leq k_1, k_2 \leq 2m$ such that the communication trace

$$(send\ no)^{k_1}send\ yes((send\ yes\ ack)(send\ no)^{k_2})^{m-1}$$

is an observable behavior of $X$. 

![Diagram](image)

**Figure 4.4: Communication Graph of Example 4.4**

**Example 4.5** Still consider the system in Figure 4.3, but we want to solve a CTL model checking problem $(M, X), s_0 \models AFs_2$; i.e., along all paths from $s_0$, the system
can reach state $s_1$ eventually. The problem is equivalent to

$$(M, X), s_0 \models \neg \mathcal{G} \neg s_2.$$ 

Applying our CTL algorithms to formula $h = \mathcal{G} \neg s_2$, we construct an $\mathcal{EG}$ witness graph $G = \langle N, E, L_{\text{true}} \rangle$ whose ID number is 2 and a labeling function $L_h$, where $L_{\text{true}}$ labels all three states $s_0, s_1$, and $s_3$ with ID expression 1 (as defined in Section 4.3.1, which stands for true), and $L_h$ labels all three states $s_0, s_1$, and $s_3$ with 2. The graph $G$ is depicted in Figure 4.5. From this graph as well as $L_h$, the algorithms conclude that the model checking problem is true iff the communication trace $(\text{send no})^{m-1}$ is not an observable behavior of $X$. 

\[ \square \]

Example 4.6 Now we modify the system in Figure 4.1 into a more complicated one shown in Figure 4.6. For this system, we want to check

$$(M, X), s_0 \models \neg \mathcal{E} [\neg s_2 U s_3]$$ 

i.e., starting from the initial state $s_0$, the system should never reach state $s_3$ earlier.
than it reaches $s_2$. Applying our CTL algorithms to formula

$$h = E[\neg s_2 U s_3],$$

we obtain an $EU$ witness graph $G = \langle N, E, L_1, L_2 \rangle$ whose ID number is 2 and a labeling function $L_h$, where $L_1$ labels all four states $s_0$, $s_1$, $s_3$ and $s_4$ with 1, $L_2$ just labels $s_3$ with 1, and $L_h$ labels states $s_0$, $s_1$, and $s_4$ with 2, and labels $s_3$ with 1. The graph $G$ is depicted in Figure 4.7. From this graph as well as $L_h$, the algorithms conclude that the model checking problem is true iff none of communication traces in the form of send no(ack yes send no)\(^*\) and with length less than $3m$ is an observable behavior of $X$.

![Witness Graph for Example 4.6](image)

**Example 4.7** For the same system, we could consider more complicated temporal properties as follows:

1. $(M, X) \models AG(s_2 \rightarrow AFs_3)$; i.e., starting from the initial state $s_0$, whenever the system reaches $s_2$, it would eventually reach $s_3$.

2. $(M, X), s_0 \models AG(s_2 \rightarrow AXA[\neg s_2 U s_3])$; i.e., starting from the initial state $s_0$, whenever it reaches state $s_2$, the system should never reach $s_2$ again until it reaches $s_3$.

To check whether $(M, X) \models AG(s_2 \rightarrow AFs_3)$, is equivalent to checking whether

$$(M, X) \models \neg E[true \ U (s_2 \land EG \neg s_3)].$$
We describe how the formula

\[ f = E[\text{true } U(s_2 \land EG\neg s_3)] \]

is processed by Handle\text{CTL} from bottom to up as follows.

1. The atomic sub-formula \( s_2 \) is processed by Handle\text{CTL}, and a labeling function \( L_1 = \{(s_2, 1)\} \) is returned.

2. The atomic sub-formula \( s_3 \) is processed, and a labeling function \( L_2 = \{(s_3, 1)\} \) is returned.

3. To process \( \neg s_3 \), Handle\text{Negation} is called with \( L_2 \) to return a labeling function \( L_3 = \{(s_0, 1), (s_1, 1), (s_2, 1), (s_4, 1)\} \).

4. To process \( EG\neg s_3 \), Handle\text{EG} is called with \( L_3 \) to construct an \text{EG graph} \( G_1 = \langle N, E, L_3 \rangle \) with id 2 (see Figure 4.8) and return a labeling function \( L_4 = \{(s_0, 2), (s_1, 2), (s_2, 2)\} \).

5. To process \( s_2 \land EG\neg s_3 \), Handle\text{Negation} and Handle\text{Union} are called with \( L_1 \) and \( L_4 \) to return a labeling function \( L_5 = \{(s_2, 2)\} \).

6. To process \( E[\text{true } U(s_2 \land EG\neg s_3)] \), Handle\text{EU} is called with \( L_5 \) to construct an \text{EU graph} \( G_2 = \langle N, E, L_5 \rangle \) with id 3 (see Figure 4.9) and return a labeling function

\[ L_f = \{(s_0, 3), (s_1, 3), (s_2, 3), (s_3, 3), (s_4, 3)\} \]

Since \( s_0 \) is labeled by \( L_f \) with an ID expression 3 instead of 1 (i.e., true), we need to test whether the ID expression 3 can be evaluated true at \( s_0 \) by calling TestWG.
with \( s_0 \) and \( G_2 \). It’s easy to see that, essentially \( TestWG \) would be testing whether some communication trace (with bounded length) with two consecutive symbol pairs (\textit{send yes}) is an observable behavior of \( X \). It returns \textit{false} if such trace exists, or vice versa.

To check whether \((M, X), s_0 \models AG(s_2 \rightarrow AXA[\neg s_2Us_3])\), is equivalent to checking whether

\[
(M, X) \models \neg E[true \ U(s_2 \land EX(E[\neg s_3U(s_2 \land \neg s_3)] \lor EG\neg s_3))].
\]

We describe how the formula

\[
f = E[true \ U(s_2 \land EX(E[\neg s_3U(s_2 \land \neg s_3)] \lor EG\neg s_3))]
\]

is processed by \textit{HandleCTL} from bottom to up in the following six step process.

1. The atomic sub-formula \( s_2 \) is processed by \textit{HandleCTL}, and a labeling function \( L_1 = \{(s_2, 1)\} \) is returned.

2. The atomic sub-formula \( s_3 \) is processed, and a labeling function \( L_2 = \{(s_3, 1)\} \) is returned.

3. To process \( \neg s_3 \), \textit{HandleNegation} is called with \( L_2 \) to return a labeling function \( L_3 = \{(s_0, 1), (s_1, 1), (s_2, 1), (s_4, 1)\} \).

4. To process \( s_2 \land \neg s_3 \), \textit{HandleNegation} and \textit{HandleUnion} are called with \( L_1 \) and \( L_3 \) to return a labeling function \( L_4 = \{(s_2, 1)\} \).
5. To process $E[\neg s_3 U (s_2 \land \neg s_3)]$, $HandleEU$ is called with $L_3$ and $L_4$ to construct an $EU$ graph $G_1 = \langle N, E, L_3, L_4 \rangle$ with id 2 (see Figure 4.10) and return a labeling function $L_5 = \{(s_0, 2), (s_1, 2), (s_2, 1)\}$.

![Figure 4.10: Witness Graph 2 for Example 4.7.2](image)

6. To process $EG\neg s_3$, $HandleEG$ is called with $L_3$ to construct an $EG$ graph $G_2 = \langle N, E, L_3 \rangle$ with id 3 (see Figure 4.11) and return a labeling function $L_6 = \{(s_0, 3), (s_1, 3), (s_2, 3)\}$.

![Figure 4.11: Witness Graph 3 for Example 4.7.2](image)

7. To process $E[\neg s_3 U (s_2 \land \neg s_3)] \lor EG\neg s_3$, $HandleUnion$ is called with $L_5$ and $L_6$ to return a labeling function $L_7 = \{(s_0, 2 \lor 3), (s_1, 2 \lor 3), (s_2, 1)\}$.

8. To process $EX(E[\neg s_3 U (s_2 \land \neg s_3)] \lor EG\neg s_3)$, $HandleEX$ is called with $L_7$ to construct an $EX$ graph $G_3 = \langle N, E, L_7 \rangle$ with id 4 (see Figure 4.12) and return a labeling function $L_8 = \{(s_0, 4), (s_1, 1), (s_2, 4), (s_3, 4)\}$.

![Figure 4.12: Witness Graph 4 for Example 4.7.2](image)
9. To process $s_2 \land EX(E[\neg s_3 U (s_2 \land \neg s_3)] \lor EG \neg s_3)$, \textit{HandleNegation} and \textit{HandleUnion} are called with $L_1$ and $L_8$ to return a labeling function $L_9 = \{(s_2, 4)\}$.

10. To process $E[true U (s_2 \land EX(E[\neg s_3 U (s_2 \land \neg s_3)] \lor EG \neg s_3))]$, \textit{HandleEU} is called with $L_9$ to construct an $EU$ graph $G_4 = \langle N, E, L_5 \rangle$ with id 5 (see Figure 4.13) and return a labeling function

$$L_f = \{(s_0, 5), (s_1, 5), (s_2, 5), (s_3, 5), (s_4, 5)\}.$$

![Figure 4.13: Witness Graph 5 for Example 4.7.2](image)

Since $s_0$ is labeled by $L_f$ with an ID expression 5 instead of 1 (i.e., $true$), we need to test whether the ID expression 5 can be evaluated true at $s_0$ by calling $TestWG$ with $s_0$ and $G_4$. It’s easy to see that, essentially $TestWG$ would be testing whether some communication trace (with bounded length) with two consecutive symbol pairs ($send$ \textit{yes}) is an observable behavior of $X$. It returns \textit{false} if such trace exists and returns \textit{true} otherwise.

### 4.5 Discussions

In this chapter, we present algorithms for LTL and CTL model checking driven black-box testing. The algorithms create communication graphs and witness graphs, on which a bounded and nested depth-first search procedure is employed to run black-box testing over the black-box component. Our algorithms are both sound and complete. Though we do not have an exact complexity analysis result, our preliminary studies
show that, in the liveness testing algorithm for LTL, the maximal length of test-cases fed into the black-box component $X$ is bounded by $O(n \cdot m^2)$. For CTL, the length is bounded by $O(k \cdot n \cdot m^2)$. In here, $k$ is the number of CTL operators in the formula to be verified, $n$ is the state number in the host system, and $m$ is the state number in the component.

The next natural step is to implement the algorithms and see how well they work in practice. In the implementation, there are further issues to be addressed.

### 4.5.1 Practical Efficiency

Similar to the traditional black-box testing algorithms to check conformance between Mealy machines, the theoretical (worst-case) complexities are high in order to achieve complete coverage. However, worst-cases do not always occur in a practical system. In particular, we need to identify scenarios that our algorithms can be made more efficient. For instance, using existing ideas of abstraction [31], we might obtain a smaller but equivalent model of the host system before running the algorithms. We might also, using additional partial information about the component, to derive a smaller state number for the component and to find ways to expedite the model checking process. Notice that the number is actually the state number for a minimal automaton that has the same set input/output sequences as the component. Additionally, in the implementation, we also need a database to record the test results that have been performed so far (so repeated testing can be avoided). Algorithms are needed to make use of the test results to aggressively trim the communication/witness graphs such that less test-cases are performed but the complete coverage is still achieved. Also, we will study algorithms to minimize communication/witness graphs such that duplicate test-cases are avoided. Lastly, it is also desirable to modify our algorithms such that the communication/witness graphs are generated with the process of generating test-cases and performing black-box testing over the black-box component $X$. In this way, a dynamic algorithm could be designed to trim the graphs on-the-fly.
4.5.2 Coverage Metrics

Sometimes, a complete coverage will not be achieved when running the algorithms on a specific application system. In this case, a coverage metric is needed to tell how much the test-cases that have run so far cover. The metric will give a user some confidence on the partial model checking results. Furthermore, such a metric would be useful in designing conservative algorithms to debug/verify the temporal specifications that sacrifice the complete coverage but still bring the user reasonable confidence.

4.5.3 More Complex System Models

The algorithms can be generalized to systems containing multiple black-box components. Additionally, we will also consider cases when these components interacts between each other, as well as cases when the host system communicates with the components asynchronously. Obviously, when the black-box component (as well as the host system) has an infinite state space, both the traditional model checking techniques and black-box techniques are not applicable. One issue with infinite-state systems is that, the internal structure of a general infinite-state system can not be learned through the testing method. Another issue is that model checking a general infinite-state system is an undecidable problem. It is desirable to consider some restricted classes of infinite-state systems (such as real-time systems modeled as timed automata [4]) where our algorithms generalize. This is interesting, since through the study we may provide an algorithm for model checking driven black-box testing for a real-time system that contains an (untimed) black-box component. Since the algorithm will generate test-cases for the component, real-time integration testing over the composed system is avoided.
Chapter 5

Decompositional Testing of Systems With Black-box Components\(^1\)

In the previous chapter, we studied new model checking algorithms for systems with only one black-box component. But the algorithms assume that all the components in a system has a finite state space. Also the CTL algorithms are very difficult to be extended to systems with multiple black-boxes. So, in this chapter, we consider a testing problem for system with multiple black-boxes. We present an automata-theoretic approach that decomposes the global testing problem for a system with multiple black-box components into a series testing activities over each individual black-box.

This chapter is organized as follows. Section 5.1 is an overview of our approach. In Section 5.2, we present the detail of our push-in technique for the decompositional testing. In Section 5.3, we examine a set of experiments and analyze the results. Finally, Section 5.4 summarizes this chapter.

\(^1\)The content of this chapter is based upon the joint work with Z. Dang in [100]
5.1 Introduction

In our setup, a component-based system $Sys = (Gluer, B_1, \ldots, B_k)$ is defined as in (2.1), but it consists of a fully specified, and finite-state component (called the Gluer) and a number of black-box components $B_1, \ldots, B_k$, which are not necessarily of finite state space. A global behavior is just an observable behavior of the system. The global testing problem studied in this chapter is to verify (with a definite answer) that, for the given set $Bad$, none of the the system’s global behaviors is in $Bad$.

A straightforward approach to solve the global testing problem is to perform integration testing over the system as a whole and see if the system exhibits a bad behavior. However, there are fundamental difficulties with this approach. For instance, in some applications [87], integration testing may not be applicable at all. Even when integration testing is possible in some situations, the system itself is often nondeterministic. The combinatorial blow-up on the number of the executions caused by nondeterministic interleavings among the concurrent components in the system generally makes it infeasible to do thorough integration testing, while we are looking for a definite answer to the global testing problem. Due to the same reason, even when one has a way to handle the nondeterminism [88], the size of the given set $Bad$ (which could be very large, e.g., more than $10^{24}$ in some of our experiments shown later) may also make exhaustive integration testing infeasible.

A less straightforward approach is to combine testing with some formal method. For instance, one can extensively test each black-box alone and try to build [79] a partial model of the black-box from the test results. Then, one can run a formal method like model checking on the partial system model built from the partial models of the black-boxes to solve the global testing problem. However, this approach is also difficult to implement. For instance, it is hard to choose effective test sequences to build a partial model of a black-box, and it is also hard to know when the tests over a black-box are adequate. Moreover, the partial (and hence approximated) system model might not help us obtain a definite answer to the global testing problem. To avoid the above difficulties, one may also try, using some formal method, to derive an expectation condition over a black-box’s behaviors such that when every black-box
behave as expected, the system guarantees to not have a global bad behavior. Then the expectation conditions can be used to generate test sequences for the black-boxes. However, the interactions among the concurrent black-boxes make it difficult to derive such conditions automatically.

In this chapter, we introduce a novel approach (called the “push-in” technique) to solve the problem, which does not entail any integration testing. Instead, in our approach, the global testing problem is reduced to testing individual black-boxes in the system one by one in some given order. Using an automata-theoretic approach, test sequences for each individual black-box are generated from the system’s description as well as the test results of black-boxes prior to the black-box in the given order.

The first step of our approach is to compute an auxiliary set $A_1$ of sequences of observable actions for black-boxes $B_1, \cdots, B_k$ and a set $U_1$ of test sequences for black-box $B_1$. Then we test the black-box $B_1$ with test sequences in $U_1$ and collect all successful test sequences into a surviving set $SUV_1$. In the second step, from the surviving set $SUV_1$ and the auxiliary set $A_1$, we compute the auxiliary set $A_2$ (for black-boxes $B_2, \cdots, B_k$) and the test sequence set $U_2$ for black-box $B_2$. Again, after testing black-box $B_2$ with test sequences in $U_2$, we collect all successful testing sequences into a surviving set $SUV_2$. Subsequent steps follow similarly, and eventually, in the last step (i.e., step $k$), the global testing problem will be decided from the surviving sets. That is, the system has no global bad behavior iff, for some $1 \leq i \leq k$, the surviving set $SUV_i$ is empty. We also provide a procedure to recover a global bad behavior when the answer to the original problem is “no”.

Since the sets (i.e., $U_i$ and $A_i$) are provably finite and, in many cases, huge, we use (finite) automata that accept the sets as their symbolic representations, and standard automata operations are used to manipulate these sets. Also, the global testing problem is decomposed into a series of testing problems over each individual black-box in the system. Hence, our approach is an automata-theoretic and decompositional approach. Moreover, the “push-in” technique is both complete and sound, and can be carried out automatically. In particular, we show that the technique is “optimal” in the sense that each test we run over a black-box has the potential to discover a global bad behavior (i.e., we never run useless tests). In general, exhaustive integration
testing over a concurrent system is infeasible. However, our experiments show that, using the push-in technique, we can completely solve the global testing problem with a substantially smaller number of tests over the individual black-boxes, even for an extremely large set of Bad (some of our experiments performed only about $10^5$ unit tests for a Bad of size more than $10^{24}$).

### 5.2 The Push-in Technique

In this section, we present the “push-in” technique to completely solve the global testing problem, by performing unit testing over each individual black-box in the system. A test sequence is a string or a word. A finite set of test sequences is therefore a regular language and, in this chapter, we use a (finite) automaton that accepts the finite set as the symbolic representation of the set. Our push-in technique is automata-theoretic. For each $1 \leq i \leq k$, the technique generates two automata: $U_i$ and $A_i$. Automaton $U_i$, called a *unit test sequence automaton*, accepts words in alphabet $\Sigma_i$; i.e., it represents a set of test sequences for black-box $B_i$. Automaton $A_i$, called an *auxiliary automaton*, accepts words in alphabet $\Sigma_i \cup \cdots \cup \Sigma_k$ (observable actions for the black-boxes $B_i, \cdots, B_k$). Our push-in technique works in the following $k$ steps, where $i$ is from 1 to $k$:

**Step i.** The step consists of two tasks:

(Automaton Generation) This task generates the unit test sequence automaton $U_i$ and the auxiliary automaton $A_i$. We first generate the auxiliary automaton $A_i$. Initially when $i = 1$, the generation is based on the Sys’s description (i.e., the gluer $G$ and the interfaces for $B_1, \cdots, B_k$) and the given set Bad. When $i > 1$, the generation is based on the auxiliary automaton $A_{i-1}$ and the surviving set $SUV_{i-1}$ (see below) obtained from the previous **Step i−1**. If the empty string is accepted by the auxiliary automaton $A_i$, then the global testing problem (none of observable behaviors of the system Sys is in Bad) returns “no” (i.e., a bad behavior of the system exists) — no further steps need to run. We then generate the unit test sequence automaton $U_i$ directly from the auxiliary automaton $A_i$ constructed earlier. This task is purely automata-theoretic and does not involve any testing.
(Surviving Set Generation) In this second task, using **BBtest**, we perform unit testing over the black-box $B_i$ for all test sequences accepted by the test sequence automaton $U_i$ ($U_i$ always accepts a finite set). We use $SUV_i$, called the surviving set, to denote all the successful test sequences. If the surviving set is empty, then the global testing problem returns “yes” (i.e., none of observable behaviors of the system $Sys$ is in $Bad$). Otherwise, if $i < k$ (i.e., it is not the last step), we goto the following **Step** $i + 1$. If $i = k$ (i.e., it is the last step and the surviving set is not empty), then the global testing problem returns “no” (i.e., some observable behaviors of the system $Sys$ is indeed in $Bad$).

In the rest of this section, we will clarify how Automata Generation and Surviving Set Generation in the $k$ steps can be done. Since our technique heavily depends on automata theory, we would like to first build the theory foundation of our technique before we proceed further.

### 5.2.1 Theory Foundation of the Push-in Technique

Let us first make a pessimistic (the name is borrowed from the discussions in [35]) modification of the original system $Sys$ by assuming that each black-box $B_i$, $1 \leq i \leq k$, can demonstrate any observable behavior in $\Sigma_i^*$ (recalling that $\Sigma_i$ is the interface of the black-box). The resulting system is denoted by $\tilde{Sys}$. Clearly, every observable behavior of $Sys$ is also an observable behavior of $\tilde{Sys}$ (but the reverse is not necessarily true).

Notice that $\tilde{Sys}$ does not have any black-boxes since the original black-box $B_i$, after the pessimistic modification, can be considered as a finite-state component $\hat{B}_i$ with only one state, where each action in $\Sigma_i \cup \{\epsilon\}$ is a label on a transition from the state back to the state. According to the semantics definition presented in Chapter 2, $\tilde{Sys}$ itself, after the composition of the gluer $G$ with all the one-state components $\hat{B}_1, \cdots, \hat{B}_k$, is a finite-state transition system with $|G|$ (the number of states in the gluer) states and with actions in $\Sigma \cup \{\epsilon\}$. (Recall that $\Sigma = \Sigma_0 \cup \cdots \cup \Sigma_k$ is the union of all observable actions in the gluer and the black-boxes.) The pessimistic system can also be treated as a pessimistic (finite) automaton by making each state
be an accepting state and each $\epsilon$-transition be an $\epsilon$-move. In this way, the language (a subset of $\Sigma^*$) accepted by the automaton is exactly all of the observable behaviors of the pessimistic system.

As we have mentioned earlier, the set $Bad \subseteq \Sigma^*$ is a finite and hence regular set. Suppose that the symbolic representation of the set is given as an automaton $M_{Bad}$ (whose state number is written $|M_{Bad}|$); i.e., the language accepted by $M_{Bad}$ is exactly the set $Bad$.

Using a standard Cartesian product construction, one can build an automaton $M_{global}$, called the global test sequence automaton, to accept the intersection of the language accepted by the pessimistic automaton $Sys$ and the language accepted by the automaton $M_{Bad}$. That is, $M_{global}$ accepts exactly the bad and observable behaviors of the pessimistic system. Clearly, the state number in $M_{global}$ is at most $|G| \cdot |M_{Bad}|$.

For a word $\alpha \in \Sigma^*$, we use $\alpha \downarrow_{\Sigma_i}$, $1 \leq i \leq k$, to denote the result of dropping all symbols not in $\Sigma_i$ from $\alpha$. That is, if $\alpha$ is an observable behavior of the system $Sys$, then $\alpha \downarrow_{\Sigma_i}$ is the corresponding observable behavior of black-box $B_i$. The theory foundation of our push-in technique can be summarized in the following theorem, which can be shown using the semantics defined in Chapter 2.

**Theorem 5.1** For any global test sequence $\alpha$ in $\Sigma^*$, the following two items are equivalent:

1. $\alpha$ is a bad (i.e., in $Bad$) observable behavior of the system $Sys$ of black-boxes $B_1, \cdots, B_k$, and

2. $\alpha$ is accepted by the global test sequence automaton $M_{global}$, and each of the following $k$ conditions holds:

   2.1. $\alpha \downarrow_{\Sigma_1}$ is an observable behavior of $B_1$,  
   
   $\vdots$  

   2. $\alpha \downarrow_{\Sigma_k}$ is an observable behavior of $B_k$.

We use “class C” to denote all the $\alpha$’s that satisfy Theorem 5.1 (2). Obviously, the global testing problem (i.e., there is no bad behavior in $Sys$) is equivalent to the emptiness of class C.
In the push-in technique, the jobs of Step 1, · · · , Step \( k \) are to establish the emptiness of class \( C \) using both automata theory and black-box testing. One naive approach for the emptiness is to use Theorem 5.1 (2) directly: repeatedly pick a global test sequence \( \alpha \) accepted by \( M_{global} \) (note that \( M_{global} \) accepts a finite language) and, using black-box testing, make sure that one of the conditions (2.\( i \)), \( 1 \leq i \leq k \), is false. This naive approach works but inefficiently. This is because, when \( M_{global} \) accepts a huge set (such as more than \( 10^{24} \) in our experiments shown later), trying every such element is not only infeasible but also unnecessary. Our approach of doing the job aims at eliminating the inefficiency. First, we do not pick a global test sequence \( \alpha \). Instead, we compute the test sequences run on black-box \( B_i \) from the testing results on black-box \( B_{i-1} \) in the previous Step \( i-1 \). As we have mentioned at the beginning of this section, each Step \( i \) has two tasks to perform: Automata Generation and Surviving Set Generation, which are presented in detail as follows.

5.2.2 Automata Generation in Step \( i \)

This task in Step \( i \) is to generate two automata: the unit test sequence automaton \( U_i \) and the auxiliary automaton \( A_i \).

Initially when \( i = 1 \), \( A_1 \) is constructed as \( A_1 = M_{global} \downarrow \Sigma_1 \cup \cdots \cup \Sigma_k \), i.e., the result of dropping every transition in \( M_{global} \) that is labeled with an observable action not in \( \Sigma_1 \cup \cdots \cup \Sigma_k \). \( U_1 \) is constructed as the automaton \( U_1 = A_1 \downarrow \Sigma_1 \) (i.e., the result of dropping every transition in \( A_1 \) that is labeled with an observable action not in \( \Sigma_1 \)). Observe that \( A_1 \) accepts the language \( A_1 = \{ \alpha \downarrow \Sigma_1 \cup \cdots \cup \Sigma_k : \alpha \text{ accepted by } M_{global} \} \) and \( U_1 \) accepts the language \( U_1 = \{ \alpha \downarrow \Sigma_1 : \alpha \text{ is in } A_1 \} \). The state number in either of the two automata, in worst cases, is \( |M_{global}| \).

When \( i > 1 \), the two automata \( A_i \) and \( U_i \) are constructed from the auxiliary automaton \( A_{i-1} \) and the surviving set \( SUV_{i-1} \) obtained in the previous step. To construct \( A_i \), we first build an automaton \( suv_{i-1} \) to accept the finite set \( SUV_{i-1} \). Then, we build an intermediate automaton \( M_{i-1} \) that works as follows: on an input word in \( (\Sigma_{i-1} \cup \cdots \cup \Sigma_k)^* \), \( M_{i-1} \) starts simulating \( A_{i-1} \) and \( suv_{i-1} \) on the word, in parallel. During the simulation, whenever \( suv_{i-1} \) reads an input symbol that is not in
\[ \Sigma_{i-1} \text{ (note that } suw_{i-1} \text{ only accepts words in } \Sigma^*_i\text{), it skips the input symbol. } M_{i-1} \text{ accepts the input word when both } A_{i-1} \text{ and } suw_{i-1} \text{ accept. Finally, the auxiliary automaton } A_i \text{ is constructed as } A_i = M_i \downarrow \Sigma_i \cup \cdots \cup \Sigma_k. \text{ The unit test sequence automaton } U_i \text{ is constructed as } U_i = A_i \downarrow \Sigma_i. \]

One can show that each of the two automata \( A_i \) and \( U_i \) has, in worst cases, a state number of \(|A_{i-1}| \cdot |suw_{i-1}|\). Also, \( A_i \) accepts the language \( A_i = \{ \alpha \downarrow \Sigma_i \cup \cdots \cup \Sigma_k \colon \alpha \in (\Sigma_i \cup \cdots \cup \Sigma_k)^* \text{ is in } A_{i-1} \text{ and } \alpha \downarrow \Sigma_{i-1} \text{ is in } SUV_{i-1} \} \) and \( U_i \) accepts the language \( U_i = \{ \alpha \downarrow \Sigma_i \colon \alpha \in (\Sigma_i \cup \cdots \Sigma_k)^* \text{ is in } A_i \} \).

As we have mentioned earlier, when the empty string is accepted by the auxiliary automaton \( A_i \) (a standard membership algorithm can be used to validate the acceptance), our push-in technique will return a “no” answer on the global testing problem (i.e., the system does have a bad observable behavior) and no further steps need to run.

### 5.2.3 Surviving Set Generation in Step \( i \)

The surviving set \( SUV_i \) is the set of all successful unit test sequences \( \alpha \in U_i \); i.e., \( SUV_i = \{ \alpha \in \Sigma^*_i : \alpha \in U_i \text{ and } \alpha \text{ is an observable behavior of black-box } B_i \} \).

A straightforward way to obtain the set is to run the black-box testing procedure \texttt{BBtest} over the black-box \( B_i \) with every test sequence in \( U_i \). This is, however, not efficient, in particular when the set \( U_i \) is huge. Observable behaviors of a component are prefix-closed: if \( \alpha \) is not an observable behavior of \( B_i \), then, for any \( \beta \), \( \alpha \beta \) can not be (i.e., test sequence \( \alpha \beta \) need not be run). With prefix-closeness and \texttt{BBtest}, we use the following automata-theoretic procedure to generate the surviving set \( SUV_i \).

Recall that \( U_i \) is a finite set of unit test sequences and, as a regular language, accepted by the unit test sequence automaton \( U_i \). Let \( m \) be the maximal length of all test sequences in \( U_i \) (the length can be obtained using a standard longest path algorithm over the transition graph of automaton \( U_i \)). Our procedure consists of the following \( m \) jobs. Each \texttt{Job}_j, where \( j \) is from 1 to \( m \), is to identify (using black-box testing) all the successful test sequences (with length \( j \)) which are prefixes (which are not necessarily proper) of some test sequences in \( U_i \). In order to do this efficiently,
the job makes use of the previous testing results in \( \Theta_{j-1} \). More precisely, each Job\(_j\) has two parts (by assumption, let \( \Theta_0 \) contain only the empty word.).

1) Define \( P_j \) to be the set of all the prefixes with length \( j \) of all the unit test sequences in \( \mathbb{U}_i \). Calculate the set \( \hat{P}_j \subseteq P_j \) such that each element in \( \hat{P}_j \) has a prefix (with length \( j - 1 \)) in \( \Theta_{j-1} \). To implement this part, one can first construct an automaton (from automaton \( U_i \)) to accept the language \( P_j \). Then, construct another automaton to accept the set \( \Theta_{j-1} \). Finally, an automaton \( M \) can be constructed from these two automata to accept the language \( \hat{P}_j \). All the constructions are not difficult and do not involve testing.

2) Using BBtest, generate the set \( \Theta_j \) that consists of all the successful test sequences over black-box \( B_i \) in \( \hat{P}_j \). Hence, one only runs test sequences in \( \hat{P}_j \) instead of the entire \( P_j \), thanks to the previous testing results in \( \Theta_{j-1} \).

It is left to the reader to verify that, after the jobs are completed, the surviving set \( SUV_i \) can be obtained as \( \mathbb{U}_i \cap (\cup_{0 \leq j \leq m} \Theta_j) \). Again, this set can be accepted by an automaton, treated as a symbolic representation of the set, constructed from automaton \( U_i \) and the automata built in the above jobs to accept \( \Theta_j \), \( 1 \leq j \leq m \). One can choose the procedure to output the explicit set \( SUV_i \) or its symbolic representation \( suv_i \).

### 5.2.4 Correctness and Bad Behavior Generation

Since the global testing problem is equivalent to the emptiness of class \( C \), we only need to show that the emptiness is answered correctly with the push-in technique. Clearly, the technique always terminates with a yes/no answer. It returns “yes” only at some Step \( i \), \( 1 \leq i \leq k \), whose surviving set \( SUV_i = \emptyset \). It returns “no” only

CASE1. at some Step \( i \), \( 1 \leq i \leq k \), when the auxiliary automaton \( A_i \) accepts the empty word, or

CASE2. at the last Step \( k \) when \( SUV_k \neq \emptyset \).

In these two cases, in order to demonstrate a global bad behavior of the system, we first define an operation called select\(_j(\cdot)\), \( 1 \leq j \leq k \). Given a sequence \( \alpha_j \), the
operation returns a sequence $\alpha_{j-1}$ (when $j = 1$, it simply returns $\alpha_j$) satisfying the following conditions: $\alpha_{j-1} \in A_{j-1}$, $\alpha_{j-1} \downarrow \Sigma_{j-1} \in SUV_{j-1}$ and $\alpha_{j-1} \downarrow \Sigma_{j-1} \cup \cdots \cup \Sigma_k = \alpha_j$. The returned sequence $\alpha_{j-1}$ may not be unique. In this case, any sequence (such as a shortest one) satisfying the conditions will be fine. Now, we define another operation called $\text{BadGen}_j(\cdot)$, $1 \leq j \leq k$, as follows. Given a sequence $\alpha_j$, we first calculate $\alpha_{j-1} = \text{select}_j(\alpha_j)$. Then, we calculate $\alpha_{j-2} = \text{select}_{j-1}(\alpha_{j-1})$, and so on. Finally, we obtain $\alpha_1$. At this time, the operation $\text{BadGen}_j(\alpha_j)$ returns any sequence $\alpha$ satisfying the following conditions: $\alpha$ is accepted by $M_{\text{global}}$ and $\alpha \downarrow \Sigma_1 \cup \cdots \cup \Sigma_k = \alpha_1$. All these operations can be easily implemented through automata constructions.

Coming back to bad behavior generation, in CASE1, we return $\text{BadGen}_1(\lambda)$ (where $\lambda$ is the empty sequence) as a global bad behavior. In CASE2, we simply pick any sequence $\alpha_k$ from $SUV_k$ and return $\text{BadGen}_k(\alpha_k)$ as a global bad behavior.

One can show that our technique is indeed correct.

**Theorem 5.2** If the class $C$ is empty then the push-in technique returns “yes”, otherwise it returns “no”. When the technique returns yes, it shows that the system doesn't have any of the global bad behaviors in Bad, otherwise it indicates that the system does exhibit bad behaviors in Bad.

In each step of our algorithm, one can use standard algorithms in automata theory to make the obtained automata like $U_i$'s and $A_i$’s smaller. The algorithms include eliminating unreachable states and/or minimization. Additionally, the algorithms as well as all the automata constructions mentioned in the push-in technique can be implemented using existing automata manipulation tools like Grail [1].

From the correctness theorem, we know that the push-in technique is sound and complete. However, one question still remains unsolved: Are test sequences (for black-box $B_i$) in each $U_i$ more than necessary (in solving the global testing problem)? We can show that each $U_i$ derived from our push-in technique is “optimal” in the following sense. Suppose that we have completed the first $i - 1$ steps (i.e., the black-boxes $B_1, \cdots, B_{i-1}$ have been tested) and have obtained $U_i$ to start the subsequent steps (i.e., the remaining black-boxes $B_i, \cdots, B_k$ are not tested yet). Each test sequence $\alpha_i$ in $U_i$ has to be run, since one can show the following two statements: there are
black-boxes $B_i^*, \cdots, B_k^*$, such that $\alpha_i$ is a successful (resp. unsuccessful) test sequence for $B_i^*$ and the system $G(B_1, \cdots, B_{i-1}, B_i^*, \cdots, B_k^*)$ has (resp. does not have) a global bad behavior.

<table>
<thead>
<tr>
<th>Case</th>
<th>$\lambda = 0$</th>
<th>$\lambda = 20$</th>
<th>$\lambda = 30$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00x10^4</td>
<td>148</td>
<td>47</td>
</tr>
<tr>
<td>2</td>
<td>1.12x10^6</td>
<td>320</td>
<td>73</td>
</tr>
<tr>
<td>3</td>
<td>1.12x10^6</td>
<td>320</td>
<td>73</td>
</tr>
<tr>
<td>4</td>
<td>1.03x10^6</td>
<td>178</td>
<td>32</td>
</tr>
</tbody>
</table>

Table 5.1: Experiment Results: Counts of Test Sequences

5.3 Experiments

All the experiments were performed on a PC with a 800MHz Pentium III CPU and 128MB memory. The Grail [1] tool was used to perform almost all the automata operations\(^2\). The entire experiment process was driven by a Perl script and carried out automatically. Our experiments were run on the system of black-boxes shown in Figure 2.1. In the experiments, we designated black-boxes Timer, Sensor and Comm as $B_1$, $B_2$, and $B_3$, respectively. The internal implementations of the black-boxes are shown in Figures 2.4, 2.5 and 2.6, on which the unit testing in the experiments was performed. We have totally run twelve experiments (each experiment is a complete execution of the push-in technique), which are divided into four cases. Each of the four cases consists of three experiments, which are illustrated in detail as follows.

**Case 1** Firstly, we wish that whenever a *pause* event takes place, there should be no more *send* until a *resume* occurs. The corresponding bad behaviors are specified as a regular expression, $\Sigma^* p(\Sigma - \{r\})^* s \Sigma^*$, where $\Sigma$ is the set of all the twelve events in the system; $p$, $r$, and $s$ stand for the *pause*, *send*, and *resume*, respectively (such

\(^2\)We implemented (in C) three additional operations to manipulate automata with $e$-moves and to count the number of words in a finite language accepted by an automaton, which are not provided in Grail.
CHAPTER 5. DECOMPOSITIONAL TESTING

abbreviation will be used throughout this section). For the first experiment run in this case, we chose the Bad to be all words in the regular expression that are not longer than 10 (denoted by “maxlength=10”). The remaining two experiments were run with “maxlength=20” and “maxlength=30”, respectively. To understand the results shown in Table 5.2.4, we go through the third experiment (i.e., “maxlength=30”). The results of the experiment are shown in the box at the right upper corner in the table (i.e., under the four columns associated with “maxlength=30” and in the three rows (“step_1”, “step_2”, “step_3”) associated with “case 1”). The three steps in the experiment correspond to the three Steps (since there are three black-boxes) in the push-in technique. The auxiliary automaton $A_1$ calculated in Step 1 accepts totally $#A_1 = 2.16 \times 10^{24}$ test sequences. The unit test sequence automaton $U_1$ accepts $#U_1 = 4.14 \times 10^7$ test sequences. Using the black-box testing procedure in Section 5.2.3, we actually only performed $TC_1 = 2.87 \times 10^5$ unit tests over $B_1$ (the Timer), among which $#SUV_1 = 2.23 \times 10^5$ tests survived. In Step 2 and Step 3, we obtained $#A_2, #U_2, #A_3, #U_3$ similarly as shown in the table. In particular, we actually performed $TC_2 = 2940$ unit tests over the Sensor in Step 2 and $TC_3 = 1577$ unit tests over the Comm in Step 3. Since the last surviving set $SUV_3$ is not empty ($#SUV_3 = 274$), the experiment detects a global bad behavior specified in this case.

Notice that the total number of unit tests run in this experiment is $TC_1 + TC_2 + TC_3$, which is not more than $2.92 \times 10^5$. This number essentially indicates the actual “cost” of the experiment in deciding whether there is a global bad behavior specified in the case and whose length is bounded by 30. This number is quite good considering the astronomical number $#A_1 = 2.16 \times 10^{24}$ which would be the number of integration test sequences if one run integration testing, since $M_{global} = A_1$ in the system. The other two experiments (“maxlength=10” and “maxlength=20”) also detected a global bad behavior and results are shown in the first three rows under “maxlength=10” and “maxlength=20” in Table 5.2.4 (the costs of these two experiments, which are 148 and 5262 respectively, become much smaller).

Case 2 The detected bad behaviors are due to the concurrency nature of these black-boxes: a fire was issued before the pause is sent to Timer, which eventually leads to another send. For instance, a global bad behavior could be like the following:
fire data send msg fire data send crrr fire data pause send. From this observation, we believed that the system might also have other bad behaviors: after a crrr takes place, there could be another crrr coming before a resume occurs. Such bad behaviors are encoded by $\Sigma^* c(\Sigma - \{r\})^* c\Sigma^*$. The three experiments in this case, however, did not detect such bad behaviors (i.e., $#SUV_3 = 0$ for all lengths, shown in the third row “step$_3$” associated with “case 2” in Table 5.2.4).

Case 3 Based upon the experiments in the previous case, we carefully studied the system and realized that the implementation of Comm might be wrong: after an error occurs (i.e., a crrr outputs), Comm is supposed to retain its state prior to the output of the crrr, while it does not. After correcting this bug (by making the internal implementation of Comm, shown in Figure 2.6, move to state $s2$ instead of $s0$ after a crrr is output), in this case, we run the three experiments again. The experiments detected bad behaviors only with length more than 10 (i.e., $#SUV_3 = 0$ when maxlength is 10 and $#SUV_3 > 0$ when maxlength is 20 and 30, shown in Table 5.2.4).

Case 4 Now we want to test that: after an error occurs in Sensor (i.e., a serr is issued), there will be at most one more fire issued before a resume occurs. The corresponding bad behaviors are encoded by $\Sigma^* serr(\Sigma - \{r\}) f(\Sigma - \{r\})^* f(\Sigma - \{r\})^* r\Sigma^*$, where $f$ stands for fire. Our experiments did not detect any such behaviors for all the three choices of maxlength: 10, 20, 30. In fact, in the experiments, no testing over Comm was needed. This is because, as shown in the last three rows of Table 5.2.4, $#SUV_2$ is 0 for all the three choices.

We measured the total time that our script used for automata manipulations in each of the twelve experiments, shown in Table 5.2. In the table, the “result” shows whether a global bad behavior was detected in an experiment; i.e., “x” (resp. “√”) indicates “detected” (resp. “not detected”). As shown in the table, the total time is within a minute for all the four experiments with “maxlength=10”. For “maxlength=20”, the time is still acceptable (within an hour). When the maxlength is increased to 30, the time is still within our patience (which was set to be 24 hours). Yet, our script could not finish within the time limit for any experiment when we tried to push maxlength to 40. Even though determinization and minimization are optional
in our push-in technique, we made them mandatory in our experiments. In this way, we can cross-compare the sizes of the automata obtained in each step of the experiments. The largest size of all the automata constructed in the twelve experiments, after determinization and minimization, is with 726 states and 2138 transitions. In an experiment with maxlen=40, the script tried to make an automaton (with 1182 states) deterministic and failed to do so within our patience.

Exhaustive integration testing over a concurrent system is in general infeasible. However, the experiments show that, using the push-in technique, we can completely solve the global testing problem with a substantially smaller number of tests over each individual black-box only, even for an extremely large set of Bad. For instance, the total number of unit tests ($TC_i$'s) performed in each of the four experiments with “maxlen=30” is in the order of $10^5$, while each Bad is in the order of $10^{24}$ (notice that each Bad is always larger than each #A_1, shown in Table 5.2.4).

<table>
<thead>
<tr>
<th>Cases</th>
<th>maxlen=10</th>
<th>maxlen=20</th>
<th>maxlen=30</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>result</td>
<td>time</td>
</tr>
<tr>
<td>Case 1</td>
<td>~25s</td>
<td>×</td>
<td>~40m</td>
</tr>
<tr>
<td>Case 2</td>
<td>~34s</td>
<td>✓</td>
<td>~58m</td>
</tr>
<tr>
<td>Case 3</td>
<td>~36s</td>
<td>✓</td>
<td>~56m</td>
</tr>
<tr>
<td>Case 4</td>
<td>~17s</td>
<td>✓</td>
<td>~22m</td>
</tr>
</tbody>
</table>

Table 5.2: Experiment Results: Time Efficiency

5.4 Summary

In this chapter, we presented an automata-theoretic and decompositional technique (the "push-in" technique) to address a global testing problem; i.e., testing a system of concurrent black-boxes against a finite set of bad behaviors. Our technique is automatic, sound, and complete.

Essentially, the global testing problem is a verification problem since we are looking for a definite answer. In the area of formal verification, there has been a long
CHAPTER 5. DECOMPOSITIONAL TESTING

history of research on exploiting compositionality in system verification, and a common technique is to follow the “assume-guarantee” reasoning paradigm [65, 81, 58, 26, 3, 35, 29, 5]. However, a successful application of the paradigm depends on the correct assumptions for the components in a system, which are, in general, formulated manually. Several researchers suggest solutions to the problem of automated assumption generation [50, 57, 41, 43]. But the solutions require that the source code and/or the finite-state design for a component is available, which, unfortunately, is not the case in our setup. Although our push-in technique relies on black-box testing instead of an “assume-guarantee” mechanism, it can be extended to a system where a black-box is associated with environmental assumptions.

The heart of our “push-in” technique is based on an observation that global behaviors of a concurrent system can be projected onto behaviors for each constituent component. Moreover, the behaviors of each individual component are also constrained by the behaviors of other components (because of synchronizations). The similar observations have also been made in [24, 25] as Context Constraints for compositional reachability analysis and used in [61, 60] for structural testing of concurrent programs.

Our technique can be generalized to many other forms of bad behavior specifications (i.e., the finite set Bad). For instance, we may that specify that Bad consist of all observable sequences not longer than 40, each of which can make the gluer enter a given (undesired) state. But the exact formalisms for bad behavior specifications need further investigation. Our model of the system is based on synchronized communications. Therefore, it would be interesting to see whether the approach can be generalized to some forms of asynchronous (e.g., shared-variable) systems. Black-boxes in our model are event-driven; it is also worthwhile to study other decompositional testing approaches for data-driven black-boxes. Sometimes, our push-in technique fails to complete, due to an extremely large bad behavior set Bad (e.g., our experiments with “maxlength=40” shown earlier, whose global test sequences deduced from Bad are roughly as many as $10^{33}$). In this case, we need study methods to (symbolically) partition the set into smaller subsets such that the push-in technique can be run over each smaller subset. In this way, a global bad behavior could instead be found. In
our definition of the push-in technique, there is not a pre-defined ordering in testing the black-boxes. For instance, in our experiments, the ordering was Timer, Sensor, Comm, based on the size of a black-box’s interface. Clearly, more studies are needed to clarify the relationship between the efficiency of our technique and the choices of the ordering.
Chapter 6

The Linear Reachability Problem of Finite-state Labeled Transition Systems\textsuperscript{1}

Although both CTL and LTL are expressive, many temporal properties are out of their scope. For instance, event counting is a fundamental concept to specify some important fairness properties. So, to study the automatic verification of a component-based system, it is worthwhile to study how to verify a single, finite-state, and specified component against some non-temporal property.

The rest of this chapter is organized as follows. Section 6.1 gives an overview of this study. Section 6.2 introduces some known results on minimal solutions to linear Diophantine equation systems, which are needed later in the chapter. In Section 6.3, we obtain a bounding box for the linear reachability problem for finite-state labeled transition systems. Based on the bounding box, Section 6.4 establishes a time complexity bound for the linear liveness problem for finite-state labeled transition systems. Section 6.5 is a brief summary.

\textsuperscript{1}The content of this chapter is based upon the joint work with C. Li and Z. Dang in [96]
CHAPTER 6. THE LINEAR REACHABILITY PROBLEM

6.1 Introduction

As a motivating example, we consider the design of a process scheduler, depicted as a finite-state labeled transition system $A$ in Figure 6.1. The scheduler schedules two kinds of processes: $P_r$ and $P_w$ according to some scheduling strategy. A transition with label $P_r$ (resp. $P_w$) is taken when the scheduler chooses a $P_r$ (resp. $P_w$) process to run. It is required that the design shall satisfy some fairness properties; e.g., starting from state $s_0$, whenever $s_0$ is reached, the number of $P_r$ processes scheduled is greater than or equal to the number of $P_w$ processes scheduled and less than or equal to twice the number of $P_w$ processes scheduled. To ensure that the design meets the requirement, we need to check whether for any path $p$ that starts from and ends with $s_0$, the linear constraint, $\#_{P_r}(p) \leq \#_{P_w}(p) \leq 2\#_{P_w}(p)$, is satisfied, where $\#_{P_r}(p)$ (resp. $\#_{P_w}(p)$) stands for the count of labels $P_r$ (resp. $P_w$) on path $p$. Notice that this property is non-regular [14] and, since the counts could go unbounded, the property is not expressible in CTL or LTL.

In general, by considering its negation, the property can be formulated as the linear reachability problem for finite-state labeled transition systems (FLTSs) as follows.

- **Given:** A finite-state labeled transition system $A$ with labels $a_1, \ldots, a_k$, two designated states $s_{init}$ and $s_{final}$, and a linear constraint $U(x_1, \ldots, x_k)$.

- **Question:** Is there a path $p$ of $A$ from $s_{init}$ to $s_{final}$ such that $p$ satisfies $U$ (i.e., $U(\#_{a_1}(p), \ldots, \#_{a_k}(p))$ holds)?

The reachability problem is decidable. To see this, one can treat $A$ as a finite automaton with initial state $s_{init}$ and final state $s_{final}$. Then a naive decision procedure can be constructed in the following three steps: (i). Compute a regular expression for the regular language (over alphabet $\{a_1, \ldots, a_k\}$) accepted by $A$. (ii). Calculate the semi-linear set of the regular expression defined by a Presburger formula $R$ [76], and (iii). Check the satisfiability of the Presburger formula $R \land U$. Unfortunately, the time complexity of this procedure is at least $O(2^{|S|})$, where $|S|$ is the number of

\footnote{In this chapter, we do not distinguish labels for internal actions, input actions, and output actions.}
states in $A$, even when $k$ is fixed. This is because the size of the regular expression, in worst cases, is exponential in $|S|$ [53].

![Diagram]

Figure 6.1: An Example of A Scheduler

In this chapter, we present a new algorithm solving the linear reachability problem. This algorithm is completely different from the naive one. In our algorithm, we estimate a bound $B$ (called a bounding box) from $A$ and $U$ such that, the Question-part is true iff the truth is witnessed by some $p$ on which the count $#a_i(p)$ for each label $a_i$ is bounded by $B$. Interestingly, after a complex loop analysis, estimating a bounding box $B$ is reduced to a number theory problem: finding nonnegative minimal solutions to linear Diophantine equation systems. There has been much research on this latter problem for homogeneous/inhomogeneous systems with (nonnegative) integer solutions [12, 13, 37, 82]. Suppose that $U$ is in a disjunctive normal form over linear equations/inequalities. Using the Borůvka-Flahive-Treybig bound in [12], we are able to show that, in worst cases, when $|S|$ is $\gg$ the size (which will be made clear later in the chapter) of $U$, the bounding box $B$ is bounded by $O(|S|^{k+L+3})$, where $L$ is the maximal number of conjunctions in a single disjunctive term of $U$. (We assume that $U$ is written in the disjunctive normal form over atomic linear constraints such as $2x_1 - 3x_2 + 4x_3 \geq 5$.) The Borůvka-Flahive-Treybig bound has been used in solving the boundedness problem for vector addition systems [83]. However, the path rearrangement technique used in [83] is not applicable (at least not in an easy way) to obtaining the bounding box $B$. With the bounding box, one can easily show that the linear reachability problem is solvable in time $O(|S|^{2(k+L+3)+2})$, when $|S| \gg k$ and the size of $U$. In particular, when $k$ and $U$ are fixed, the complexity is polynomial in $|S|$. This is in contrast to the complexity of the naive algorithm that is exponential in the state number $|S|$. This new complexity result will be further used in this chapter.
to obtain complexity bounds (which were unknown) for some other linear counting
problems that involve linear constraints over counts, e.g., the linear liveness problem
[34] for A.

6.2 Definitions

First, we slightly modify Definition 2.1 such that a finite-state labeled transition system
A is defined as

\[ A = \langle S, Init, \nabla, R \rangle, \]

where \( S \) is a finite set of states, \( Init \subseteq S \) is a set of initial states, \( \nabla = \{ a_1, \ldots, a_k \} \) is a
set of labels, \( R \subseteq S \times (\nabla \cup \{ \epsilon \}) \times S \) is the transition relation. Notice that, here we do
not distinguish labels for internal actions, input actions and output actions, we also
allow the label of a transition to be empty (i.e., labeled with \( \epsilon \)). When \( R \subseteq S \times \{ \epsilon \} \times S \),
A is called a finite-state machine. A path \( p \) of A is a finite sequence of alternating
states and labels \( s_0 \tau_0 s_1 \ldots s_{n-1} \tau_{n-1} s_n \) for some \( n \) such that for each \( 0 \leq i < n, \)
\((s_i, \tau_i, s_{i+1}) \in R \). Path \( p \) is a simple cycle if \( s_0, \ldots, s_{n-1} \) are distinct and \( s_0 = s_n \).
Path \( p \) is a simple path if \( s_0, \ldots, s_{n-1}, s_n \) are all distinct. For any path \( p \) of A, recall
that \( \#(p) \) denotes the Parikh map of \( p \) (i.e., the \( k \)-ary vector \((\#_{a_1}(p), \ldots, \#_{a_k}(p))\),
where each \( \#_{a_i}(p) \) stands for the number of label \( a_i \)'s occurrences on \( p, 1 \leq i \leq k \)).

Let \( U \) be a linear constraint defined in Section 2.4. Without loss of generality,
throughout this chapter, we assume that \( U \) is written as a disjunction \( U_1 \vee \ldots \vee U_m \), for some \( m \), of conjunctions of atomic linear constraints. When \( m = 1 \), \( U \)
is called a conjunctive linear constraint. \( U \) is made homogeneous if each atomic
linear constraint in \( U \) is made homogeneous; we use \( U^{\text{hom}} \) to denote the result. In
particular, a conjunctive linear constraint \( U \) is a linear Diophantine equation system
if each atomic linear constraint in \( U \) is an equation.

Suppose that \( U \) is a conjunctive linear constraint, which contains \( e \) equations and
\( l - e \) inequalities. One may write \( U \) into \( Bx \sim b \), where \( \sim \in \{ =, \geq \}^l \), \( B \) (\( l \) by \( k \))
and \( b \) (\( l \) by 1) are matrices of integers, and \( x \) is the column of variables \( x_1, \ldots, x_k \).
As usual, \((B, b)\) is called the augmented matrix of \( U \), and \( B \) is called the coefficient
matrix of \( U \). We use \( ||B||_{1,\infty} \) to denote \( \max_i\{\sum_j |b_{ij}|\} \) (\( b_{ij} \) is the element at row \( i \) and column \( j \) in \( B \)) and use \( ||b||_\infty \) to denote the maximum of the absolute values of all the elements in \( b \). Assume \( r \) is the rank of \((B,b)\), and \( \Gamma_1 \) (resp. \( \Gamma_2 \)) is the maximum of the absolute values of all the \( r \times r \) minors of \( B \) (resp. \((B,b)\)).

When \( U \) is a linear Diophantine equation system (i.e., \( e = l \), for any given tuples \((v_1,\ldots,v_k)\) and \((v'_1,\ldots,v'_k)\) in \( \mathbb{N}^k \), we say \((v_1,\ldots,v_k) \leq (v'_1,\ldots,v'_k)\) if \( v_i \leq v'_i \) for all \( 1 \leq i \leq k \). We say \((v_1,\ldots,v_k) < (v'_1,\ldots,v'_k)\) if \((v_1,\ldots,v_k) \leq (v'_1,\ldots,v'_k)\) and \( v_i < v'_i \) for some \( 1 \leq i \leq k \). A tuple \((v'_1,\ldots,v'_k)\) is a minimal solution to \( U \) if \((v'_1,\ldots,v'_k)\) is a solution to \( U \) but any \((v_1,\ldots,v_k)\) with \((0,\ldots,0) < (v_1,\ldots,v_k) < (v'_1,\ldots,v'_k)\) is not. Clearly, there are only finitely many minimal solutions to \( U \). It has been an active research area to estimate a bound for minimal solutions, and the following Borosh-Flahive-Treybig bound \([12]\) is needed in this chapter.

Theorem 6.1 (Borosh-Flahive-Treybig bound) A linear Diophantine equation system \( U \) has solutions in nonnegative integers iff it has a solution \((x_1,\ldots,x_k)\) in nonnegative integers, such that \( r \) unknowns are bounded by \( \Gamma_1 \) and \( k - r \) unknowns are bounded by \((\max(k,l) - r + 1)\Gamma_2\).

The Borosh-Flahive-Treybig bound gives a bound for one of the minimal solutions in nonnegative integers to the inhomogeneous system \( U \).

An inequality can be translated into an equation by introducing a slack variable (e.g., \( x_1 - 2x_2 \geq 3 \) into \( x_1 - 2x_2 - u = 3 \), where \( u \), a new variable on \( \mathbb{N} \), is the slack variable). So if \( U \) is a conjunctive linear constraint (in which there are \( e \) equations and \( l - e \) inequalities) over \( x_1,\ldots,x_k \), we may write \( U \) into an equation system \( U(x_1,\ldots,x_k, y_1,\ldots,y_{l-e}) \) with \( l \) equations, where \( y_1,\ldots,y_{l-e} \) are the slack variables.
6.3 A Bounding Box for the Linear Reachability Problem

Let $A$ be a finite-state labeled transition system specified in (6.1). A set $Q \subseteq \mathbb{N}^k$ is a small linear set (with respect to the given $A$) if $Q$ is in the form of

$$\{e_0 + \sum_{1 \leq j \leq r} X_j e_j : \text{each } X_j \geq 0\},$$

where nonnegative integer $r$ satisfies $r \leq |S|^k$, $k$-ary nonnegative integer vectors $e_0, \ldots, e_r$ satisfy $|e_0|_\infty \leq |S|^2$, and for each $j = 1, \ldots, r$, $|e_j|_\infty \leq |S|$. $Q$ is a small semi-linear set if it is a union of finitely many small linear sets.

Recall that the linear reachability problem for $A$ is to decide whether there exists a path $p$ in $A$ from $s_{\text{init}}$ to $s_{\text{final}}$ such that $p$ satisfies a given linear constraint $U(x_1, \ldots, x_k)$. Let $\mathcal{P}$ be all paths of $A$ from $s_{\text{init}}$ to $s_{\text{final}}$. We use $\#(\mathcal{P})$ to denote the set of $k$-ary nonnegative integer vectors $\{\#(p) : p \in \mathcal{P}\}$. Using a complex loop analysis technique to reorganize simple loops on a path, one can show that $\#(\mathcal{P})$ is a small semi-linear set.

**Lemma 6.2** $\#(\mathcal{P})$ is a small semi-linear set. That is, it can be represented as, for some $t$, \(^3\)

$$\#(\mathcal{P}) = \bigcup_{1 \leq i \leq t} Q_i,$$

where each $Q_i$ is a small linear set in the form of (6.2).

**Proof.** Let $p$ be a path $s_0s_1s_2\ldots s_{n-1}s_n$ of $A$. We use $|p| = n$ to denote the length of $p$, $S_p$ to denote the set of states appearing on $p$, and $p^i$ to denote the prefix of $p$ whose length is $i$. Obviously, $|p| \leq |S|$ when $p$ is a simple cycle, and $|p| < |S|$ when $p$ is a simple path. Path $p$ passes a state $s$ whenever $s \in S_p$. Given two paths $p_1$ and $p_2$, we use $S_{p_1 \cap p_2}$ to denote $S_{p_1} \cap S_{p_2}$, which is the set of all the states that appear on both $p_1$ and $p_2$. If $S_{p_1 \cap p_2} \neq \emptyset$, we say that $p_1$ touches $p_2$ with touch states $S_{p_1 \cap p_2}$. Otherwise, we say that $p_1$ does not touch $p_2$.

---

\(^3\)Note that though $t$ may be large, it is irrelevant here.
Then we can extract (as shown in Algorithm 1), from \( p \), a simple path \( p_0 \) (called the \textit{basic path} of \( p \)) and a set \( C_p \) of simple cycles. It can be observed that the stack content (when reading from bottom to top) does not contain any simple cycles at any moment and hence the basic path \( p_0 \) obtained in the last step is indeed a simple path. Define \( \Delta_0 = S_{p_0} \cup \{s_0,s_n\} \). In particular, if \( p_0 \) is empty, then \( s_0 \) must be \( s_n \) (i.e., \( p \) itself forms a cycle), else \( s_0 \in S_{p_0} \) and \( s_n \in S_{p_0} \) (i.e., \( \Delta_0 = S_{p_0} \)). \( \Delta_0 \) is called the \textit{basic states}.

**Algorithm 1**

1. Initialize a stack \( ST \) and a set \( C_p \) to be empty;
2. Scan \( p \) from left to right;
3. for each transition \( e = (s_i,\tau_i,s_{i+1}) \) on \( p \) do
   1. if \( s_i = s_{i+1} \) (i.e., \( e \) itself is a simple cycle) then
      1. \( C_p := C_p \cup \{e\} \);
   else
      1. Check whether \( ST \), from top to bottom, has an element \( e' = (s,\tau,s') \) with \( s = s_{i+1} \);
      2. if yes then
         1. Pop all the elements above \( e' \) and \( e' \) itself from the stack;
         2. The popped elements together with \( e \) form a simple cycle \( c \);
         3. \( C_p := C_p \cup \{c\} \);
      else
         1. Push \( e \) into \( ST \);
   end if
4. \( p_0 \) is obtained by concatenating the remaining elements in \( ST \) from bottom to top.

Next, we partition \( C_p \) into subsets (called \textit{layers}) \( L_1,\ldots,L_m \) for some \( m \) as follows. The first layer \( L_1 \) is the set of all the simple cycles \( c \) in \( C_p \) such that \( c \) passes a state in \( \Delta_0 \); i.e., \( L_1 = \{c : c \in C_p \text{ and } S_c \cap \Delta_0 \neq \emptyset\} \). Define \( \Delta_1 = \cup_{c \in L_1} S_c \) and \( T_1 = \cup_{c \in L_1}(S_c \cap \Delta_0) = \Delta_1 \cap \Delta_0 \). \( \Delta_1 \) is the set of all the states that are passed by simple cycles in \( L_1 \). \( T_1 \) contains exactly all the touch states between \( p_0 \) and a simple cycle in \( L_1 \). In general, for \( i \geq 2 \), \( L_i \) is the set of all the simple cycles \( c \in C_p \) such that \( c \) has not been grouped into layers \( L_1,\ldots,L_{i-1} \) and \( c \) touches some simple cycle in \( L_{i-1} \); i.e., \( L_i = \{c : c \in C_p - \cup_{1 \leq j \leq i-1} L_j \text{ and } S_c \cap \Delta_{i-1} \neq \emptyset\} \). \( \Delta_i \) is the set of all the states that are passed by simple cycles in \( L_i \); i.e., \( \Delta_i = \cup_{c \in L_i} S_c \). \( T_i \) is the set of all the touch states between a simple cycle in \( L_{i-1} \) and a simple cycle in \( L_i \);
i.e., $T_i = \bigcup_{c \in L_i} (S_c \cap \Delta_{i-1}) = \Delta_i \cap \Delta_{i-1}$. It is easy to observe that, according to the above definitions, $L_i \cap L_j = \emptyset$ whenever $i \neq j$, $\Delta_i \cap \Delta_j = \emptyset$ whenever $|i - j| \geq 2$, and $T_i \cap T_j = \emptyset$ whenever $i \neq j$. In particular, since $T_i = \emptyset$ iff $L_i = \emptyset$, $T_i = \emptyset$ implies $T_{i+1} = \emptyset$. Obviously, since each $T_i \subseteq S$, there exists some value $m \leq |S|$ such that $L_1, \ldots, L_m \neq \emptyset$ but $L_{m+1} = \emptyset$. That is, the number of layers is bounded and the bound is independent of the choice of $p$. We call the tuple $\langle p_0, L_1, \ldots, L_m, T_1, \ldots, T_m \rangle$ the layered structure $\mathbb{L}_p$ of path $p$.

For instance, consider a path $p$ of the transition system in Figure 6.1 that passes through the states (in this order): $s_0 s_4 s_5 s_3 s_1 s_5 s_3 s_1 s_2 s_3 s_0 s_1 s_5 s_3 s_0 s_4$. After running Algorithm 1, we can obtain a basic path $p_0 : s_0 s_4$ and four simple cycles (the labels are omitted for simplicity), $c_1 : s_5 s_3 s_1 s_5$, $c_2 : s_3 s_1 s_2 s_3$, $c_3 : s_0 s_4 s_5 s_3 s_0$, and $c_4 : s_0 s_1 s_5 s_3 s_0$. From the above definitions, they are arranged into two layers as shown in Figure 6.2. In particular, $T_1 = \{s_0, s_4\}$ and $T_2 = \{s_1, s_3, s_5\}$ are indeed disjoint. Now, suppose that we are given a layered structure $\mathbb{L}_p$, then how can we obtain the path $p$? Hereafter in this chapter, we will use formulas in the form of $p_0 + \sum_{c \in C_p} X_c c$, $X_c \geq 0$, to stand for those paths obtained from $\mathbb{L}_p$ by traversing the basic path $p_0$ once, and each simple cycle $c \in C_p$ for $X_c$ times during the traversal of $p_0$. Obviously, constraints must be put over these $X_c$’s to ensure that we can always obtain a path of the corresponding transition system. For instance, consider the layered structure in Figure 6.2. In order to obtain $p$, each of $c_i$ ($i = 1, 2, 3, 4$) must be traversed at least once (though, for now, we are not interested in the exact numbers of traversals) during the traversal of the basic path $p_0$. Failing to do so will not allow us to obtain a path; e.g., $p_0 + 2c_1 + 3c_2 + 0c_3 + 0c_4$ corresponds to no path of the transition system in Figure 6.1 at all. For a layered structure $\mathbb{L}_p$ of a path $p$ of any finite-state labeled transition system $\mathbb{A}$, we define $\text{Span}(\mathbb{L}_p)$ as the set of paths obtained by traversing $p_0$ for once, and traversing each simple cycle in every layer for at least once. That is, $\text{Span}(\mathbb{L}_p)$ is the set $\{q_0 + \sum_{c \in C_p} X_c c : each \ X_c \geq 0\}$, where $q_0 = p_0 + \sum_{c \in C_p} c$. Clearly, each path in $\text{Span}(\mathbb{L}_p)$ is indeed a path of $\mathbb{A}$, and

\footnote{As we are only interested in the counts information of a path, the order in which these cycles should be traversed is irrelevant here.}
Algorithm 2

Initialize $\mathbb{C}$ to be empty;
for each $i = m, \ldots, 2$ do
    for each $s \in T_i$ do
        Choose an arbitrary simple cycle $c \in L_{i-1}$ that passes $s$;
        Add $c$ to $\mathbb{C}$.
    end for
end for

$p \in \text{Span}(\mathbb{L}_p)$. Recall that the main objective here is to obtain a small bounding box. However, $C_p$, the set of simple cycles extracted from $p$, may be exponentially large (in $|S|$); $q_0$ may therefore be too long to result in a useful bound. We need to improve the representation of $\text{Span}(\mathbb{L}_p)$ by making $q_0$ shorter.

![Figure 6.2: A Layered Structure](image)

For each simple cycle $c$ in $L_i$ ($i = 1, \ldots, m$), it can be observed that $S_c \cap T_i \neq \emptyset$. Also, for each $s \in T_i$ ($i = 2, \ldots, m$), there exists a simple cycle $c \in L_{i-1}$ that passes $s$. From these two observations, we can construct a smaller set $\mathbb{C}$ of simple cycles using **Algorithm 2**. Obviously $\mathbb{C}$ contains exactly $|T| - |T_1| \leq |S| - |p_0|$ simple cycles, where $T = \bigcup_{1 \leq i \leq m} T_i$, and $q'_0 = p_0 + \sum_{c \in \mathbb{C}} c$ constitutes a path of $A$. Additionally, $q'_0$ has two good properties. One is that the length $|q'_0|$ is bounded by $|p_0| + |\mathbb{C}| \cdot \max_{c \in \mathbb{C}} |c|$. Hence, $|q'_0| \leq |p_0| + (|S| - |p_0|)|S| \leq |S|^2$. Another is that $q'_0$ passes each of the touch states in $T$, i.e., $T \subseteq S_{q'_0}$. Since each simple cycle $c \in C_p$ passes at least one state in $T$, we can immediately conclude that $q'_0 + \sum_{c \in C_p} X_c c$ constitutes a path of $A$ for all $X_c \geq 0$. Then, we define $\text{Span}'(\mathbb{L}_p)$ as $\{q'_0 + \sum_{c \in C_p} X_c c : \text{each } X_c \geq 0\}$. Since $\mathbb{C} \subseteq C_p$, it is easy to see that $\text{Span}(\mathbb{L}_p) \subseteq \text{Span}'(\mathbb{L}_p)$ and $p \in \text{Span}'(\mathbb{L}_p)$. 
For every $p$, $(\text{Span}'(\mathbb{L}_p)) = \{ #(q'_0) + \sum_{c \in C_p} X_c #(c) : \text{each } X_c \geq 0 \}$ is a small linear set. This is because $\| #(q'_0) \|_\infty \leq |S|^2$, $\| #(c) \|_\infty \leq |S|$, and there are at most $r \leq |S|^k$ distinct vectors $(c)$ for all simple cycles $c \in C_p$.

Observe that there are only finitely many distinct sets $\text{Span}'(\mathbb{L}_p)$ for all $p \in \mathbb{P}$. Since, for each $p \in \mathbb{P}$, $\text{Span}'(\mathbb{L}_p) \subseteq \mathbb{P}$, we immediately obtain $\mathbb{P} = \bigcup_{1 \leq i \leq t} \text{Span}'(\mathbb{L}_{p_i})$ for some $t$ and $p_1, \ldots, p_t \in \mathbb{P}$. Define $Q_i = #(\text{Span}'(\mathbb{L}_{p_i}))$, for $1 \leq i \leq t$. The lemma follows since $\#(\mathbb{P}) = \bigcup_{1 \leq i \leq t} Q_i$ and, as we have shown, each $Q_i$ is a small linear set.

Now let us turn to the property formula $U$. Recall that $U$ is written as a disjunction of $m$ conjunctive linear constraints

$$\quad U = \bigvee_{1 \leq i \leq m} U_i. \quad (6.4)$$

Fix any $1 \leq i \leq m$. Suppose that $U_i$ contains $l$ atomic linear constraints. After adding (at most $l$) slack variables $y_1, \ldots, y_l$, $U_i$ can be written into the following form:

$$\left\{ \begin{array}{l}
  b_{i1}x_1 + \ldots + b_{ik}x_k + g_1y_1 = b_1 \\
  \vdots \\
  b_{l1}x_1 + \ldots + b_{lk}x_k + g_ly_l = b_l,
\end{array} \right. \quad (6.5)$$

where the $b$'s and $g$'s are integers (each $g$ is -1 or 0). Let $B$ be the coefficient matrix for variables $x_1, \ldots, x_k$ and $b$ be the column of $b_1, \ldots, b_l$ in (6.5). Define $w_1 = \|B\|_{1,\infty}$ and $w_2 = \|b\|_\infty$. We may assume $w_1 > 0$ (otherwise let $w_1 = 1$). In the sequel, we use the following notions: $W_1$ (the maximum of all the values $w_1$ among all $U_i$'s), $W_2$ (the maximum of all the values $w_2$ among all $U_i$'s), and $L$ (the maximum of all the values $l$ among all $U_i$'s).

Due to the disjunctive representations of (6.4) and (6.3), we can consider only one conjunction of $U$ in the form of (6.5) and only one linear set in the form of (6.2). That is, by substituting the expression in (6.2) for $\mathbf{x} = (x_1, \ldots, x_k)$ in (6.5): $\mathbf{x} = \mathbf{e}_0 + \sum_{1 \leq j \leq r} X_j \mathbf{e}_j$, the equation system (6.5) is transformed into the following
CHAPTER 6. THE LINEAR REACHABILITY PROBLEM

equation system with unknowns \(X_1, \ldots, X_r\) and \(y_1, \ldots, y_l\):

\[
\begin{align*}
  h_{11}X_1 + \cdots + h_{1r}X_r + g_1y_1 &= d'_1 \\
  \vdots \\
  h_{r1}X_1 + \cdots + h_{rr}X_r + g_ly_l &= d'_l.
\end{align*}
\]  

(6.6)

Hence, the linear reachability problem is reduced to finding a nonnegative integer solution to (6.6). With the bounds on \(e_0\) and each \(e_j\) given in (6.2), a simple calculation reveals that, in (6.6), all of the \(h\)'s are bounded by \(|S|W_1\) and all of the \(d'_1, \ldots, d'_l\) are bounded by \(|S|^2W_1 + W_2\).

We use \(\Gamma_1\) to denote the maximum of the absolute values of all the \(t \times t\), \(1 \leq t \leq l\), minors of the coefficient matrix for system (6.6) and \(\Gamma_2\) to denote that of the augmented matrix. With the above mentioned bounds for the coefficients and constants in (6.6), one can conclude that

\[
\Gamma_1 \leq (|S|W_1)^l! \quad \text{and} \quad \Gamma_2 \leq (|S|^2W_1 + W_2)(|S|W_1)^{l-1}l!.
\]  

(6.7)

A direct application of the Borůvka-Flahive-Treybig bound in Theorem 6.1 shows that system (6.6) has solutions in nonnegative integers iff the system has a solution \((X_1, \ldots, X_r, y_1, \ldots, y_l)\) in nonnegative integers, among which \(r\) unknowns are bounded by \(\Gamma_1\) and \(l\) unknowns are bounded by \((r + 1)\Gamma_2\) (here, without loss of generality, we assumed the worst case where the rank of coefficient matrix of (6.6) is \(l\)).

Applying the bounds \(\Gamma_1\) and \((r + 1)\Gamma_2\) to \(X_j\) in (6.2), the linear reachability problem is further reduced to the problem of finding a path \(p \in \mathcal{P}\) satisfying:

\[
||\#(p)||_{\infty} \leq (|S|^2 + (r - l)|S|\Gamma_1 + l|S|(r + 1)\Gamma_2)
\]  

(6.8)

and \(U(\#_{a_1}(p), \ldots, \#_{a_k}(p))\). Noticing that \(l \leq L\), and \(r \leq |S|^k\) according to (6.2), we apply the bounds of \(\Gamma_1\) and \(\Gamma_2\) in (6.7) to (6.8) and define a bounding box

\[
B = (|S|^{k+2}W_1 + L|S|(|S|^k + 1)(|S|^2W_1 + W_2))(|S|W_1)^{L-1}L! + |S|^2.
\]  

(6.9)

Hence,
CHAPTER 6. THE LINEAR REACHABILITY PROBLEM

Theorem 6.3 Given a finite-state labeled transition system $\mathbb{A}$, two states $s_{\text{init}}, s_{\text{final}} \in S$, and a linear constraint $U(x_1, \ldots, x_k)$, the following items are equivalent:

- there is a path $p$ of $\mathbb{A}$ from $s_{\text{init}}$ to $s_{\text{final}}$ satisfying $U$,
- the above item is true for some $p$ further satisfying $\|\#(p)\|_\infty \leq B$, where $B$ is defined in (6.9).

Notice that $B$ in (6.9) is independent of $m$ in (6.4). Now we measure the “size” of $U$ with $\max(W_1, W_2, L)$. When the number of states $|S|$ in $\mathbb{A}$ is $\gg k$ and the size of $U$, the bounding box is in the order of $B = O(|S|^{k+L+3})$. In this case, one can easily show the following.

Theorem 6.4 The linear reachability problem for finite-state labeled transition systems is solvable in time

$$O(|S|^{2(k+L+3)+2}),$$

when $|S| \gg k, W_1, W_2, L$.

6.4 The Linear Liveness Problem

An $\omega$-path $\pi$ of $\mathbb{A}$ is an infinite sequence such that each prefix is a path of $\mathbb{A}$. Let $s$ and $s'$ be any two designated states of $\mathbb{A}$. We say that $\pi$ is $U$-i.o. (infinitely often) at $s'$ if there are infinitely many prefixes $p$ from $s$ to $s'$ of $\pi$ such that $p$ satisfies $U$ (i.e., $U(\#a_1(p), \ldots, \#a_n(p))$ holds). The linear liveness problem for finite-state labeled transition systems can be formulated as follows.

- Given: A finite-state labeled transition system $\mathbb{A}$, two designated states $s$ and $s'$, and a linear constraint $U(x_1, \ldots, x_k)$.
- Question: Is there an $\omega$-path $\pi$ that starts from $s$ and is $U$-i.o. at $s'$?

In [34], this problem is shown decidable. However, the time complexity was unknown. In this section, we reduce the liveness problem to a linear reachability problem.

Recall that $U$ is in the form of (6.4), $U = \bigvee_{1 \leq i \leq m} U_i$, and $U_i^{\text{hom}}$ is the result of making $U_i$ homogeneous. One key observation is as follows. The Question-part in
the linear liveness problem is true iff, for some $1 \leq i \leq m$, (a). there is a path of $A$ from $s$ to $s'$ satisfying $U_i$, and, (b). there is a path of $A$ from $s'$ to $s'$ satisfying $U_i^{\text{hom}}$. A proof of this observation can be followed from [34] using the pigeon-hole principle. Both items are equivalent to the linear reachability problem for $A$ concerning $U_i$ and $U_i^{\text{hom}}$, respectively. By trying out all of the $m$ number of $U_i$'s and $U_i^{\text{hom}}$'s, and using Theorem 6.3 and (6.10), we conclude that:

**Theorem 6.5** The linear liveness problem for finite-state labeled transition systems is solvable in time shown in (6.10), when $|S| \gg m, k, W_1, W_2, L$.

### 6.5 Summary

In this chapter, we obtained a number of new complexity results for various linear counting problems (reachability and liveness) for finite-state labeled transition systems. At the heart of the proofs, we used some known results in estimating the upper bound for minimal solutions (in nonnegative integers) for linear Diophantine systems. In particular, when all the parameters (such as the number of labels/, the size of the linear constraint to be verified, etc.) except the number of states, $|S|$, of the underlying transition system are considered constants, the complexity bounds obtained in this chapter is polynomial in $|S|$. This is, as we mentioned in Section 6.1, in contrast to the exponential bounds that were previously known. In practice, a requirement specification (e.g., the $U$ in a linear counting problem) is usually small and simple [38]. In this sense, our results are useful, since the large $|S|$ is usually the dominant factor in efficiently solving these verification problems. However, in real-world applications, how to use the structural information (such as modularity) of a transition system to obtain a smaller bounding box remains a practical problem to solve.
Chapter 7

A Solvable Class of Quadratic Diophantine Equation Systems\(^1\)

As mentioned in Chapter 2, a software component in real world is often of infinite instead of finite state space. In this chapter, we study a class of infinite-state systems that contain parameterized or unspecified constants. We show that various verification problems over this class of infinite-state systems can be reduced to the satisfiability problem of a special class quadratic Diophantine equation systems. Then we devote the majority of this chapter to show that that the satisfiability problem is actually solvable.

This chapter is organized as follows. Section 7.1 provides an overview of this study. Section 7.2 introduces some definitions as well as present the decidability results for the satisfiability problem of two special classes of quadratic Diophantine systems (Lemma 7.2 and Theorem 7.3). Then in Section 7.3, we generalize the verification problem in (*) in terms of weighted semi-linear languages, and reduce the problem and its restricted versions to the classes of quadratic Diophantine systems studied in Section 7.2. In Section 7.4, we discuss the application aspects and extensions of the decidability results to other machine models. Section 7.5 is a brief summary.

\(^1\)The content of this chapter is based upon the joint work with Z. Dang and O. Ibarra in [95]
7.1 Introduction

The successful application of model checking to the verification of finite-state systems have greatly inspired researchers to develop automatic techniques for analyzing infinite-state systems (such as systems that contain integer variables and parameters). However, in general, it is not possible to develop such techniques, e.g., it is not possible to (automatically) verify whether an arithmetic program with two integer variables is going to halt [73]. Therefore, an important aspect of the research on infinite-state system verification is to identify what kinds of practically useful infinite-state models are decidable with respect to a particular form of properties (e.g., reachability).

For instance, consider a nondeterministic finite-state system $M$. Each transition in $M$ is assigned a label. On firing the transition $s \rightarrow^a s'$ from state $s$ to state $s'$ with label $a$, an activity $a$ is performed. There are finitely many labels $a_1, \ldots, a_l$ in $M$. $M$ can be used to model, among others, a finite-state process where an execution of the process corresponds to an execution path (e.g., $s_0 \rightarrow^{a_0} s_1 \rightarrow^{a_1} \cdots \rightarrow^{a_r} s_{r+1}$, for some $r$) in $M$. On the path, a sequence of activities $a^0 \cdots a^r$ are performed. Let $\Sigma_1, \ldots, \Sigma_k$ be any $k$ sets (not necessarily disjoint) of labels. An activity $a$ is of type $i$ if $a \in \Sigma_i$. An activity could have multiple types. Additionally, activities $a_1, \ldots, a_l$ are associated with weights $w_1, \ldots, w_l$ that are unspecified (or parameterized) constants in $\mathbb{N}$, respectively. Depending on the various application domains, the weight of an activity can be interpreted as, e.g., the time in seconds, the bytes of memory, or the budget in dollars, etc., needed to complete the activity. A type of activities is useful to model a “cluster” of activities. When executing $M$, we use nonnegative integer variables $W_i$ to denote the accumulated weight of all the activities of type $i$ performed so far, $1 \leq i \leq k$. One verification question concerns reachability:

(*) whether, for some values of the parameterized constants $w_1, \ldots, w_l$, there is an execution path from a given state to another state and on which

\[ w_1, \ldots, w_l, W_1, \ldots, W_k \] satisfy a given Presburger formula $P$ (a Boolean combination of linear constraints and congruences).

One can easily find applications for the verification problem; e.g., examples studied
later in the chapter.

In this chapter, we study the verification problem in (*) and its variants. First, we show that the problem is undecidable, in general. Then, we consider various restricted as well as modified cases in which the problem becomes decidable. For instance, if \( P \) in (*) has only one linear constraint that contains some of \( W_1, \ldots, W_k \), then the problem is decidable. Also, rather surprisingly, if in the problem in (*) we assume that the weight of each activity \( a_i \) can be nondeterministically chosen as any value between a concrete constant (such as 5) and a parameterized constant \( w_i \), then it becomes decidable. We also consider cases when the transition system is augmented with other unbounded data structures, such as a pushdown stack, dense clocks, and other restricted counters.

In the heart of our decidability proofs, we first show that some special classes of systems of quadratic Diophantine equations/inequalities are decidable (though in general, these systems are undecidable [71]). This nonlinear Diophantine approach toward verification problems is significantly different from many existing techniques for analyzing infinite-state systems (e.g., automata-theoretic techniques in [64, 15, 32], computing closures for Presburger transition systems [30, 19], etc.). Then, we study a more general version of the verification problem by considering weighted semi-linear languages in which a symbol is associated with a weight. Using the decidability results on the restricted classes of quadratic Diophantine systems, we show that various verification problems concerning weighted semi-linear languages are decidable. Finally, as applications, we “re-interpret” the decidability results for weighted semi-linear languages into the results for some classes of machine models, whose behaviors (e.g., languages accepted, reachability sets, etc) are known to be semi-linear, augmented with weighted activities.

Adding weighted activities to a transition system can be found, for instance, in [66]. In that paper, a “price” is associated with a control state in a timed automaton [4]. The price may be very complex; e.g., linear in other clock values etc. In general, the reachability problem for priced timed automata is undecidable [66]. Here, we are mainly interested in the decidable cases of the problem: what kind of “prices” (i.e., weights) can be placed such that some verification queries are still decidable,
for transition systems like pushdown automata, restricted counter machines, etc., in addition to timed automata.

7.2 Preliminaries

Let $P(x_1, x_2)$ be a Presburger formula over two nonnegative integer variables. $P$ is unitary if each linear constraint in $P$ is of the form $a_1 x_1 + a_2 x_2 > b$ with $a_2 = 0, -1$, or 1. $P$ is 1-congruence-free (resp. 2-congruence-free) if $P$ does not contain any congruences of the form $x_1 \equiv_b c$ (resp. $x_2 \equiv_b c$) where $b \neq 0, 0 \leq c < b$. $P$ is a point if $P$ is $x_1 = a_1 \wedge x_2 = a_2$ for some $a_1, a_2 \in \mathbb{N}$. $P$ is a line if $P$ is $x_2 = ax_1 + b$ with $a, b \in \mathbb{N}$, or $P$ is $x_1 = b$ (called a vertical line) with $b \in \mathbb{N}$. $P$ is a sector if $P$ is $x_2 \geq ax_1 + b$ with $a, b \in \mathbb{N}$, or $P$ is $ax_1 + b \leq x_2 \leq a'x_1 + b'$ with $a < a' \in \mathbb{N}$ and $b \leq b' \in \mathbb{N}$. Observe that if $P$ is congruence-free (i.e., both 1-congruence-free and 2-congruence-free) and unitary, then there is a large number $d$ such that $P(d + x_1, x_2)$ can be written into a (finite) disjunction of points, lines, and sectors.

A linear polynomial is a polynomial of the form $a_0 + a_1 x_1 + \ldots + a_n x_n$ where each coefficient $a_i$, $0 \leq i \leq n$, is an integer. The polynomial is constant if each $a_i = 0$, $1 \leq i \leq n$. The polynomial is nonnegative if each $a_i$, $0 \leq i \leq n$, is in $\mathbb{N}$. The polynomial is positive if it is nonnegative and $a_0 > 0$. A variable appears in a linear polynomial iff its coefficient in that polynomial is nonzero. The following result is needed in the chapter.

**Lemma 7.1** It is decidable whether an equation of the following form has a solution in nonnegative integer variables $s_1, \ldots, s_m, t_1, \ldots, t_n$:

$$L_0 + L_1 t_1 + \ldots + L_n t_n = 0 \tag{7.1}$$

where $L_0, L_1, \ldots, L_n$ are linear polynomials over $s_1, \ldots, s_m$. The decidability remains even when the solution is restricted to satisfy a given Presburger formula $P$ over $s_1, \ldots, s_m$.

**Proof.** The first part of the lemma has already been proved in [33], while the second part is shown below using a “semi-linear transform”. As we mentioned earlier, the set
of all \((s_1, \cdots, s_m) \in \mathbb{N}^m\) satisfying \(P\) is a semi-linear set (i.e., a finite union of linear sets). For each linear set of \(P\), one can find nonnegative integer variables \(u_1, \cdots, u_k\) for some \(k\) and a nonnegative linear polynomial \(p_i(u_1, \cdots, u_k)\) for each \(1 \leq i \leq m\) such that \((s_1, \cdots, s_m)\) is in the linear set iff each \(s_i = p_i(u_1, \cdots, u_k)\), for some \(u_1, \cdots, u_k\).

The second part follows from the first part by substituting \(p_i(u_1, \cdots, u_k)\) for \(s_i\) in \(L_0, L_1, \cdots, L_n\).

Let \(I, J\) and \(K\) be three pairwise disjoint subsets of \(\{1, \cdots, n\}\). An \(n\)-inequality is an inequality over \(n\) nonnegative integer variables \(t_1, \cdots, t_n\) and \(m\) (for some \(m\)) nonnegative integer variables \(s_1, \cdots, s_m\) of the following form:

\[
D_1 + a\left(\sum_{i \in I} L_{1i} t_i + \sum_{j \in J} L_{1j} t_j\right) \leq D_2 + \sum_{i \in I} L_{2i} t_i + \sum_{k \in K} L_{2k} t_k
\]

\[
\leq D'_1 + a'\left(\sum_{i \in I} L_{1i} t_i + \sum_{j \in J} L_{1j} t_j\right),
\]

(7.2)

where \(a < a' \in \mathbb{N}\), the \(D\)'s (resp. the \(L\)'s) are nonnegative (resp. positive) linear polynomials over \(s_1, \cdots, s_m\), and \(D_1 \leq D'_1\) is always true (i.e., true for all \(s_1, \cdots, s_m \in \mathbb{N}\)).

**Lemma 7.2** For any \(n\), it is decidable whether an \(n\)-inequality in (7.2) has a solution in nonnegative integer variables \(s_1, \cdots, s_m, t_1, \cdots, t_n\). The decidability remains even when the solution is restricted to satisfy a given Presburger formula \(P\) over \(s_1, \cdots, s_m\).

**Proof.** Here we only show the first part of the lemma. The proof for the second part will use a "semi-linear transform" similar to that in the proof of Lemma 7.1.

The proof is an induction on \(n\). Clearly, the lemma holds when \(n = 0\), since (7.2) will be Presburger. Then assuming that the lemma holds when the number of \(t\)-variables is \(0, \cdots, n - 1\), we are going to show the lemma when the number is \(n\). We only consider the case when \(a > 0, I \neq \emptyset, J = \emptyset\) and \(K = \emptyset\). The other cases for \(I = \emptyset, J \neq \emptyset, K \neq \emptyset\) or for \(a = 0\) can be handled either easily or analogously. For notational convenience, we simply let \(a = 1\) and \(a' = 2\); readers can generalize the...
proof for any $0 < a < a'$. Under all these assumptions, (7.2) can be written as

$$D_1 + \sum_{i \in I} L_{1i} t_i \leq D_2 + \sum_{i \in I} L_{2i} t_i \leq D'_1 + 2 \sum_{i \in I} L_{1i} t_i.$$  

(7.3)

Case 1. $L_{1i_0} < L_{2i_0} < 2L_{1i_0}$ is satisfiable for some $i_0 \in I$. In this case, (7.3) has solutions by making $t_{i_0}$ large but all the other $t_i$’s zero; recall that the $L$’s in (7.3) are positive linear polynomials.

Case 2. For each $i \in I$, $L_{2i} \geq 2L_{1i} \lor L_{2i} \leq L_{1i}$ is always true (i.e., true for all $s_1, \ldots, s_m \in \mathbb{N}$). This case can be split into the following two subcases:

Case 2.1. There are $i_1 \neq i_2 \in I$ such that $L_{2i_1} \geq 2L_{1i_1} \land L_{2i_2} \leq L_{1i_2}$ is satisfiable, witnessed by some $(s^0_{i_1}, \ldots, s^0_{i_2})$. Then (7.3) has a solution in which $(s_1, \ldots, s_m)$ takes the value of $(s^0_{i_1}, \ldots, s^0_{i_2})$, and all the $t_1, \ldots, t_n$ are zero except that $t_{i_1}$ and $t_{i_2}$ are chosen as below. Pick a positive rational number $\delta$ that satisfies

$$\frac{L_{1i_2}(s^0_{i_1}, \ldots, s^0_{i_2}) - L_{2i_2}(s^0_{i_1}, \ldots, s^0_{i_2})}{L_{2i_1}(s^0_{i_1}, \ldots, s^0_{i_2}) - L_{1i_1}(s^0_{i_1}, \ldots, s^0_{i_2})} < \delta < \frac{2L_{1i_2}(s^0_{i_1}, \ldots, s^0_{i_2}) - L_{2i_2}(s^0_{i_1}, \ldots, s^0_{i_2})}{2L_{2i_1}(s^0_{i_1}, \ldots, s^0_{i_2}) - 2L_{1i_1}(s^0_{i_1}, \ldots, s^0_{i_2})}.$$

$\delta$ exists according to the definition of $(s^0_{i_1}, \ldots, s^0_{i_2})$ and the fact that the $L$’s are positive linear polynomials. The desired values for $t_{i_1}$ and $t_{i_2}$ can be obtained by making both of them large enough while satisfying $\frac{t_{i_1}}{t_{i_2}} = \delta$.

Case 2.2. For each $(s_1, \ldots, s_m)$, either $L_{2i} \geq 2L_{1i}$ for all $i \in I$, or, $L_{1i} \geq L_{2i}$ for all $i \in I$. We only consider the situations (values for $(s_1, \ldots, s_m)$) when $L_{2i} \geq 2L_{1i}$ holds for all $i \in I$; the other situations can be handled similarly. Under those situations, (7.3) has no solution whenever $D_2 > D'_1$. On the other hand, if $D_1 \leq D_2 \leq D'_1$, then (7.3) has a trivial zero solution. Another simple case is when $L_{2i} = 2L_{1i}$ for each $i \in I$; it is not hard to solve (7.3) under this case using Lemma 7.1, after a semi-linear transform upon constraint $D_2 \leq D'_1 \land \bigwedge_{i \in I} L_{2i} = 2L_{1i}$. So, we shall only look at those $(s_1, \ldots, s_m)$ satisfying $\bigwedge_{i \in I^+} L_{2i} > 2L_{1i}$ and $\bigwedge_{i \in I^-} L_{2i} = 2L_{1i}$ for any fixed $I^+ \neq \emptyset$ and $I^- = I - I^+$. Therefore, after removing all the simple cases for $(s_1, \ldots, s_m)$, it suffices for us to solve (7.3) under the following Presburger constraint
CHAPTER 7. QUADRATIC DIOPHANTINE EQUATION SYSTEMS

For \((s_1, \ldots, s_m)\):

\[
D_2 < D_1 \land (\bigwedge_{i \in I^+} L_{2i} > 2L_{1i}) \land (\bigwedge_{i \in I^-} L_{2i} = 2L_{1i}). \tag{7.4}
\]

Of course, if \(P\) is not satisfiable, then we are done, since (7.3) will have no solution under \(P\). The case when \(P\) is satisfiable is nontrivial. To handle this nontrivial case, we rewrite (7.3) into the following:

\[
\begin{align*}
\sum_{i \in I^+} (L_{2i} - 2L_{1i})t_i + \sum_{i \in I} L_{1i}t_i + D_1^t - D_1 & \geq D_1^t - D_2 \\
\sum_{i \in I^+} (L_{2i} - 2L_{1i})t_i & \leq D_1^t - D_2.
\end{align*}
\tag{7.5}
\]

At this point, a semi-linear transform can be applied upon Presburger constraint \(P\) in (7.4) in a similar way to that in the proof of Lemma 7.1.

After the transform, solving (7.5) under the constraint \(P\) is reduced to solving the following:

\[
\begin{align*}
\sum_{i \in I^+} F_{11}t_i + \sum_{i \in I} F_{2i}t_i + G_2 & \geq G_1 \\
\sum_{i \in I^+} F_{11}t_i & \leq G_1
\end{align*}
\tag{7.6}
\]

where \(G_1, G_2\) are nonnegative linear polynomials, and all the \(F\)'s are positive linear polynomials. All these linear polynomials in (7.6) are over a new set of nonnegative integer variables \(u_1, \ldots, u_k\) (for some \(k\)) as a result of the semi-linear transform upon \(P\). To find a solution \((u_1, \ldots, u_k; t_1, \ldots, t_n)\) for (7.6), we firstly try to find a free variable in (7.6). We say that \(u_j\) (\(1 \leq j \leq k\)) is a free variable in (7.6) if \(u_j\) appears in \(G_1\) but not in any \(F_{11}, i \in I^+\). Assume

\[
G_1 = g_0 + \sum_{1 \leq j \leq k} g_ju_j,
\]

for some \(g_0, g_1, \ldots, g_k \in \mathbb{N}\). If there are no free variables in (7.6), then by looking at the second inequality of (7.6), one can observe that any solution \((u_1, \ldots, u_k; t_1, \ldots, t_n)\) of (7.6) must satisfy the following condition: \(t_i, \text{ for some } i \in I^+\), is bounded by \(1 + \max(g_0, g_1, \ldots, g_k)\). Hence, the induction hypothesis can be applied to solve the
original inequality (7.3) but with a smaller $n$. However, if (7.6) has at least one free variable (say $u_1$), then we argue that (7.6) has a solution as follows. We firstly rewrite (7.6) by separating the free variable from $G_1$:

$$
\begin{align*}
\sum_{i \in I^+} F_{1i}t_i - g_0 - \sum_{1 < j \leq k} g_j u_j + \sum_{i \in I} F_{2i}t_i + G_2 &\geq g_1 u_1 \\
\sum_{i \in I^+} F_{1i}t_i - g_0 - \sum_{1 < j \leq k} g_j u_j &\leq g_1 u_1
\end{align*}
$$

(7.7)

Notice that $g_1 > 0$. Then pick any element $i$ (say $i = 1$) in $I^+$. Let all the $u_2, \ldots, u_k$ and $t_2, \ldots, t_n$ be zero in (7.7). Clearly, a solution $(u_1, 0, \ldots, 0; t_1, 0, \ldots, 0)$ to (7.7) can always be obtained by making $t_1$ sufficiently large and picking $u_1$ properly, noticing that all the $F$’s in (7.7) are positive linear polynomials.

This completes the proof of Lemma 7.2.

An $E(n)$ system $E(s_1, \ldots, s_m; t_1, \ldots, t_n)$, for some $m$, is a predicate over nonnegative integer variables $s_1, \ldots, s_m, t_1, \ldots, t_n$ in the following form:

$$
P(D_1 + \sum_{1 \leq i \leq n} L_{1i} t_i, D_2 + \sum_{1 \leq i \leq n} L_{2i} t_i)
$$

(7.8)

where $P$ is a Presburger formula over two nonnegative integer variables and the $D$’s and the $L$’s are nonnegative linear polynomials over $s_1, \ldots, s_m$. $E$ is a 1-congruence-free (resp. 2-congruence-free, unitary, point, line, sector) $E(n)$ system if $P$ is 1-congruence-free (resp. 2-congruence-free, unitary, a point, a line, a sector). We use $\exists t_1, \ldots, t_n E$ to denote the set of all $(s_1, \ldots, s_m) \in \mathbb{N}^m$ satisfying

$$
\exists t_1, \ldots, t_n \in \mathbb{N}. E(s_1, \ldots, s_m; t_1, \ldots, t_n).
$$

The rest of this section is to show that,

Claim. For any $n$, the emptiness problem of $E$ (i.e., whether $\exists t_1, \ldots, t_n E$ is an empty set) is decidable for any $E(n)$ system $E$.

The Claim will be used later in the chapter. We first point out that the Claim can be simplified.
• It suffices for us to show the Claim for those $E$’s that are 2-congruence-free. Suppose that $E$ in (7.8) is not 2-congruence-free. Let $B$ be the multiplication of all the $b$’s appearing in congruences $X_2 \equiv_b c$ in Presburger formula $P(X_1, X_2)$. Fix any tuple $\tau = (c_1, \ldots, c_m; c^1, \ldots, c^n)$ in $\{0, \ldots, B - 1\}^{m+n}$. Define $E_{\tau}(s'_1, \ldots, s'_m; t'_1, \ldots, t'_n) \equiv_{def} E(Bs'_1 + c_1, \ldots, Bs'_m + c_m; Bt'_1 + c^1, \ldots, Bt'_n + c^n)$. $E_{\tau}$ is 2-congruence-free since, under the transform, a congruence like $X_2 \equiv_b c$ has a true/false value (i.e., the congruence can be eliminated). The argument of this item follows from the fact that $\exists^{t'_1, \ldots, t'_n} E$ equals to the set of all the $(s_1, \ldots, s_m)$ satisfying

$$\forall \exists s'_1, \ldots, s'_m \in \mathbb{N}. s_1 = Bs'_1 + c_1 \land \cdots \land s_m = Bs'_m + c_m \land \exists^{t'_1, \ldots, t'_n} E_{\tau}.$$ (7.9)

(Clearly, from above, $\exists^{t'_1, \ldots, t'_n} E$ is empty iff each $\exists^{t'_1, \ldots, t'_n} E_{\tau}$ is and we have finitely many $\tau$’s.)

• It suffices for us to show the Claim for those $E$’s that are 2-congruence-free and unitary. Suppose that $E$ is 2-congruence-free but not unitary. Let $B$ be the absolute value of the multiplication of all the non-zero coefficients $a_2$ of $x_2$ in linear constraints $a_1 X_1 + a_2 X_2 > b$ appearing in Presburger formula $P(X_1, X_2)$. Again, fix any tuple $\tau = (c_1, \ldots, c_m; c^1, \ldots, c^n)$ in $\{0, \ldots, B - 1\}^{m+n}$. We use $X_j(s_1, \ldots, s_m; t_1, \ldots, t_n)$, $j = 1, 2$, to denote $D_j + \sum_{1 \leq i \leq n} L_j t_i$ in (7.8). Define $X_j(\tau)(s'_1, \ldots, s'_m; t'_1, \ldots, t'_n) \equiv_{def} X_j(Bs'_1 + c_1, \ldots, Bs'_m + c_m; Bt'_1 + c^1, \ldots, Bt'_n + c^n)$. Let $d_\tau = X_1(\tau)(0, \ldots, 0; 0, \ldots, 0)$. It is noticed that $d_\tau$ is a number in $\mathbb{N}$ and $X_1(\tau)(s'_1, \ldots, s'_m; t'_1, \ldots, t'_n) - d_\tau$ is divisible by $B$ for all $s'_1, \ldots, s'_m, t'_1, \ldots, t'_n$. Define a Presburger formula $P_\tau(X_1, X_2) \equiv_{def} P(BX_1 + d_\tau, X_2)$. It can be shown that $P_\tau$ is unitary. Now, we construct an $E_\tau(\tau)(s'_1, \ldots, s'_m; t'_1, \ldots, t'_n)$ that is of the form $P_\tau(Y_1, Y_2)$ where $Y_1 \equiv_{def} (X_1(\tau) - d_\tau)/B$ and $Y_2 \equiv_{def} X_2(\tau)$. $E_\tau$ is a 2-congruence-free and unitary $E(m)$ system. The argument of this item follows from the fact (which is left to the reader to check) that $\exists^{t'_1, \ldots, t'_n} E$ equals to the set of all the $(s_1, \ldots, s_m)$ satisfying (7.9).

• It suffices for us to show the Claim for those $E$’s that are congruence-free and
unitary. Suppose that \( E \) is 2-congruence-free and unitary but not 1-congruence-free. The argument of this item can be established similarly as the first item above.

- So, now we assume that \( P(X_1, X_2) \) in defining \( E \) is congruence-free and unitary. It suffices for us to show the Claim by further assuming that \( P \) is a finite union of points, lines, and sectors. To see this, recall that \( P(d + X_1, X_2) \), as a unitary and congruence-free Presburger formula over two nonnegative integer variables, is a finite union of points, lines, and sectors for some large \( d \in \mathbb{N} \). So we need only to show that \( E(s_1, \ldots, s_m; t_1, \ldots, t_n) \) in (7.8) can be rewritten into \( \hat{E}(s_1', \ldots, s_m'; t_1', \ldots, t_n') \) in the following form:

\[
P(d + \hat{D}_1 + \sum_{1 \leq i \leq n} \hat{L}_{1i}t_i', \hat{D}_2 + \sum_{1 \leq i \leq n} \hat{L}_{2i}t_i')
\]  

(7.10)

where the \( D \)'s and the \( L \)'s are nonnegative linear polynomials over \( s_1', \ldots, s_m' \). The Claim for \( E \) is now reduced to the Claim but for \( \hat{E} \). Notice that both \( E \) and \( \hat{E} \) has the same number \( n \) of \( t \)-variables, but the number of \( s \)-variables may be different.

In order to obtain (7.10) from (7.8), we try to write \( D_1 + \sum_{1 \leq i \leq n} L_{1i}t_i \) in (7.8) into the form \( d + \hat{D}_1 + \sum_{1 \leq i \leq n} \hat{L}_{1i}t_i' \) in (7.10). We have the following cases to consider. We first consider a constraint

\[
D_1 \geq d
\]  

(7.11)

over \( s_1, \ldots, s_m \). Under (7.11), after a semi-linear transform on the \( s \)-variables, one can easily write (7.8) into (7.10). We now consider another constraint

\[
\bigwedge_{1 \leq i \leq n} t_i \geq d \land \bigvee_{1 \leq i \leq n} L_{1i} > 0.
\]  

(7.12)

Under (7.12), after a semi-linear transform on the \( s \)-variables (upon the second conjunction of (7.12)) and after a transform \( t_i' = t_i + d \), for each \( 1 \leq i \leq n \), one
can write (7.8) into (7.10). What comes out of constraints (7.11) and (7.12) is either
\[ t_i < d \quad \forall 1 \leq i \leq n \] (7.13)
or
\[ L_{1i} = 0 \land D_1 < d. \] (7.14)

In the case of (7.13), an induction can be used to show the Claim on a smaller number of \( n \) (since one of the \( t_i \)'s has a bounded value less than \( d \); in particular, when \( n = 0 \), the Claim trivially holds). In the case of (7.14), the Claim is already decidable (using a semi-linear transform upon (7.14) and applying Lemma 7.1).

So, now we assume that \( P(X_1, X_2) \) in defining \( E \) is a finite union of points, lines, and sectors. Therefore, it suffices for us to show the Claim for those \( E \)'s that are point (resp. line, sector) \( E(n) \) systems.

When \( P \) in (7.8) is a point (i.e., \( X_1 = a \land X_2 = b \) for some \( a, b \in \mathbb{N} \)), (7.8) is equivalent to the following system:
\[
\begin{align*}
D_1 + \sum_{1 \leq i \leq n} L_{1i} t_i &= a \\
D_2 + \sum_{1 \leq i \leq n} L_{2i} t_i &= b
\end{align*}
\] (7.15)
The Claim obviously holds for this case since all solutions to \((s_1, \ldots, s_m; t_1, \ldots, t_n)\) is definable by a Presburger formula (noticing that each term \( L_{1i} t_i, L_{2i} t_i, D_1, D_2 \) must be bounded by numbers \( a \) and \( b \)).

Now, we consider the case when \( P \) is a line (i.e., \( X_2 = aX_1 + b \) with \( a, b \in \mathbb{N} \), or \( X_1 = b \) with \( b \in \mathbb{N} \)). The case for \( X_1 = b \) is trivial. For \( X_2 = aX_1 + b \), (7.8) is equivalent to \( D_2 + \sum_{1 \leq i \leq n} L_{2i} t_i = a(D_1 + \sum_{1 \leq i \leq n} L_{1i} t_i) + b \), which can be reduced to the equation in Lemma 7.1. Hence, the Claim holds for lines.

The last case is when \( P \) is a sector (i.e., \( X_2 \geq aX_1 + b \) or \( aX_1 + b \leq X_2 \leq a'dX_1 + b' \), for some \( a < a' \in \mathbb{N} \) and \( b \leq b' \in \mathbb{N} \)). For \( X_2 \geq aX_1 + b \), (7.8) is equivalent to \( D_2 + \sum_{1 \leq i \leq n} L_{2i} t_i = u + a(D_1 + \sum_{1 \leq i \leq n} L_{1i} t_i) + b \), which can be reduced to the equation
in Lemma 7.1. For \( a \mathbf{X}_1 + b \leq \mathbf{X}_2 \leq a' \mathbf{X}_1 + b' \), (7.8), i.e.,

\[
a(D_1 + \sum_{1 \leq i \leq n} L_{1i}t_i) + b \leq D_2 + \sum_{1 \leq i \leq n} L_{2i}t_i \leq a'(D_1 + \sum_{1 \leq i \leq n} L_{1i}t_i) + b'
\]

is equivalent to a finite disjunctions of \( n \)-equalities in the form of (7.2), as follows.

Let \( I, J, K, H \) is any fixed partition of \( \{1, \cdots, n\} \). \( I \) makes all the \( L_{1i} \) and \( L_{2i} \) positive, each \( i \in I \). \( J \) makes all the \( L_{1i} \) positive and all the \( L_{2i} \) zero, each \( i \in J \). \( K \) makes all the \( L_{2i} \) positive and all the \( L_{1i} \) zero, each \( i \in K \). And finally, \( H \) makes all the \( L_{2i} \) and all the \( L_{1i} \) zero, each \( i \in H \). That is, the partition derives a Presburger constraint over \( s_1, \cdots, s_m \):

\[
\bigwedge_{i \in I} (L_{1i} > 0 \land L_{2i} > 0) \land \bigwedge_{i \in J} (L_{1i} > 0 \land L_{2i} = 0) \land \bigwedge_{i \in K} (L_{1i} = 0 \land L_{2i} > 0) \land \bigwedge_{i \in H} (L_{1i} = 0 \land L_{2i} = 0).
\]

By applying a semi-linear transform upon the Presburger constraint, (7.16) can be transformed into a finite disjunction of \( n \)-equalities in the form of (7.2). More \( n \)-equalities are obtained when one enumerates all the possible partitions for \( \{1, \cdots, n\} \). From Lemma 7.2, the Claim holds for sectors.

To sum up,

**Theorem 7.3** It is decidable whether a system in the following form has a solution in nonnegative integer variables \( s_1, \cdots, s_m, \ t_1, \cdots, t_n \): \( P(D_1 + \sum_{1 \leq i \leq n} L_{1i}t_i, D_2 + \sum_{1 \leq i \leq n} L_{2i}t_i) \), where \( P \) is a Presburger formula over two nonnegative integer variables and the \( D \)'s and the \( L \)'s are nonnegative linear polynomials over \( s_1, \cdots, s_m \).

### 7.3 Semi-linear Languages with Weights

Recall the definition of semi-linear languages and Parikh maps in Section 2.4. Now, we add “weights” to a language \( L \). A *weight measure* is a mapping that maps a symbol in \( \Sigma \) to a weight in \( \mathbb{N} \). We shall use \( w_1, \cdots, w_l \) to denote the weights for
CHAPTER 7. QUADRATIC DIOPHANTINE EQUATION SYSTEMS

$a_1, \ldots, a_l$, respectively, under the measure. Let $\Sigma_1, \ldots, \Sigma_k$ be any $k$ fixed subsets of $\Sigma$. For each $1 \leq i \leq k$, we use $W_i(\alpha)$ to denote the total weight of all the occurrences for symbols $a \in \Sigma_i$ in word $\alpha$; i.e.,

$$W_i(\alpha) = \sum_{a_j \in \Sigma_i} w_j \cdot \#_{a_j}(\alpha).$$

(7.18)

$W_i(\alpha)$ is called the accumulated weight of $\alpha$ wrt $\Sigma_i$. We are interested in the following $k$-accumulated weight problem:

- **Given:** An effectively semi-linear language $L$, $k$ subsets $\Sigma_1, \ldots, \Sigma_k$ of $\Sigma$, and a Presburger formula $P$ over $l + k$ variables.

- **Question:** Is there a word $\alpha$ in $L$ such that, for some $w_1, \ldots, w_l \in \mathbb{N}$,

$$P(w_1, \ldots, w_l, W_1(\alpha), \ldots, W_k(\alpha))$$

holds?

In a later section, we shall look at the application side of the problem. The rest of this section investigates the decidability issues of the problem by transforming the problem and its restricted versions to a class of Diophantine equations.

A $k$-system is a quadratic Diophantine equation system that consists of $k$ equations over nonnegative integer variables $s_1, \ldots, s_m, t_1, \ldots, t_n$ (for some $m, n$) in the following form:

$$\begin{cases}
\sum_{1 \leq j \leq l} B_{1j}(t_1, \ldots, t_n)A_{1j}(s_1, \ldots, s_m) = C_1(s_1, \ldots, s_m) \\
\vdots \\
\sum_{1 \leq j \leq l} B_{kj}(t_1, \ldots, t_n)A_{kj}(s_1, \ldots, s_m) = C_k(s_1, \ldots, s_m)
\end{cases}$$

(7.20)

where the $A$’s, $B$’s and $C$’s are nonnegative linear polynomials, and $l, n, m$ are positive integers.

**Theorem 7.4** For each $k$, the $k$-accumulated weight problem is decidable iff it is decidable whether a $k$-system has a solution.
CHAPTER 7. QUADRATIC DIOPHANTINE EQUATION SYSTEMS

Proof. $\iff$. Consider an instance of the $k$-accumulated weight problem given in (7.19). Since $L$ is semi-linear, the Parikh map $(L)$

$$\{(\#_{a_1}(\alpha), \ldots, \#_{a_l}(\alpha)) : \alpha \in L\}$$

(7.21)

is a semi-linear set. For simplicity, we first assume that $(L)$ is a linear set. Hence, $\#_{a_1}(\alpha), \ldots, \#_{a_l}(\alpha)$ in (7.21) can be written into nonnegative linear polynomials $B_1$, $\ldots, B_1$, respectively, over $(t_1, \ldots, t_n)$, for some $n$. Presburger formula $P$ in (7.19) also defines a semi-linear set in $\mathbb{N}^{t+l}$. Again, for simplicity, we assume that the set is linear. Hence, $w_1, \ldots, w_l, W_1(\alpha), \ldots, W_k(\alpha)$ in (7.19) can be written into nonnegative linear polynomials $A_1, \ldots, A_l, C_1, \ldots, C_k$, respectively, over nonnegative integer variables $s_1, \ldots, s_m$, for some $m$. From (7.18), we may obtain a $k$-system in the form of (7.20) by letting $B_{ij} = B_j, A_{ij} = A_j$ if $a_j \in \Gamma_i, A_{ij} = 0$ if $a_j \notin \Gamma_i$, for each $1 \leq i \leq k, 1 \leq j \leq l$. It is not hard to prove that the instance of the accumulated weight problem is equivalent to the existence of solutions to the system. When $(L)$ and $P$ define semi-linear sets instead of linear sets, the instance is equivalent to a disjunction of finitely many $k$-systems.

$\Rightarrow$. Direct. $\blacksquare$

It is known [55] that there is a fixed $k$ such that there is no algorithm to solve Diophantine systems in the following form:

$$t_1 F_1 = G_1,$$

$$t_1 H_1 = I_1,$$

$$\vdots,$$

$$t_k F_k = G_k,$$

$$t_k H_k = I_k,$$

where the $F$’s, $G$’s, $H$’s, $I$’s are nonnegative linear polynomials over nonnegative integer variables $s_1, \ldots, s_m$, for some $m$. Observe that the above systems are $2k$-systems. Therefore, from Theorem 7.4,

**Theorem 7.5** There is a fixed $k$ such that the $k$-accumulated weight problem is undecidable.
CHAPTER 7. QUADRATIC DIOPHANTINE EQUATION SYSTEMS

Currently, it is an open problem to find the maximal $k$ such that the $k$-accumulated weight problem is decidable. Clearly, when $k = 1$, the problem is decidable. This is because 1-systems are decidable (Lemma 7.1). Below, using Theorem 7.3, we show that the problem is decidable when $k = 2$. Interestingly, it is still open whether the decidability remains for $k = 3$.

Consider a 2-system in (7.20) with $k = 2$. The procedure to solve the system uses an induction on the number $n$ of the $t$-variables. Clearly, if $n = 0$, the system is immediately decidable since (7.20) now is Presburger. For a general $n$, (7.20) with $k = 2$ can be written into

\[
\begin{align*}
\sum_{1 \leq i \leq n} \left( \sum_{1 \leq j \leq l} A_{1j} b_{1j}^0 \right) t_i &= C_1 - \sum_{1 \leq j \leq l} A_{1j} b_{1j}^0 \\
\sum_{1 \leq i \leq n} \left( \sum_{1 \leq j \leq l} A_{2j} b_{2j}^0 \right) t_i &= C_2 - \sum_{1 \leq j \leq l} A_{2j} b_{2j}^0
\end{align*}
\] (7.22)

where the $A$‘s and $C$‘s are nonnegative linear polynomials, and the $b$‘s are nonnegative integers for the coefficients of the $B$‘s in (7.20). Notice that in (7.22), as the left hand sides of both the equations are always nonnegative, so are the right hand sides. Similar to the proof of Lemma 7.1, a semi-linear transform can be applied upon a Presburger constraint $C_1 - \sum_{1 \leq j \leq l} A_{1j} b_{1j}^0 \geq 0 \land C_2 - \sum_{1 \leq j \leq l} A_{2j} b_{2j}^0 \geq 0$ on $(s_1, \cdots, s_m)$. The result is that (7.22) will be transformed into a finite disjunction of equation systems in the following form ((7.22) has a solution iff one of the systems in the disjunction has a solution):

\[
\begin{align*}
\sum_{1 \leq i \leq n} L_{1i} t_i &= D_1 \\
\sum_{1 \leq i \leq n} L_{2i} t_i &= D_2
\end{align*}
\] (7.23)

where the $L$‘s and the $D$‘s are nonnegative linear polynomials over a set of new nonnegative integer variables that are resulted from the semi-linear transform. For notational convenience, we still assume that the $L$‘s and the $D$‘s in (7.23) are nonnegative linear polynomials over $s_1, \cdots, s_m$. Also, w.l.o.g., we assume (7.23) has the following property\footnote{If (7.23) does not have the property, then an equivalent system can be constructed in the following way. We use $Q_1$ (resp. $Q_2$) to denote the first (resp. second) equation of (7.23). The new}: for any $s_j, 1 \leq j \leq m$, it appears in $L_{1i}$ if it appears in $L_{2i}$,
$1 \leq i \leq n$, and it appears in $D_1$ iff it appears in $D_2$. We say $s_j$ ($1 \leq j \leq m$) is a free variable in (7.23) if it appears in $D_1$ (hence in $D_2$) but not in any $L_{1i}$ (hence, neither $L_{2i}$), $1 \leq i \leq n$.

Assume that in (7.23), $D_1 = d_1^0 + \sum_{1 \leq i \leq m} d_i^1 s_i$ and $D_2 = d_2^0 + \sum_{1 \leq i \leq m} d_i^2 s_i$. If there are no free variables in (7.23), then by looking at one of the equations in (7.23), one can observe that (7.23) has a solution iff it has a solution $(s_1, \ldots, s_m; t_1, \ldots, t_n)$ satisfying the following condition: $t_i$, for some $1 \leq i \leq n$, is bounded by $2 \max(d_1^0, d_1^1, \ldots, d_m^0, d_m^1, \ldots, d_m^m)$. Hence, the induction hypothesis can be applied to solve the original system (7.22) but with a smaller $n$. Otherwise, let $I_r$ be the set of indices for all the free variables in (7.23), then (7.23) can be written into

\[
\begin{align*}
\sum_{1 \leq i \leq n} L_{1i} t_i - \sum_{j \in I_r} d_i^1 s_j &= d_1^0 + \sum_{j \in I_r} d_i^1 s_j \\
\sum_{1 \leq i \leq n} L_{2i} t_i - \sum_{j \in I_r} d_i^2 s_j &= d_2^0 + \sum_{j \in I_r} d_i^2 s_j
\end{align*}
\] (7.24)

where $I_l = \{1, \ldots, m\} - I_r$.

Let $d = \max(d_1^1, d_2^1 : j \in I_l)$. If (7.24) has a solution with some $t_i$ ($1 \leq i \leq n$) bounded by $d$, then an induction step can be made on (7.22) with a smaller number of $t$-variables (by plugging-in a nonnegative number not larger than $d$ for the $t_i$). Otherwise, a transform $t_i := t_i^1 + d$ for all $1 \leq i \leq n$ can be applied to (7.24). The result is right in the following form:

\[
\begin{align*}
D_1 + \sum_{1 \leq i \leq n} L_{1i} t_i &= d_1^0 + \sum_{j \in I_r} d_i^1 s_j \\
D_2 + \sum_{1 \leq i \leq n} L_{2i} t_i &= d_2^0 + \sum_{j \in I_r} d_i^2 s_j
\end{align*}
\] (7.25)

where the $d'$s are nonnegative integers, the $D$'s and the $L$'s are nonnegative linear polynomials over $s_1, \ldots, s_m$ and each $s_j, j \in I_r$, does not appear in any one of them.

Clearly, $\{(X_1, X_2) : X_1 = d_1^0 + \sum_{j \in I_r} d_i^1 s_j, X_2 = d_2^0 + \sum_{j \in I_r} d_i^2 s_j, \text{each } s_j \in \mathbb{N}\}$ is a linear set. Let $P(X_1, X_2)$ be a Presburger formula defining the set. Observe that system consists of two equations $Q_1' := Q_1 + 2Q_2$ (multiply both sides of $Q_2$ by 2 and add to $Q_1$) and $Q_2' := 2Q_1 + Q_2$. The new system has the property.
(7.25) has a solution iff
\[
P(D_1 + \sum_{1 \leq i \leq n} L_{1i} t_i, D_2 + \sum_{1 \leq i \leq n} L_{2i} t_i)
\]
has a solution.

From Theorem 7.3, we know that it is decidable whether a 2-system has a solution. Therefore, from Theorem 7.4,

**Theorem 7.6** The 2-accumulated weight problem is decidable.

In some restricted cases, the accumulated weight problem is decidable for a general \( k \). We are now going to elaborate these cases. Consider a \( k \)-accumulated weight problem such that (7.19) is a disjunction of formulas in the following special form:

\[
Q(w_1, \ldots, w_l) \land a_1 W_1(\alpha) + \cdots + a_k W_k(\alpha) + b_1 w_1 + \cdots + b_l w_l \sim a_0 \quad (7.26)
\]

where \( Q \) is a Presburger formula over \( l \) variables, the \( a' \)'s and \( b' \)'s are integers, and \( \sim \in \{=, \neq, >, <, \geq, \leq\} \). Under this restriction, the \( k \)-accumulated weight problem is decidable.

**Theorem 7.7** For each \( k \), the \( k \)-accumulated weight problem, in which (7.19) is a disjunction of formulas in the form of (7.26), is decidable.

**Proof.** After a semi-linear transform on Presburger formula \( Q(w_1, \ldots, w_l) \), similar to the proof of Theorem 7.4, (7.26) can be written into a disjunction of Diophantine equations in the form of (7.1). The decidability follows from Lemma 7.1. □

Currently we do not know whether Theorem 7.7 still holds if (7.26) is conjoined with one additional inequality: \( a'_1 W_1(\alpha) + \cdots + a'_k W_k(\alpha) + b'_1 w_1 + \cdots + b'_l w_l \sim a'_0 \).

As in the statement of the problem at the beginning of this section, a weight measure assigns weights \( w_1, \ldots, w_l \) to symbols \( a_1, \ldots, a_l \) respectively. Instead of a fixed one, suppose that the weight of a symbol \( a_i \) can take any value between a given number \( q_i \) and \( w_i \). That is, the weight measure defines a possible weight range that
a symbol can have, with the given number \( q_i \) being the lowest possible weight. Thus, in contrast to (7.18), \( W_i(\alpha) \), \( 1 \leq i \leq l \), will be a set:

\[
\{ \tilde{W}_i : \sum_{a_j \in \Sigma_i} q_j \cdot \#a_j(\alpha) \leq \tilde{W}_i \leq \sum_{a_j \in \Sigma_i} w_j \cdot \#a_j(\alpha) \}.
\]

(7.27)

For instance, suppose \( \Sigma_1 = \{a_1\} \), \( q_1 = 2 \), \( w_1 = 7 \), and a word \( \alpha = a_1a_1a_1 \). Clearly, 12 is a weight in \( W_1(\alpha) \) according to (7.27).

With the new definition of \( W_i(\alpha) \), the following *loose* \( k \)-accumulated weight problem can be formulated:

- **Given:** An effectively semi-linear language \( L \), numbers \( q_1, \ldots, q_l \in \mathbb{N} \), \( k \) subsets \( \Sigma_1, \ldots, \Sigma_k \) of \( \Sigma \), and a Presburger formula \( P \) over \( l + k \) variables.

- **Question:** Is there a word \( \alpha \) in \( L \) such that, for some \( w_1, \ldots, w_l \in \mathbb{N} \), and for some \( \tilde{W}_1, \ldots, \tilde{W}_k \),

\[
\tilde{W}_1 \in W_1(\alpha) \land \cdots \land \tilde{W}_k \in W_k(\alpha) \land P(w_1, \ldots, w_l, \tilde{W}_1, \ldots, \tilde{W}_k)
\]

holds?

Notice that the lower weight bounds \( q_1, \ldots, q_l \) are in the **Given**-part, hence they are constants; while the upper bounds \( w_1, \ldots, w_l \) in the **Question**-part, are essentially unspecified parameters. (Otherwise, if the lower bounds \( q_1, \ldots, q_l \) are moved into the **Question**-part; i.e., both the lower and the upper bounds are parameterized constants, then the \( k \)-accumulated weight problem is a special case of the loose \( k \)-accumulated weight problem under this definition, by letting the lower bound and the upper bound be the same parameterized constant for each activity.)

The following result shows that the loose \( k \)-accumulated weight problem is decidable for each \( k \). It is in contrast to Theorem 7.5 that the \( k \)-accumulated weight problem is undecidable for some large \( k \).

**Theorem 7.8** *For each \( k \), the loose \( k \)-accumulated weight problem is decidable.*

See Appendix A.2.1 for the proof of this theorem.
CHAPTER 7. QUADRATIC DIOPHANTINE EQUATION SYSTEMS

7.4 Applications

In this section, we will apply the results presented in the previous section to some verification problems concerning infinite systems containing parameterized constants. We start with a general definition.

A transition system $M$ can be described as a relation $T \subseteq S \times \Gamma^* \times \Sigma \times S \times \Gamma^*$, where $S$ is a finite set of states, $\Gamma$ is the configuration alphabet, and $\Sigma$ is the activity alphabet. Obviously, we always assume that $M$ can be effectively described; i.e., $T$ is recursive. A configuration $\langle s, \beta \rangle$ of $M$ is a pair of a state $s$ in $S$ and a word $\beta$ in $\Gamma^*$. In the description of $M$, an initial configuration is also designated. According to the definition of $T$, an activity in $\Sigma$ transforms one configuration to another. More precisely, we write $\langle s, \beta \rangle \overset{s'}{\rightarrow} \langle s', \beta' \rangle$ if $T(s, \beta, a, s', \beta')$. Let $\alpha \in \Sigma^*$ with $\alpha = a_1 \cdots a_m$ for some $m$. We say that $\langle s, \beta, \alpha \rangle$ is reachable if, for some configurations $\langle s_0, \beta_0 \rangle, \cdots, \langle s_m, \beta_m \rangle$, the following is satisfied

$$\langle s_0, \beta_0 \rangle \overset{a_1}{\rightarrow} \cdots \overset{a_m}{\rightarrow} \langle s_m, \beta_m \rangle,$$

(7.29)

where $\langle s_0, \beta_0 \rangle$ is the initial configuration, $s_m = s$ and $\beta_m = \beta$. We use $L_s$ to denote the set $\{(\beta, \alpha) : \langle s, \beta, \alpha \rangle$ is reachable$\}$. $M$ is a semi-linear system if $L_s$ is an effectively semi-linear language for each $s \in S$ (i.e., the semi-linear set of $L_s$ is computable from the description of $M$). As before, we use $w_1, \cdots, w_l$ to denote a weight measure of $\Sigma = \{a_1, \cdots, a_l\}$, and use $\Sigma_1, \cdots, \Sigma_k$ to denote $k$ subsets of $\Sigma$. We may introduce weight counters $W_1, \cdots, W_k$ into $M$ to indicate that the accumulated weight on each $\Sigma_i$ is incremented by $w_i$ whenever an activity $a_j \in \Sigma_i$ is performed. That is, on a transition $\langle s, \beta \rangle \overset{a_j}{\rightarrow} \langle s', \beta' \rangle$ in $M$, the counters are updated as follows, for each $1 \leq i \leq k$, if $a_j \in \Sigma_i$ then $W_i := W_i + w_j$ else $W_i := W_i$. Similarly, for a loose weight measure $(q_1, w_1), \cdots, (q_l, w_l)$, the counters are updated on the transition as follows: for each $1 \leq i \leq k$, if $a_j \in \Sigma_i$ then $W_i := W_i + p_j$ else $W_i := W_i$, for some $q_j \leq p_j \leq w_j$ (i.e., $p_j$ is nondeterministically chosen between $q_j$ and $w_j$). Starting with 0, the weight counters are updated along an execution path in (7.29). We say that $\langle s, \beta, \alpha, W_1, \cdots, W_k \rangle$ is reachable (under the weight measure $w_1, \cdots, w_l$) if the weight counters have values $W_1, \cdots, W_k$ at the end of an execution path in (7.29) witnessing that $\langle s, \beta, \alpha \rangle$ is
CHAPTER 7. QUADRATIC DIOPHANTINE EQUATION SYSTEMS

reachable.

Let \( y_1, \ldots, y_u \) and \( z_1, \ldots, z_v \) be distinct variables. A \((u,v)\)-formula, denoted by

\[
P([y_1, \ldots, y_u]; [z_1, \ldots, z_v]),
\]

is a Presburger formula that is a Boolean combination (using \( \land \) and \( \neg \)) of Presburger formulas over \( y_1, \ldots, y_u \) and Presburger formulas over \( z_1, \ldots, z_v \). For the \( M \) specified in above, we let \( u = |\Gamma| + l \) and \( v = l + k \). Now, we consider the \( k \)-reachability problem for \( M \): given a state \( s \) and a \((u,v)\)-formula \( P \), are there \( w_1, \ldots, w_l \in \mathbb{N} \) such that

\[
P([\#(\alpha), \#(\beta)]; [w_1, \ldots, w_l, W_1, \ldots, W_k])
\]

holds for some reachable \( \langle s, \beta, \alpha, W_1, \ldots, W_k \rangle \) (under the weight measure \( w_1, \ldots, w_l \))? The loose \( k \)-reachability problem for \( M \) can be defined similarly where the lower weights \( q_1, \ldots, q_l \) are given. From Theorems 7.6, 7.7 and 7.8, one can show the following results.

**Theorem 7.9** The 2-reachability problem is decidable for semi-linear systems.

*Proof.* It suffices to show the result by assuming that \( P \) in (7.30) is in the form of \( P_1(\#(\alpha), \#(\beta)) \land P_2(w_1, \ldots, w_l, W_1, \ldots, W_k) \), where \( P_1 \) and \( P_2 \) are two Presburger formulas. Let \( L = \{ \alpha : \text{there is } \beta \text{ such that } (\beta, \alpha) \in L_s \text{ and } P_1(\#(\alpha), \#(\beta)) \} \). Since \( L_s \) is semi-linear, it is not hard to show that \( L \) is also a semi-linear language. The result follows using Theorem 7.6 on \( L \) and \( P_2(w_1, \ldots, w_l, W_1, \ldots, W_k) \). 

**Theorem 7.10** For each \( k \), the \( k \)-reachability problem is decidable for semi-linear systems, when \( P \) in (7.30) is a disjunction of formulas in the following form:

\[
Q([\#(\alpha), \#(\beta)]; [w_1, \ldots, w_l]) \land c_1 W_1 + \cdots + c_k W_k + d_1 w_1 + \cdots + d_l w_l \sim c_0,
\]

where \( Q \) is a \((u,l)\)-formula, the \( c \)'s and \( d \)'s are integers, and \( \sim \in \{=,\neq,>,<,\geq,\leq\} \).

*Proof.* Similar to the proof of Theorem 7.10, but using Theorem 7.7 instead of Theorem 7.6. 

Theorem 7.11  
For each $k$, the loose $k$-reachability problem for semi-linear systems is decidable.

Proof. Similar to the proof of Theorem 7.10, but using Theorem 7.8 instead of Theorem 7.6.

Many machine models are essentially semi-linear systems. In the following subsections, we will study two of them: finite-state systems (and their extensions), and timed automata.

7.4.1 Finite-State Systems and Their Extensions

We start with a simple model. Consider a nondeterministic finite state system $M$, which is specified in Section 1 with a designated initial state. Notice that, in this case, the configuration alphabet $\Gamma = \emptyset$. Let $s$ be a state. Clearly, $L_s$, the set of all the activity sequences when $M$ moves from the initial state to $s$ is a regular (and hence semi-linear) language. Therefore, Theorems 7.9, 7.10 and 7.11 hold for the $M$. In this case of $M$, the formula $P$ in (7.30) is a Boolean combination of Presburger formulas on $w_1, \ldots, w_l, W_1, \ldots, W_k$ and, since $\Gamma = \emptyset$ (so there is no $\beta$ in (7.30)), Presburger formulas on the counts for activities $a_1, \ldots, a_l$ respectively (these counts are represented by $\#(x)$ in (7.30), by definition, the Parikh map of activity sequence of $x$).

Conversely, for any semi-linear language $L$, one can construct, from the semi-linear set of $L$, a regular language whose semi-linear set is the same as the semi-linear set of $L$ [76]. From the regular language, one can easily construct a $M$ and a state $s$ such that the regular language is exactly $L_s$. It is routine to establish the fact that the $k$-reachability problem is decidable (for the $M$) iff the $k$-accumulated weight problem is decidable (for the $L$). From Theorem 7.5, one can show

Theorem 7.12  
There is a fixed $k$ such that the $k$-reachability problem is undecidable for finite-state systems $M$.

In the definition of the $k$-reachability problem, the Presburger formula $P$ in (7.30) is to specify the undesired values for the $w$’s and the $W$’s. When $M$ is understood as a
design of some system, a positive answer to the instance of the $k$-reachability problem indicates a design bug. In software engineering, it is highly desirable that a design bug is found as early as possible, since it is very costly to fix a bug once a system has already been implemented. It is noticed that in a specific implementation of the design, the parameterized constants are concrete, though the values differ from one implementation to another. Of course, one may test the specification by plugging in a particular choice for the concrete values. However, it is important to guarantee that for any concrete values for the parameterized constants, the design $M$ is bug-free.

![Simplified Packet-Based Network Switch](image)

**Figure 7.1:** A Simplified Packet-Based Network Switch

**Example 7.13** Let’s look at a simplified packet-based network switch depicted in Figure 7.1. This switch has five input links $c_1, ..., c_5$ and two output links $o_1, o_2$; there are also two switching units $sw_1$ and $sw_2$ that decide which incoming packet should go to which output link. Each input link and each switching unit is associated with a buffer which can be considered as a queue; there is also a scheduling unit that decides from which queue of the input links to transfer a packet to which queue of the two switching units$^3$. Suppose that each queue of the input link $c_i$ is assigned a weight $w_i$ for $1 \leq i \leq 5$, and each time when the scheduling unit visits the queue of input link

$^3$The real-world switches may not have such a structure as the separated switching units in Figure 7.1, but here we can view them in this simplified way.
c_t, an amount of packets (assume that all packets have the same length) that is in proportion to its weight \( w_i \) will be sent to a specific switching unit\(^4\). Suppose that the scheduling unit decides which packet goes to which switching unit in a simple way: packets from input links \( c_1 \) and \( c_2 \) go to \( sw_1 \); packets from input links \( c_4 \) and \( c_5 \) go to \( sw_2 \); packets from input link \( c_3 \) go to either \( sw_1 \) or \( sw_2 \) which is nondeterministically chosen (but we further require that the difference between the counts of activities \( c_3^1 \) and \( c_3^2 \) should always be in the range of \([5,10]\), and this can be done by adding a bounded counter \( x \) to the scheduling unit). Then the design for the scheduling unit can be depicted as a labeled finite-state transition system in Figure 7.2 where labels \( a_1, a_2, \) and \( a_3^1 \) (resp. \( a_4, a_5, \) and \( a_3^2 \)) represent the scheduling unit’s visiting queues of input links \( c_1, c_2, \) and \( c_3 \) (resp. \( c_4, c_5, \) and \( c_3 \)) and forwarding their packets to \( sw_1 \) (resp. \( sw_2 \)).

Suppose that the scheduling unit is required to achieve a fairness property that no matter how the weights are assigned, the total packets sent to \( sw_1 \) must be greater than that of \( sw_2 \) and less than double that of \( sw_2 \) if only the summation of weights \( w_1, w_2, \) and \( w_3 \) is greater than that of \( w_3, w_4, \) and \( w_5 \) and less than double that of \( w_3, w_4, \) and \( w_5 \) (we assume that each connection \( c_i \) has more than \( w_i \) packets available

\(^4\)The basic idea here comes from the weighted fairness queue scheduling algorithm which is a common feature of modern network switches and whose concrete implementation may be quite complex.
CHAPTER 7. QUADRATIC DIOPHANTINE EQUATION SYSTEMS

at any time); i.e.,

\[(w_3 + w_4 + w_5) < w_1 + w_2 + w_3 < 2(w_3 + w_4 + w_5) \land (W_1 \leq W_2 \lor W_1 \geq 2W_2)\]

is unsatisfiable, where \(W_1\) (resp. \(W_2\)) denotes the accumulated weights of activities \(a_1, a_2,\) and \(a_3^1\) (resp. \(a_4, a_5,\) and \(a_3^2\)). From Theorem 7.9, we know this fairness property can be automatically verified. When there are \(k > 2\) switching units involved in the example switch, a fairness property can be similarly formulated as a conjunction of the fairness between any two servers. In this case, the fairness property is hard to be automatically verified, because of Theorem 7.12.

\[\square\]

Remark. Note that if we do not assume that each connection \(c_i\) has more than \(w_i\) packets available at any time, then the weight of each activity \(a_i\) is actually a range \([0, w_i]\). Thus from Theorem 7.11, we know the fairness property formulated in the above example can still be automatically verified, even when the switch has \(k > 2\) switching units.

\(M\) can be further generalized as a pushdown system after being augmented with a pushdown stack. Each transition in \(M\) now is in the following form: \(s \rightarrow^{a, b, \gamma} s'\), indicating that \(M\) moves from state \(s\) to state \(s'\) while performing an activity \(a\) and also updating the stack (replacing the top symbol \(b\) in the stack by a stack word \(\gamma\)). There are only finitely many transitions in the description of \(M\). Initially, the stack contains a designated initial symbol (i.e., an initialized stack) and the machines stays at the initial state. Now, the configuration alphabet \(\Gamma\) is exactly the stack alphabet. Notice that, for this model of \(M\), \(L_s\) is a permutation of a context-free (hence semi-linear) language. Therefore, \(M\) is still a semi-linear system. \(M\) can be further augmented with a finite number of reversal-bounded counters. A nonnegative integer counter is reversal-bounded [54] if it alternates between a nondecreasing mode and a non-increasing mode (and vice versa) for a given finite number of times, independent of the computations. Hence, a transition in \(M\), in addition to the stack operation, can increment/decrement a counter by one and test a counter against zero. When the counter values are encoded as unary strings (using new symbols in addition to the stack alphabet), it is not hard to show that the language of \(L_s\) is a semi-linear
CHAPTER 7. QUADRATIC DIOPHANTINE EQUATION SYSTEMS

language [54]. Hence, Theorems 7.9, 7.10 and 7.11 apply to all these $M$. In this case of $M$, the formula $P$ in (7.30) is a Boolean combination of Presburger formulas on $w_1, \ldots, w_l, W_1, \ldots, W_k$ and Presburger formulas on the following counts (standing for $\#(\alpha)$ and $\#(\beta)$ in (7.30)):

- the counts for activities $a_1, \ldots, a_l$ respectively,
- the counts for individual stack symbols appearing in a stack,
- the values for reversal-bounded counters.

Example 7.14 Still consider the design for the scheduling unit of the network switch in Figure 7.1, but this time we take the switching units into consideration. We assume that switching unit $sw_1$ has an unbounded buffer while $sw_2$ has a buffer with bounded length $L$, the a simple design for the scheduling unit can be described by the following three concurrent processes in C-like pseudo code:

```c
Integer w[5];
Integer x, y, z;
Integer L = 1000;
process SchedulingUnit
    for (i = 1; i <= 5; i++) do
        switch (i)
            case 1, 2:
                y = y + w[i];
                break;
            case 3:
                nondeterministic
                    if (x <= 10) then
                        x + +;
                        y = y + w[i];
                    end if
                    if (x >= 5) then
                        x - -;
```
if \( z + w[i] \leq L \) then
\[
z = z + w[i];
\]
end if
end if
end nondeterministic
break;
case 4, 5:
if \( z < L \) then
\[
z = z + w[i];
\]
end if
break;
end switch
end for
end process
process SwitchingUnit1
while \( y > 0 \) do
\[
y = -;
\]
end while
end process
process SwitchingUnit2
while \( z > 0 \) do
\[
\quad z = -;
\]
end while
end process

This design can be viewed as a finite-state system with an unbounded free counter \( y \). We know the fairness property formulated in Example 7.13 can also be automatically verified for this design.
7.5 Summary

In this chapter, we showed that some special classes of quadratic Diophantine equation/inequality systems are decidable. Using the decidability results, we then looked at some applications to verification problems. In particular, we studied an application concerning weighted semi-linear languages by assigning a weight to each symbol in the language alphabet. The verification problems are related to some interesting problems in verification of infinite-state systems, in particular those containing parameterized constants.
Chapter 8

Conclusions

Establishing a science and technology foundation for achieving predictable quality in component-based systems is considered to be an key factor to the long-term success of component-based software development and it has attracted lots of research efforts in the area of software engineering. As part of the efforts, this work deals with some fundamental issues on a decompositional and hybrid approach for the automatic verification of component-based systems. Our approach works on an abstract model that is essentially a labeled transition system with synchronous communications. One striking point of our approach is that verifying a component-based system can be reduced to verifying an individual component. This decompositional nature of our approach enables us to avoid the major pitfalls of existing verification/testing approaches that work directly on an entire system (e.g., the “state-explosion” problem of model checking, the difficulty of integration testing on a system of concurrent black-boxes, etc.). On the other hand, our approach combines the model checking techniques with traditional black-box testing techniques to handle systems with black-box components. In the approach, model checking is used to derive verification conditions for an individual black-box component, which are in turn used to derive test cases for testing the black-box component. We first studied the possibility of verifying a system with black-box components through testing by introducing a theoretic tool called Oracle Automata. Then we studied specific model checking (both LTL and CTL) algorithms
for systems with only one and finite-state black-box. Next, we showed a decompositional technique on testing a system with multiple black-boxes. We also considered the problem of verifying a system with only one component but against non-temporal properties, as well as verifying a system with only one but infinite-state component. These studies made us believe the advantages of our approach, but also inspired us to further study the practical issues that are crucial to the application of our approach to real-world systems.

8.1 Related Work

Most of current work on the quality assurance problem for component-based systems is based on the traditional software testing techniques and considers the problem from component developers' point of view; i.e., how to ensure the quality of components before they are released.

Voas [91, 92] proposed a component certification strategy with the establishment of independent certification laboratories performing extensive testing of components and then publishing the results. Technically, this approach would not provide much improvement for solving the problem, since independent certification laboratories can not ensure the sufficiency of their testing either, and a testing-based technique alone is not sufficient to establish a solid confidence in the quality of a reliable software component. Some researchers [86, 75] suggested an approach to augment a component with additional information to increase the customer’s understanding and analyzing capability of the component behavior. A related approach [93] is to automatically extract a finite-state machine model from the interface of a software component, which is delivered along with the component. This approach can provide some convenience for customers to test the component, but again, how much a customer should test is still a big problem. To address the issue of testing adequacy, Rosenblum defined in [84] a conceptual basis for testing component-based software, by introducing two notions of $C$-adequate-for-$P$ and $C$-adequate-for-$M$ (with respect to certain adequacy criteria) for adequate unit testing of a component and adequate integration testing for a component-based system, respectively. But this is still a purely testing-based
strategy. In practice, how to establish the adequacy criteria is an unclear issue.

Recently, Bertolino et. al. [10] recognized the importance of testing a software component in its deployment environment. They developed a framework that supports functional testing of a software component with respect to customer’s specification, which also provides a simple way to enclose a component with the developer’s test suites which can be re-executed by the customer. Yet their approach requires the customer to have a complete specification about the component to be incorporated into a system, which is not always possible. McCamant and Ernst [72] considered the issue of predicting the safety of dynamic component upgrade, which is part of the problem we consider. But their approach is completely different since they try to generate some abstract operational expectation about the new component through observing a system’s run-time behavior with the old component.

In the formal verification area, there has been a long history of research on verification of systems with modular structure. A key idea [65, 63, 51] in modular verification is the assume-guarantee paradigm: A module should guarantee to have the desired behavior once the environment with which the module is interacting has the assumed behavior. There have been a variety of implementations for this idea (see, e.g., [48, 5, 77, 36, 21, 94]). The assume-guarantee ideas can be applied to our problem setup if we consider the unspecified component as the host system’s environment (though this is counter-intuitive). But the key issue with the assume-guarantee style reasoning is how to obtain assumptions about the environment. Giannakopoulou et. al. [45, 44] introduced a novel approach to generate assumptions that characterize exactly the environment in which a component satisfies its property. Their idea is the closest to ours, still there are non-trivial differences: (1) theirs is a purely formal verification technique (model checking) while we combine both model checking and black-box testing to handle systems with unspecified components; and (2) theirs uses a labeled transition system to specify the reachability property of a system while we use CTL formulas, which are more expressive and harder to manipulate. Although not within the assume-guarantee paradigm, Fisler et. al. [40, 68] introduced a similar idea of deducing a model checking condition for extension features from the base feature for model checking feature-oriented software designs. Unfortunately, their algorithms
CHAPTER 8. CONCLUSIONS

are not sound (have false negatives). Furthermore, their approach is not applicable to component-based systems where unspecified components exist. This paper is also different from our previous work [98] where an automata-theoretic approach is used to solve a similar LTL model checking problem.

In the past decade, there has also been increasing interest in combining model checking and testing techniques for system verification. But most of the work only utilizes model-checkers' ability of generating counter-examples from a system's specification to produce test cases against an implementation [20, 52, 39, 42, 7, 11, 6]. Callahan et. al. [20] used the model-checker SPIN [52] to check a program's execution traces generated during white-box testing and to generate new test-cases from the counter-example found by SPIN; in [39], SPIN was also used to generate test-cases from counter-examples found during model checking system specifications. Gargantini and Heitmeyer [42] used SMV to both generate test-cases from the operational SCR specifications and as test oracles. In [7, 11], Ammann et. al. also exploited the ability of producing counter-examples with the model-checker SMV [59]; but their approach is by mutating both specifications and properties such that a large set of test cases can be generated. (A detailed introduction on using model-checkers in testing can be found in [6]).

Peled et. al. [79, 47, 78] studied the issue of checking a black-box against a temporal property (called black-box checking). But their focus is on how to efficiently establish an abstract model of the black-box through black-box testing, and their approach requires a clearly-defined property (LTL formula) about the black-box, which is not always possible in component-based systems. Kupferman and Vardi [63] investigated module checking by considering the problem of checking an open finite-state system under all possible environments. Module checking is different from the problem in (*) mentioned in this chapter in the sense that a component understood as an environment in [63] is a specific one.

Our idea of deriving from the system specification a verification condition for an individual component is related with the work by Fisler et. al. [40, 68] who proposed an approach to deducing a model checking condition for extension features from the base feature in feature-oriented software designs. Their approach relies totally on
CHAPTER 8. CONCLUSIONS

model checking techniques; and most of all, their algorithms have false negatives and do not handle LTL formulas. Our idea of decompositional testing is based on the observation that global behaviors of a concurrent system can be projected onto behaviors for each constituent component. Moreover, the behaviors of each individual component are also constrained by the behaviors of other components (because of synchronizations). The similar observations have also been made in [24, 25] as Context Constraints for compositional reachability analysis and used in [61, 60] for structural testing of concurrent programs.
Appendix A

Proofs Omitted In The Dissertation

Disclaimer. The following proofs are put in this appendix because the main ideas in the proofs are from the co-authors of the relevant papers. Thus they shall not be considered to be an integral part of this dissertation.

A.1 Proofs Omitted From Chapter 3

A.1.1 Proof of Theorem 3.1

Proof. Assume that $BT(n)$ is computable. Let $M_0$ be a fixed PDA that accepts language $\Sigma^*$. Suppose that $M_0$ has $n_0$ states. Now, we are going to solve the following totalness problem:

Given: a PDA $M$,

Question: $L(M) = \Sigma^*$?

Assume that $M$ has $n$ states and without loss of generality, $n \geq n_0$ (otherwise one can add dummy states into $M$). From this $n$, one computes $BT(n)$ and makes sure that $w \in L(M)$ for each $w \in \Sigma^*$ that is not longer than $BT(n)$ (there are finitely many such $w$'s). If this is true, then Question returns yes. Otherwise, Question returns no. According to the definition of $BT$, this indeed gives an algorithm to solve
the totalness problem. This is a contradiction, since the totalness problem is known
undecidable. The theorem follows. ■

A.1.2 Proof of Theorem 3.2

Proof. (a). Let $O_1, \ldots, O_t$ be any $t$ oracles in $\text{DFA}(n)$. For each $1 \leq i \leq t$, we use $A_i$ to denote a $\text{DFA}(n)$ that recognizes $O_i$. Now, we construct a finite automaton $A$
that simulates $M^{\text{DFA}(n)}(O_1, \ldots, O_t)$ as follows. $A$ does not have an input. Whenever
$M$ reads an input symbol $a$, $A$ guess an input symbol and makes sure that it is $a$. Whenever $M$ writes a symbol $a$ to the $i$-th query tape, $A$ runs $A_i$ on this symbol. Whenever $M$ queries the current content of the $i$-th query tape, $A$ answers the query by checking whether $A_i$ is in an accepting state (notice that since $A_i$ is a DFA, a negative query can be faithfully answered). Whenever $M$ executes a transition that resets the $i$-th query tape, $A$ brings $A_i$ directly to $A_i$’s initial state. In each case, $A$ performs the same state transition as in $M$. Initially, $A$ stays at the initial state of $M$ and each $A_i$ also stays at its own initial state. Clearly, $M^{\text{DFA}(n)}(O_1, \ldots, O_t)$
accepts a nonempty language iff $A$ has a run that ends with an accepting state of $M$.
Moreover, the run can be further restricted to be not longer than the number $|A|$ of
states in $A$. Notice that within this length of the run, $M$ can not query an oracle
with query strings longer than the length. Hence, the query bound is at most $|A|$, which can be easily calculated as $O(n^t \cdot |M|)$. Notice that this query bound does not depend on the particular choices of $O_i$’s and hence $A_i$’s. Therefore, the $M^{\text{DFA}(n)}$ is
$O(n^t \cdot |M|)$-testable.

(b). The result follows from the fact that a language in $\text{FA}(n)$ is contained in
$\text{DFA}(2^n)$.

A.1.3 Proof of Theorem 3.3

Proof. (a.1). Let $M$ be an OFA that is single (i.e., $t = 1$) and is associated with an
oracle $O \in \text{PDA}$. $M$ first guesses a query string and writes it on the query tape. On
a negative query result, $M$ enters the accepting state. Clearly, $M$ is 1-query, and, $M$
accepts a nonempty language iff $O \neq \Sigma^*$. Now $M$ is not testable since, otherwise, the
totalness problem for context-free languages would become decidable.

(a.2). Let \( A \) be a deterministic two-counter machine where \( x \) and \( y \) are the counters and \( S \) is the set of states in \( A \). Recall that a transition in \( A \) leads from \( s \) to \( s' \) while updating the counters (i.e., incrementing/decrementing the counters by 1 or testing for 0). We may use a string \( C \)

\[
\#s#1^x#1^y#
\]

to denote a configuration of \( A \) where \( s \) is the state and \( x, y \) are the counter values. Notice that in the string, \( 1^x \) is the unary representation of value \( x \) (\( x \) number of 1’s). For the same configuration, we use \( \tilde{C} \) to denote the reverse configuration

\[
\$s\$1^y\$1^x\$.\]

A string is *even-valid* if it is in one of the following two forms for some \( m \):

\[
C_0\tilde{C}_1 \cdots C_m\tilde{C}_{m+1}
\]

(A.1)

or

\[
C_0\tilde{C}_1 \cdots \tilde{C}_mC_{m+1},
\]

(A.2)
such that each \( C_i \) is a configuration with \( C_0 \) being the initial configuration (with the state being the initial state and the counters being 0). Additionally, for each even \( i \leq m, C_i \rightarrow C_{i+1}; \) i.e., there is a transition in \( A \) that leads from \( C_i \) to \( C_{i+1} \). The string is *odd-valid* if the additional condition is changed to be: for each odd \( i \leq m, C_i \rightarrow C_{i+1} \). Now, we use \( L_{\text{even}} \) (resp. \( L_{\text{odd}} \)) to denote the set of even-valid (resp. odd-valid) strings. It is left to the reader to verify that both \( L_{\text{even}} \) and \( L_{\text{odd}} \) are in DPDA. Now, we define \( O \) to be the union of the following two languages: \( L_{\text{even}} \) and \( \{w\# : w \in L_{\text{odd}} \} \). Notice that, because of the additional suffix \# appended after each odd-valid string, these two languages are disjoint. Obviously, \( O \) is in PDA. We now construct a single OFA \( M \) that works as follows. \( M \) keeps guessing a configuration and writes it to the query tape. We use \( C_i \) to denote the result of the \( i \)-th guess (\( i \) starts from 0). In fact, when \( i \) is odd, \( M \) writes the reverse configuration \( \tilde{C}_i \) instead of \( C_i \) to the tape. Nondeterministically, \( M \) decides to enter the accepting state. Before
it does this, $M$ first makes sure that the most recently written configuration is an accepting configuration of $A$ (i.e., the configuration contains the accepting state of $A$). Then, $M$ makes a positive query to $O$. On a yes answer, $M$ writes an additional symbol $\#$ to the query tape, and performs one more positive query to $O$. A yes answer on this latter query leads $M$ to accept. Notice that, on $M$’s accepting, the content of the query tape forms a halting execution sequence of configurations of $A$. Therefore, $M$ accepts a nonempty language iff $A$ enters the accepting state (i.e., halts). From here, the result follows, since it is undecidable to decide whether a two counter machine halts [73], and, clearly, $M$ is positive, 2-query, and single.

(a.3). Still, we let $A$ be the two-counter machine in (a.2). We use $O_1$ (resp. $O_2$) to denote $L_{\text{even}}$ (resp. $L_{\text{odd}}$). Notice that both $O_1$ and $O_2$ are in DPDA. Now, we construct an OFA $M$ that has two query tapes and works similarly as in (a.2): $M$ keeps guessing a configuration and writes it to both tapes (for the $i$-th guess with $i$ being odd, a reverse configuration is written). Nondeterministically, $M$ decides to enter the accepting state. To do this, $M$ performs two positive queries: querying the first tape to oracle $O_1$ and querying the second tape to oracle $O_2$. At this time, $M$ also makes sure that the most recently written configuration is an accepting configuration of $A$. Similar to (a.2), $M$ accepts a nonempty language iff $A$ halts. The result follows, noticing that $M$ is 2-query, positive, and the oracles are from DPDA.

(b.1). Let $O$ be any prefix-closed oracle in PDA($n$). We use $A'$ to denote a PDA($n$) that accepts $O$. Now we construct a pushdown automaton $A$ to simulate $M_{\text{PDA}(n)}^O$ as follows. Whenever $M$ reads an input symbol $a$, $A$ guess a symbol and makes sure that it is $a$. Whenever $M$ writes a symbol $a$ to the query tape, $A$ runs $A'$ on this symbol. Whenever $M$ performs a query, $A$ guesses and checks later one of the following two cases:

CASE 1. the current query is the last query before the next reset transition or before $M$ accepts. In this case, $A$ makes sure that $A'$ is in an accepting state (i.e., the query is positive),

CASE 2. the current query is not the last query before the next reset transition or before $M$ accepts. In this case, $A$ assume that the query
returns yes. This is valid, recalling that the oracle is prefix-closed.

Whenever \( M \) executes a reset transition, \( A \) brings \( A' \) back to its initial state (and also cleans up the stack). On each transition of \( M \), \( A \) performs the same state transition, and additionally, \( A \) reads an input symbol \( a \) from \( A' \)'s input tape. Initially, \( A \) stays at the initial state of \( M \) and each \( A' \) also stays at its own initial state (with an empty stack). Notice that \( A \) is indeed a pushdown automaton that only accepts unary words in the form of \( a^B \) for some \( B \geq 0 \). Clearly, \( M^{DFA(n)}(O) \) accepts a nonempty language iff \( A \) does. If \( A \) accepts a nonempty language, then what is the length of the shortest word in the language? The length can be calculated as follows. The number of states in \( A \) is \( O(|M| \cdot n) \). One may use a textbook technique to translate \( A \) into a context-free grammar \( G \) in Chomsky-Normal Form and to calculate the desired length, which is bounded by \( 2^{O(|M|^2 \cdot n^3 \cdot |\Sigma|)} \). \(^1\) Notice that \( A \) accepts a unary word \( a^B \) iff \( M \) has an accepting run on some input word where the length of the run is exactly \( B \). Obviously, during the run, \( M \) does not query the oracle with query strings longer than \( B \). From here, we may conclude that \( 2^{O(|M|^2 \cdot n^3 \cdot |\Sigma|)} \) is the query bound for \( M \). Since the bound is independent of the choice of \( O \), the result follows.

(b.2). The proof of (b.1) still works here since \( M \), being 1-query, need not worry about CASE 2 in the proof of (b.1).

(b.3). Similar to (b.2), one need not CASE 2 in the proof of (b.1).

(b.4). Let \( O \) be any oracle in \( DPDA(n) \). We use \( A' \) to denote a DPDA(\( n \)) that accepts \( O \). Now we construct a pushdown automaton \( A \) to simulate \( M^{DPDA(n)}(O) \). \( A \) works exactly as in (b.1) except when \( M \) performs a query. In this case, \( A \) obtains the query result by inspecting whether the deterministic \( A' \) is in an accepting state. The result follows after the exact query bound analysis that was done in (b.1).

\[ \text{A.1.4 Proof of Theorem 3.4} \]

Proof. (a.1). Let \( A \) be a deterministic two-counter machine specified in the proof of Theorem 3.3 (a.2). That is, \( A \) has two counters \( x \) and \( y \), and, we let \( s_0, \ldots, s_m \), for some \( m \), are the states in \( A \). Without loss of generality, we assume that \( s_0 \) is

\(^1\)Recall that a PDA each time pushes/pops at most one symbol.
the initial state and it is not an accepting state. In particular, we define, for each state $s_i$, $|s_i|$ to be the index $i$. For the purpose of describing the OFA to be built, we introduce an alphabet $\Sigma$ that contains the following symbols:

\[
\begin{align*}
\theta^+, \theta^-, \\
\hat{\theta}^+, \hat{\theta}^-,
\end{align*}
\]

\[
\begin{align*}
a^+, a^-, b^+, b^-,
\end{align*}
\]

\[
\begin{align*}
\hat{a}^+, \hat{a}^-, \hat{b}^+, \hat{b}^-.
\end{align*}
\]

A word $w$ in $\Sigma^*$ corresponds to a pair of configurations, called the pre-configuration $C_w$ and the post-configuration $C'_w$, as follows. In the pre-configuration $C_w$,

- the state is $s_i$, where the index $i$ is $|w|_{\theta^+} - |w|_{\theta^-}$ (recalling that $|w|_{\theta^+}$ is the number of symbols $\theta^+$ appearing in $w$);
- the value for counter $x$ is $|w|_{a^+} - |w|_{a^-}$;
- the value for counter $y$ is $|w|_{b^+} - |w|_{b^-}$.

In the post-configuration $C'_w$,

- the state is $s'_i$, where the index $i'$ is $|w|_{\hat{\theta}^+} - |w|_{\hat{\theta}^-}$;
- the value for counter $x$ is $|w|_{\hat{a}^+} - |w|_{\hat{a}^-}$;
- the value for counter $y$ is $|w|_{\hat{b}^+} - |w|_{\hat{b}^-}$.

Not every $w$ will make two legal configurations; one has to further restrict that indexes $i$ and $i'$ are in the range of $0..m$, counter values in both of the configurations are nonnegative. We call this restriction as $R_{\text{conf}}$.

Each instruction in $A$ is to increment/decrement a counter by 1 or test for 0. For example, with respect to counter $x$, an instruction can be in one of the following three forms:

1. $s : x := x + 1$, goto $s'$;
2. $s : x := x - 1$, goto $s'$;
3. $s : \text{if } x = 0 \text{ then goto } s' \text{ else goto } s''$. 


APPENDIX A. PROOFS OMITTED IN THE DISSERTATION

Instructions for counter $y$ can be defined similarly. For each instruction $I$, one can formulate a restriction, called $R_I$, on $w$ such that the pre-configuration $C_w$ reaches the post-configuration $C'_w$ after firing $I$. For instance, when $I$ is in Form (1), we require that

- both $C_w$ and $C'_w$ are legal configurations; i.e., $R_{\text{conf}}$ is satisfied;
- the state in $C_w$ is $s$; i.e., $|w|_{\theta^+} - |w|_{\theta^-} = [s]$. The state in $C'_w$ is $s'$; i.e., $|w|_{\hat{\theta}^+} - |w|_{\hat{\theta}^-} = [s']$;
- the value for counter $x$ in $C'_w$ is equal to the value for counter $x$ in $C_w$ plus 1; i.e., $|w|_{\alpha^+} - |w|_{\alpha^-} + 1 = |w|_{\hat{\alpha}^+} - |w|_{\hat{\alpha}^-}$;
- the value for counter $y$ does not change; i.e., $|w|_{b^+} - |w|_{b^-} = |w|_{\hat{b}^+} - |w|_{\hat{b}^-}$.

When $I$ in the other forms, similar $R_I$ can be defined.

Clearly, each $R_I$ defines a semi-linear commutative language over $\Sigma$. Let $R = \bigcup_{I\in \mathbb{I}} R_I$, where $\mathbb{I}$ is the set of all instructions in $A$. $R$ is also a semi-linear commutative language.

Now, we are ready to build the single OFA $M$. $M$ is associated with the semi-linear commutative oracle $R$ and works in rounds. We first sketch the ideas behind the following construction. At the beginning of each round, the content $w$ of $M$’s current query tape already encodes the pre-configuration $C_w$ and the post-configuration $C'_w$ such that $C_w \rightarrow C'_w$. That is, $C_w$ reaches $C'_w$ by firing some instruction in $A$; i.e. $w \in R$. The job of the round is to change the tape content from $w$ to $w'$. The new content $w'$ also encodes $C_w'$ and $C'_w'$, such that

- $C_w' \rightarrow C'_w$, and,
- $C_w'$ is exactly $C'_w$.

The first item can be ensured by performing a positive query to the oracle $R$. The difficulty is how to ensure the second item, which essentially creates an execution chain of configurations in $A$. Fortunately, the difference between $C_w$ and $C'_w$ can be remembered by $M$ and hence can be used to update $C_w$ to $C'_w$. Below is the formal
APPENDIX A. PROOFS OMITTED IN THE DISSERTATION

construction of $M$. There are four phases in each round. In the first phase of the $r$-th round ($r$ starts from 1), $M$ guesses a state $s^r$. Keep in mind that the states in $C_w$ and $C'_w$ are $s^{r-1}$ and $s^{r-2}$, respectively, where $w$ is the current tape content. The job of the second phase is to change the current tape content such that the states in the pre-configuration and the post-configuration encoded by the new content are $s^r$ and $s^{r-1}$, respectively. Formally, in the second phase, the following activities are performed, assuming that $s^0$ and $s^{-1}$ are defined to be $s_0$:

- if $[s^r] \geq [s^{r-1}]$, then $M$ writes $q$ number of symbols $\hat{\theta}^+$ to the query tape, where $q = [s^r] - [s^{r-1}]$;

- if $[s^r] < [s^{r-1}]$, then $M$ writes $q$ number of symbols $\hat{\theta}^-$ to the query tape, where $q = [s^{r-1}] - [s^r]$;

- if $[s^{r-1}] \geq [s^{r-2}]$, then $M$ writes $q$ number of symbols $\theta^+$ to the query tape, where $q = [s^{r-1}] - [s^{r-2}]$;

- if $[s^{r-1}] < [s^{r-2}]$, then $M$ writes $q$ number of symbols $\theta^-$ to the query tape, where $q = [s^{r-2}] - [s^{r-1}]$.

An update $\Delta$ is a pair $(\Delta_x, \Delta_y)$ where $\Delta_x, \Delta_y \in \{1, 0, -1\}$. It indicates that, after the update, the amount of the change to counter $x$ (resp. $y$) is $\Delta_x$ (resp. $\Delta_y$). The job of the third phase is to change the current tape content such that the new counter values in the post-configuration encoded by the new content are the result of a guessed update on the old counter values in the post-configuration encoded by the old content. Formally, in the third phase, $M$ guesses an update $\Delta'$ and does the following:

- if $\Delta'_x = 1$, then $M$ writes a symbol $\hat{a}^+$ to the query tape;

- if $\Delta'_x = -1$, then $M$ writes a symbol $\hat{a}^-$ to the query tape;

- if $\Delta'_x = 0$, then $M$ writes nothing.

The job of the fourth phase is to change the current tape content such that the new counter values in the pre-configuration encoded by the new content are the result
of the update performed in the last round on the old counter values in the pre-
configuration encoded by the old content. Formally, in the fourth phase, (we define
\( \Delta^0 = (0, 0) \))

- if \( \Delta_x^{r-1} = 1 \), then \( M \) writes a symbol \( a^+ \) to the query tape;
- if \( \Delta_x^{r-1} = -1 \), then \( M \) writes a symbol \( a^- \) to the query tape;
- if \( \Delta_x^{r-1} = 0 \), then \( M \) writes nothing.

At the end of the fourth phase, \( M \) makes a positive query to the oracle with the
current tape content and then starts a new round. Nondeterministically at the end
of some round, \( M \) guesses that \( A \) halts. \( M \) accepts after making sure that the state
\( s^r \) guessed in the round is the accepting state of \( A \).

Since \( A \) is deterministic, for any word \( w, w \in R \) implies that there is a unique
\( I \in \mathbb{I} \) satisfying \( w \in R_I \). From this property, it is not hard to show that \( M \) accepts a
nonempty language iff \( A \) has a halting execution; i.e., \( A \) halts. The result follows.

(a.2). Still, let \( A \) be a deterministic two-counter machine specified in the proof
of (a.1). Similar to what we have mentioned in the proof of Theorem 3.3 (a.2), a
configuration \( C \) of \( A \) can be specified as a string, denoted by \([C]\),

\[
\$\theta^i\$a^x\$b^y\$.
\]

In above, \( \theta^i \) is to encode the state \( s_i, 1 \leq i \leq m \), and, \( a^x \) and \( b^y \) are for the counter
values. Additionally, we may use the following string, denoted by \( \langle C \rangle \),

\[
\#\hat{\theta}^i\#\hat{a}^x\#\hat{b}^y\#
\]

to represent the same configuration \( C \). Similar to what we have in (a.1), one can con-
struct from \( A \) a semi-linear commutative language \( R \) (over alphabet \( \{\$\, \#, \theta, \hat{\theta}, a, \hat{a}, b, \hat{b}\} \)) such that, for any two configurations \( C \) and \( C' \), the string \([C]\langle C'\rangle \in R \) iff
\( C \rightarrow C' \).

We now construct an OFA \( M \) associated with two oracles (both are \( R \)) to simulate
\( A \). Initially, \( M \) writes the initial configuration of \( A \) to the first query tape, in the form
APPENDIX A. PROOFS OMITTED IN THE DISSERTATION

of \([C_0]\). Then, \(M\) works in rounds. The \(r\)-th round \((r\) starts from 1\) is to perform the following two items:

- \(M\) guesses a configuration \(C_r\). Then \(M\) writes \(\langle C_r \rangle\) to the first query tape. In parallel to this, \(M\) also writes \([C_r]\) to the second query tape.

- \(M\) performs a positive query with the content of the first query tape and right after this, \(M\) erases the first tape (so \(M\) is memoryless).

The above description only works when \(r\) is odd. In the case when \(r\) is even, one needs to replace “first” with “second” (and vice versa) in the description of the two items. Nondeterministically at the end of some \(r\)-th round, \(M\) guesses that it is the time to accept. Then \(M\) makes sure that the state encoded in \(C_r\) is the accepting state of \(A\). Clearly, \(M\) accepts a nonempty language iff \(A\) has a halting execution; i.e., \(A\) halts. The result follows.

(b.1). Let \(M\) be a \(k\)-query \(\text{OFA}^{\text{LIN}(n)}\). Without loss of generality, we assume that \(M\) makes exactly \(k\) queries in an accepting run. Also, we assume that the queries are made to oracles \(O_1, \cdots, O_k\), respectively. Let \(A_1, \cdots, A_k\) be reversal-bounded DCMs (whose input has an end marker) with characteristic \(n\) and recognizing \(O_1, \cdots, O_k\), respectively. We now build another reversal-bounded NCM \(A\) to simulate \(M\). \(A\) starts with the initial state of \(M\) and simulates \(M\)’s moves. When \(M\) reads the input tape, \(A\) does nothing to its own input. When \(M\) write a symbol to a blank query tape, \(A\) makes a guess on one of the following two cases:

- there is a query to oracle \(O_i\) that will be performed on the tape before the next reset (if any) happens. In this case, \(A\) starts running \(A_i\) on every symbol that is written on the tape subsequently until \(M\) indeed queries. For each such write, \(A\) reads a symbol \(a\) from its own input tape. At the time of querying, checking whether \(A_i\) enters an accepting state gives the query answer.

- there will not be a query to oracle \(O_i\) that will be performed on the tape before the next reset (if any) happens. In this case, \(A\) does nothing on every write to this tape until the tape is reset.
APPENDIX A. PROOFS OMITTED IN THE DISSERTATION

Notice that, on every move of \( M \), \( A \) faithfully simulates \( M \)'s state transitions. \( A \) accepts when \( M \) enters an accepting state. Clearly, \( A \) accepts a nonempty language iff \( M \) does. In particular, \( A \) only accepts a unary language. A word \( a^B \) is accepted by \( A \) iff \( M \) has a successful run on some input word where query strings are not longer than \( B \). Since \( A \) is an NCM, to estimate \( B \), it is sufficient for us to calculate a characteristic for \( A \), which is a product of \( A_1, \ldots, A_k \), along with the finite state transition graph of \( M \). One can show that a characteristic of \( A \), and hence a query bound for \( M \), is \( O(n^k|M|^\mu) \).

(b.2). Suppose \( \Sigma = \{ a_1, \ldots, a_k \} \). Let \( O \) be a prefix closed language in \( \text{LIN}(n) \) and accepted by a reversal-bounded DCM \( A \) with characteristic \( n \). Observe that, from the description of \( A \), one can effectively compute a finite number of “corner points”

\[
(x_1, \ldots, x_k)
\]

such that each \( x_i \) is in \( \{0, \ldots, \infty\} \), and \( O \) is the union of all \( \{ w : |w|_{a_i} \leq x_i, 1 \leq i \leq k \} \). This gives the fact that \( O \) is regular. Hence, the result follows from Theorem 3.2. However, since we currently are unable to give a good estimation of the sizes for the corner points, the exact query bound for (b.2) is unknown.

(b.3). Since \( M \) is memoryless and single, \( M \) resets the query tape after each query. Now, we define another \( M' \) that is exactly as \( M \) but starts from a state \( s \) (in \( M \)) with blank query tape and ended with a reset right after a query (this is the only query that \( M' \) performs). Clearly, the maximal query bound for this \( M' \) (among all \( s \)) governs the desired query bound for \( M \). Notice that \( M' \) is 1-query, the result follows from (b.1).

A.1.5 Proof of Theorem 3.6

Proof.

We need only to prove the first statement. Let \( O_1, \ldots, O_t \) be any \( t \) oracles in \( \text{DFA}(n) \). For each \( 1 \leq i \leq t \), we use \( A_i \) to denote a \( \text{DFA}(n) \) that recognizes \( O_i \). Now we construct an FA \( A \) that simulates \( M^{\text{DFA}(n)}(O_1, \ldots, O_t) \). Let \( s \) be any fixed state of \( M \). \( A \) works almost the same as the \( A \) in the proof of Theorem 3.2 (a). The difference
is that, during $A$’s simulation on $M$, $A$ nondeterministically remembers point when $M$ is at state $s$. Then, $A$ continues the simulation and makes sure that, after the point, $M$ has read at least one input symbol and has passed the accepting state for at least once. At this time, $A$ still continues the simulation and, nondeterministically, it guesses that it is time to accept. At this moment, it makes sure that the current states of $M$ and all $A_i$’s are exactly the same as those at the remembered point. It is not hard to show that $A$ accepts a nonempty language for some $s$ iff $M_{DFA}^{PDA}(n)(O_1, \cdots, O_i)$ does. The result follows immediately, since $A$ has $O(n^2 \cdot |M|)$ states.

\[ \blacksquare \]

A.1.6 Proof of Theorem 3.7

Proof. (1) Since $M_{PDA}^{PDA(n)}$ is positive, single and 1-query, on an accepting $\omega$-run, $M_\omega$ does not perform any queries after certain point when an accepting state is reached. Before the point, $M_\omega$ behaves like the 1-query OFA $M_{PDA}^{PDA(n)}$. After the point, $M_\omega$ behaves like a Buchi automaton (without accessing to the oracle). Hence, it suffices to consider the query bound for testing the emptiness of the 1-query OFA, shown in Theorem 3.3 (b.2).

(2) Without loss of generality, we assume that an accepting $\omega$-run of $M_\omega$ queries the oracle for infinitely many times. One can also show that, on the run, there are two points such that at both points $M_\omega$ is at the same state and is right after a reset (resulting from a query since $M_\omega$ is memoryless). Furthermore, in between these two points, the run passes an accepting state and consumes at least one input symbol. In fact, the existence of the two points is the iff-condition on whether the $\omega$-run is an accepting run. Checking the existence can be reduced to the case of Theorem 3.3 (b.3). The result follows.

(3) Let $O$ be an oracle in DPDA(n) and $M_\omega$ be associated with $O$. On an accepting $\omega$-run of the $M_\omega$, there are two cases to consider:

Case 1. There are infinite number of reset transitions on the $\omega$-run. Recall that each reset makes the query tape blank. The existence of such an $\omega$-run can be fully decided by answering the following question for each pair of states $s$ and $s'$ in $M$:
Can $M$ start from state $s$ with blank query tape and end with state $s'$ also with blank query tape during which at least one input symbol is read and an accepting state is passed? The questions can be answered with a query bound

$$2^{O(|M|^3 \cdot n^2 \cdot |\Sigma|)}$$

(A.3)

using Theorem 3.3 (b.4).

**Case 2.** There are a finite number of reset transitions on the $\omega$-run. The $\omega$-run can be split into two parts. The first part is from the initial state $s_0$ to a state, say $s$, right after the last reset transition. The second part, starting from $s$ and with a blank query tape, is the suffix of the $\omega$-run right after the last reset transition. The existence of the first part is testable shown in (A.3) using Theorem 3.3 (b.4). Notice that $M_\omega$ does not reset on the second part, denoted by $\tau$. We further assume that on $\tau$, $M_\omega$ writes and queries infinitely many times on the query tape. Otherwise, it is easy to show that the existence of $\tau$ is testable in (A.3). Without loss of generality, we let $s$ be $s_0$. With these assumptions, $\tau$ is essentially an accepting $\omega$-run of $M_\omega$ (with oracle $O$) on which the query tape grows to infinity and an accepting state, say $s_f$, repeats infinitely often. As the result of $\tau$, we use $\alpha$ to denote the $\omega$-word that occupies the query tape. Let $A$ be a DPDA($n$) that accepts $O$. Since $A$ is deterministic, we may run $A$ along with the infinitely many write transitions performed during $\tau$: a query can be faithfully answered by looking at whether $A$ is at its accepting state. One can also observe the stack behavior during this infinite run of $A$ and pick infinitely many points on the run where the stack stays lowest (i.e., the stack height beyond the point is not lower). In particular, we have the following Property:

There must be two points $p_1$ and $p_2$ on $\tau$ such that all of the following conditions are satisfied:

- At the two points, the states of $M$ are the same;
- From point $p_1$ to point $p_2$ on $\tau$, $M$ has passed $s_f$ at least once, has read at least one input symbol, has queried the oracle for at least once, and has written at least one symbol on the query tape;
• The state (resp. top symbol of the stack) of A at point \( p_1 \) is the same as the state (resp. top symbol of the stack) of A at point \( p_2 \); (recalling that A runs along \( M \))

• From point \( p_1 \) to point \( p_2 \), A does not pop the stack content underneath the top symbol at point \( p_1 \).

In fact, one can also show that the Property implies the existence of \( \tau \) since the segment from \( p_1 \) to \( p_2 \) forms a loop. Therefore, testing the existence of \( \tau \) is equivalent to testing the Property. In the Property, the query tape content up to point \( p_2 \) can be accepted by a PDA with \( O(|M| \cdot n) \) states by composing A with \( M \) properly. Using the technique presenting in the proof of Theorem 3.3 (b.1), one can show that the Property as well as the existence of \( \tau \) is testable shown in (A.3).

The result follows by combining Case 1 and Case 2.

\[ \square \]

A.1.7 Proof of Theorem 3.8

Proof.

(1) Since \( M_\omega \) is \( k \)-query, after certain point on an \( \omega \)-run, \( M \) does not perform queries anymore. The result follows easily from Theorem 3.4 (b.1).

(2) The result follows from a similar argument made in the proof of Theorem 3.4 (b.2) and then from Theorem 3.6.

(3) Similar to the proof of Theorem 3.7 (2), except that we use Theorem 3.4 (b.3) instead of Theorem 3.3 (b.3).

\[ \square \]

A.2 Proofs Omitted From Chapter 7

A.2.1 Proof of Theorem 7.8

Proof. Consider an instance of the loose \( k \)-accumulated weight problem given above. Firstly we introduce a definition. For any word \( \alpha \), we use \( X_j(\alpha) \), \( 1 \leq j \leq l \), to denote
all the possible total weights of symbol $a_j$'s in $\alpha$. That is, $X_j(\alpha)$ defines the following set (similar to (7.27)):

$$\{x_j : q_j \cdot \#_{a_j}(\alpha) \leq x_j \leq w_j \cdot \#_{a_j}(\alpha)\}. \quad (A.4)$$

Clearly, for any $\hat{W}_i, 1 \leq i \leq k, \hat{W}_i \in W_i(\alpha)$ iff for each $a_j \in \Gamma_i$, there exists some $x_j \in X_j(\alpha)$ such that $\hat{W}_i = \sum_{a_j \in \Gamma_i} x_j$. Replacing each $\hat{W}_i$ in the Presburger formula $P(w_1, \ldots, w_l, \hat{W}_1, \ldots, \hat{W}_k)$ in (7.28) with the summation, another Presburger formula $Q(w_1, \ldots, w_l, x_1, \ldots, x_l)$ can be obtained. Then it is easy to see that the Question-part involving (7.28) is equivalent to the following: Is there a word $\alpha$ in $L$ such that, for some $w_1, \ldots, w_l \in N$, and for some $x_1, \ldots, x_l$,

$$x_1 \in X_1(\alpha) \land \cdots \land x_l \in X_l(\alpha) \land Q(w_1, \ldots, w_l, x_1, \ldots, x_l) \quad (A.5)$$

holds? Next, a semi-linear transform is applied upon the Presburger formula $Q(w_1, \ldots, w_l, x_1, \ldots, x_l)$. The result is to represent each of $w_1, \ldots, w_l, x_1, \ldots, x_l$ by a nonnegative linear polynomial over some nonnegative integer variables $s_1, \ldots, s_m$. Additionally, since $L$ is a semi-linear language, each $\#_{a_i}(\alpha)$ in (A.4) can also be represented (similar to the proof of Theorem 7.4) by a nonnegative linear polynomial over some nonnegative integer variables $t_1, \ldots, t_n$.

Finally, after the substitutions and reorganizing the result, (A.5) can be shown to be equivalent to a finite disjunction of quadratic Diophantine inequality systems, each of which is in the following form:

$$\begin{cases}
q_1 \cdot A_1(t_1, \ldots, t_n) \leq B_1(s_1, \ldots, s_m) \leq C_1(s_1, \ldots, s_m) \cdot A_1(t_1, \ldots, t_n) \\
\vdots \\
q_l \cdot A_l(t_1, \ldots, t_n) \leq B_l(s_1, \ldots, s_m) \leq C_l(s_1, \ldots, s_m) \cdot A_l(t_1, \ldots, t_n)
\end{cases} \quad (A.6)$$

where the $q$'s are the given constants in $N$, the $A_i$'s (for $\#_{a_i}(\alpha)$) are nonnegative linear polynomials over $t_1, \ldots, t_n$, the $B_i$'s (for $x_i$'s) and the $C_i$'s (for $w_i$'s) are nonnegative linear polynomials over $s_1, \ldots, s_m$. To show the theorem, it suffices to show that it is decidable whether (A.6) has a solution in $s_1, \ldots, s_m, t_1, \ldots, t_n$. We use Case $(m, n, l)$
to denote (A.6). The procedure to solve (A.6) is a complex induction process on the
numbers $m, n$ and $l$ in which Case $(m, n, l)$ is reduced to one of the followings:

- Case $(m', n', l - 1)$ where $m'$ and $n'$ could be larger than the original $m$ and $n$;
- Case $(m', n, l)$ where $m'$ is smaller than $m$;
- Case $(m, n', l)$ where $n'$ is smaller than $n$; or,
- a special case where it is directly solvable.

The induction base is: either one of $m$ and $n$ is 0, or $l = 1$. The former case
is solvable since Case $(m, n, l)$ will be Presburger. The latter case is also solvable.
That’s because, by introducing new variables to make the inequalities into equations,
(A.6) with $l = 1$ will be reduced to a system in the form of (7.20) with $k = 2$.

Now, we start with Case $(m, n, l)$ shown in (A.6) where $m > 0, n > 0, l > 1$. We
firstly assume that each $A_i$ in (A.6) is not a constant polynomial. Otherwise, say
$A_1$ is a constant polynomial $a \in \mathbb{N}$, then the first inequality in (A.6) will be the
following Presburger formula: $q_1 a \leq B_1(s_1, ..., s_m) \leq C_1(s_1, ..., s_m) \cdot a$. By applying
a semi-linear transform upon this formula, (A.6) will be reduced to a similar system
but with a smaller number $l$. This step of induction corresponds to Case $(m, n, l) \rightarrow
Case (m', n', l - 1)$ in the induction path.

We then assume that each $B_i$ in (A.6) is not a constant polynomial. Otherwise,
say $B_1$ is a constant polynomial $b \in \mathbb{N}$, then the first inequality in (A.6) will be:
$q_1 A_1(t_1, \cdots, t_n) \leq b \leq C_1(s_1, ..., s_m) \cdot A_1(t_1, \cdots, t_n)$. It is not hard to show that this
formula is equivalent to a finite disjunction of Presburger formulas in the follow-
ing form: $C_1(s_1, ..., s_m) \sim c \land A_1(t_1, \cdots, t_n) \sim a$, where $\sim \in \{=, >\}$ and $c, a \in \mathbb{N}$.
For each formula in the disjunction, a semi-linear transform can be applied upon
$C_1(s_1, ..., s_m) \sim c$, and another semi-linear transform upon $A_1(t_1, \cdots, t_n) \sim a$. In this
way, solving (A.6) will be reduced to solving finitely many systems, each of which is
still in the form of (A.6) but with a smaller number $l$. This step of induction also
corresponds to Case $(m, n, l) \rightarrow Case (m', n', l - 1)$ in the induction path.

At the last, we group all the $C_i$’s that are constant linear polynomials $c_i \in \mathbb{N}$
respectively. For notational convenience, we assume that they are $C_1, \cdots, C_k$, for
some \(0 \leq k \leq l\) (\(k = 0\) means that every \(C_i\) is not a constant). Without loss of generality, we assume \(k < l\) (since when \(k = l\), (A.6) is Presburger, and hence solvable).

Now let’s look at a system defined as follows:

\[
\begin{align*}
q_1 \cdot A_1(t_1, \ldots, t_n) &\leq B_1(s_1, \ldots, s_m) \leq c_1 \cdot A_1(t_1, \ldots, t_n) \\
&
\vdots \\
q_k \cdot A_k(t_1, \ldots, t_n) &\leq B_k(s_1, \ldots, s_m) \leq c_k \cdot A_k(t_1, \ldots, t_n) \\
q_{k+1} \cdot A_{k+1}(t_1, \ldots, t_n) &\leq B_{k+1}(s_1, \ldots, s_m) \\
&
\vdots \\
q_l \cdot A_l(t_1, \ldots, t_n) &\leq B_l(s_1, \ldots, s_m).
\end{align*}
\]  

(A.7) is the result of chopping off the inequalities on quadratic terms, \(C_{k+1}(s_1, \ldots, s_m) \cdot A_{k+1}(t_1, \ldots, t_n), \ldots, C_l(s_1, \ldots, s_m) \cdot A_l(t_1, \ldots, t_n)\), from (A.6). Clearly, every solution of (A.6) is also a solution of (A.7); but the inverse may not be true. (A.7) is Presburger, which defines a semi-linear set of tuples \((s_1, \ldots, s_m, t_1, \ldots, t_n)\). If the set is empty, then (A.6) has no solutions (in the induction path, this means Case \((m, n, l)\) is directly solvable). So assume that the semi-linear set is not empty. First, we assume that the set is a linear set

\[
S = \{ v \in \mathbb{N}^{m+n} : v = v_0 + v_1z_1 + \cdots + v_rz_r, z_1, \ldots, z_r \in \mathbb{N} \},
\]  

for some vectors \(v_0, v_1, \ldots, v_r \in \mathbb{N}^{m+n}\). Vector \(v \in \mathbb{N}^{m+n}\) denotes a tuple of \((s_1, \ldots, s_m, t_1, \ldots, t_n)\); we shall use notations like \(v(s_1)\) to denote the component value for \(s_1\) in \(v\).

If \(S\) is a finite set (i.e., \(r = 0\) in (A.8)), then it suffice to check whether \(v_0\), the only element in \(S\), constitutes a solution to (A.6). Namely, when \(S\) is finite, Case \((m, n, l)\) is directly solvable in the induction path.

In the following, we consider two situations for \(S\) when it is infinite. The first situation is that, there exists some \(s_i\) (resp. \(t_j\)) such that \(v(s_i) = v_0(s_i)\) (resp. \(v(t_j) = v_0(t_j)\)) for any \(v \in S\). This essentially means that, any \((s_1, \ldots, s_m, t_1, \ldots, t_n)\) is a solution to (A.7) only if \(s_i\) (resp. \(t_j\)) is a constant \(v_0(s_i)\) (resp. \(v_0(t_j)\)). Then, by
plugging-in the constant for $s_i$ (resp. $t_j$), (A.6) can be reduced to a similar one with a smaller $m$ (resp. $n$). In the induction path, this corresponds to a reduction from Case $(m, n, l) \rightarrow$ Case $(m', n, l)$ (resp. Case $(m, n', l)$) with $m' < m$ (resp. $n' < n$).

The other situation is that, for each $s_i$ (resp. $t_j$), there is an $e^i \in S$ (resp. $\theta^j \in S$) such that $e^i(s_i) > v_0(s_i)$ (resp. $\theta^j(t_j) > v_0(t_j)$). Take $\mu \in \mathbb{N}^{m+n}$ such that $\mu = \sum_i (e^i - v_0) + \sum_j (\theta^j - v_0)$. Clearly, $\mu$ satisfies all of the followings: for each $s_i$, $\mu(s_i) > 0$; for each $t_j$, $\mu(t_j) > 0$; and for each $z \in \mathbb{N}$, $v_0 + \mu \cdot z \in S$. Now we claim that under this situation, (A.6) has a solution. Clearly, $v_0 + \mu \cdot z$ constitutes a solution to (A.7) for each $z \in \mathbb{N}$. Next we only need to argue that for some $z$, $v_0 + \mu \cdot z$ also satisfies the quadratic inequalities that were “truncated off” from (A.6) when formulating (A.7). Recall that the truncated part is as follows:

\[
\begin{align*}
B_{k+1}(s_1, \ldots, s_m) &\leq C_{k+1}(s_1, \ldots, s_m) \cdot A_{k+1}(t_1, \ldots, t_n) \\
&\vdots \\
B_l(s_1, \ldots, s_m) &\leq C_l(s_1, \ldots, s_m) \cdot A_l(t_1, \ldots, t_n)
\end{align*}
\]  

(A.9)

It is easy to verify (left to the reader) that $v_0 + \mu \cdot z$ is a solution to (A.9) when $z$ is sufficiently large (notice that each of the $B$’s, the $A$’s, and the $C$’s is not a constant polynomial as we assumed). In the induction path, this induction step reduces Case $(m, n, l)$ to a directly solvable case.

When (A.7) defines a semi-linear set that is a union of more than one linear set $S$, we work on the $S$’s one by one.
Bibliography


