Quantitative Specifications for Verification and Synthesis

PhD Thesis

Author:
Arjun Radhakrishna

Supervisor:
Thomas A. Henzinger, IST Austria, Klosterneuburg, Austria

Thesis Committee:
Roderick Bloem, Technische Universität, Graz, Austria.
Pavol Černý, University of Colorado, Boulder, USA.
Krishnendu Chatterjee, IST Austria, Klosterneuburg, Austria.

Program Chair:
Herbert Edelsbrunner, IST Austria, Klosterneuburg, Austria

A Thesis presented to the faculty of the Graduate School of the Institute of Science and Technology Austria, Klosterneuburg, Austria, in partial fulfillment of the requirements for the degree Doctor of Philosophy (PhD).

© Arjun Radhakrishna, July, 2014.
All Rights Reserved

I hereby declare that this dissertation is my own work along with stated collaborators, and it does not contain other peoples' work without this being so stated; and that the bibliography contains all the literature that I used in writing the dissertation.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee, and that this thesis has not been submitted for a higher degree to any other university or institution.

Author:

Arjun Radhakrishna

Supervisor:

Thomas A. Henzinger

Thesis Committee:

Roderick Bloem

Pavol Černý

Krishnendu Chatterjee

Program Chair:

Herbert Edelsbrunner

To Archu and Archani...

Abstract

Standard specifications used for formal verification and synthesis of systems partition the set of all systems into "good" and "bad" systems. However, a more nuanced view is often required as not all acceptable systems are equally good, and not all unacceptable systems are equally bad. The aim of this dissertation is to explore the possibility of capturing these nuances through quantitative specifications where instead of a boolean yes/no classification, systems are rated using a quantitative metric. We extend classical specification, verification, and synthesis techniques to these quantitative specifications.

In the first part, we develop a framework of quantitative specifications for reactive systems based on distances between systems. These distances, called simulation distances, are a quantitative extension of the classical simulation relation between systems, and can be used to measure various aspects of the relation of a system to a specification such as correctness, coverage, and robustness. We then apply simulation distances to the problem of incompatible specifications — given two or more specifications having no common implementation, find the implementation that comes "closest" to satisfying each of the specifications. We also prove a number of properties of simulation distances, such as compositionality and soundness of abstractions, that aid in analysis of large systems. Further, we present a number of case studies that illustrate how the simulation distances framework can be used for various steps in the system development work-flow, such as test-suite generation and robustness analysis.

In the second part, we adapt a number of classical techniques for formal verification and synthesis of systems to handle quantitative properties. The techniques we focus on are counterexample-guided inductive synthesis, abstraction refinement, and infinite-state model checking. First, we present a counterexample-guided synthesis algorithm for synthesizing concurrent programs which are not only correct, but also have good performance. Second, we develop an abstraction and abstraction refinement framework for quantitative properties and apply it to estimation of worst-case execution time of programs. Third, we introduce battery transition systems, a modelling framework for systems that interact with an energy sources, and develop novel model checking algorithms for such systems.

Acknowledgments

I would like to express my gratitude to all the people without whom this thesis would not have been possible.

Firstly, my advisor Prof. Thomas A. Henzinger who gave me the support and guidance to work on all the diverse projects that interested me. The depth and breadth of his knowledge on a wide range of topics never failed to impress me. Working with him, I had the good fortune of observing at close quarters what it means to be a good researcher. I can only hope that some of his acuity and work ethic has rubbed off on me.

Another person whom I dare not fail to mention is Prof. Pavol Černý. It was extremely fun and satisfying working with him. While I learned from Tom what it means to be a good researcher, it was from Pavol I learned the how of it – the steps and routines needed to try and become one. His reserved optimism and sharp cynicism about our own work pointed me towards the right way of being self-critical as a scientist.

In addition, I would also like to thank the members of my thesis committee, Tom and Pavol mentioned above, and Prof. Roderick Bloem and Prof. Krishnendu Chatterjee for agreeing to review the thesis and provide suggestions.

I would also like to thank the members of the Chatterjee group and Henzinger group at IST for creating a lively and fun work environment. With Damien Zufferey, whose quips kept the office we were sharing lively; Cezara Dragoi, dropping by whenever things got too dull; Ali Sezgin, whose sharp sarcasm never missed anything; Udi Boker, who was great company for both discussion about batteries and eight-ball pool; Jan Otop, whose obscure factoids always made for good conversation material; Tatjana Petrov, who was my first office mate all the way back in 2007; Dejan Nickovic, with his casual and fun approach to both work and play; and all the rest – Martin Chmelik, Przemysław Daca, Rasmus Ibsen-Jensen, Andreas Pavlogiannis, Johannes Reiter, Sasha Rubin, Roopsha Samanta, Vasu Singh, Anmol Tomar, Throsten Tarrach, and Thomas Wies, I could have scarcely asked for a better place to work. And with Elisabeth Hacker efficiently mothering me along through the sometimes convoluted administrative and paper work, I must say that my stay at IST was a most pleasant one.

Another person from IST whom I want thank deeply is Hande Acar with whom I shared a large number of afternoon teas, weekend brunches, and quiet conversations over the years. Last, but not the least, I would like to thank the people dearest to me, my parents, my sister, and Archana Ghangrekar for all the love and support.

Record of Publications

The work appearing in this dissertation was supported in part by the Austrian Science Fund NFN RiSE (Rigorous Systems Engineering) and by the ERC Advanced Grant QUAREM (Quantitative Reactive Modeling).

Chapter 3[Pg. 30] is joint work with Pavol Černý and Thomas A. Henzinger, and contains material that first appeared in [36], [35], and [37]. Chapter 4[Pg. 67] contains material that first appeared in [34] and is joint work with Pavol Černý, Sivakanth Gopi, Thomas A. Henzinger, and Nishant Totla. However, the technical parts and the proofs in these chapters have been significantly rewritten and modified.

Chapter 6[Pg. 90] was published as [32] and is joint work with Pavol Černý, Krishnendu Chatterjee, Thomas A. Henzinger and Rohit Singh. Chapter 7[Pg. 110] was published as [37] and is joint work with Pavol Černý and Thomas A. Henzinger. Chapter 8[Pg. 144] is an extension to Chapter 7[Pg. 110] based on an as yet unpublished manuscript along with Laura Kovacs and Jakob Zwirchmayr. Chapter 9[Pg. 168] is joint work with Udi Boker and Thomas A. Henzinger and has appeared as [26].

Contents

1 Introduction **6**
- 1.1 Motivation . . . 6
- 1.2 Paradigms of Quantitative Analysis . . . 8
- 1.3 Outline . . . 14

2 Preliminaries: Systems and Specifications **16**
- 2.1 Modelling Discrete Computational Systems . . . 16
- 2.2 Automata over Infinite Words . . . 18
- 2.3 $2\frac{1}{2}$-Player Games . . . 19

I Quantities as Preference **26**

3 Simulation Distances **30**
- 3.1 Motivation . . . 30
- 3.2 Simulation Relations, Simulation Games, and Quantitative Simulation Games . . . 33
- 3.3 Simulation Distances . . . 41
- 3.4 Properties of Simulation Distances . . . 51
- 3.5 Applications of Simulation Distances . . . 59

4 Synthesis from Incompatible Specifications **67**
- 4.1 Motivation . . . 67
- 4.2 The Incompatible Specifications Problem . . . 71
- 4.3 Case studies . . . 75

5 Discussion **78**
- 5.1 Extensions and Future Work . . . 78
- 5.2 Related Work . . . 82
- 5.3 Conclusion . . . 85

II Quantities as Measurement **86**

6 Quantitative Synthesis for Concurrency **90**
- 6.1 Motivation . . . 90
- 6.2 The Quantitative Synthesis Problem . . . 93
- 6.3 Quantitative Games on Graphs . . . 96
- 6.4 Practical Solutions for Partial-Program Resolution . . . 101

6.5 Experiments 104
6.6 Summary 108

7 Quantitative Abstraction Refinement **110**
7.1 Motivation 110
7.2 Quantitative properties 114
7.3 State-based quantitative abstractions 115
7.4 Segment-based quantitative abstractions 118
7.5 Generalizations of *PathBound* abstractions 124
7.6 Quantitative refinements 135
7.7 Case study: Cache hit-rate analysis 139
7.8 Summary 143

8 Precision Refinement for Worst-Case Execution Time Analysis **144**
8.1 Motivation 144
8.2 Illustrative Examples 147
8.3 Problem Statement 151
8.4 Max-Weight Length-Constrained Paths 153
8.5 Interpolation for Segment-Based Abstraction Refinement 157
8.6 Parametric WCET Computation 161
8.7 Experimental Evaluation 163

9 Battery Transition Systems **168**
9.1 Motivation 168
9.2 Battery Models 171
9.3 Battery Transition Systems 174
9.4 The Bounded-Energy Reachability Tree 179
9.5 Model Checking 184
9.6 Case Study 191
9.7 Summary 197

10 Discussion **198**
10.1 Future Work 198
10.2 Related Work 199
10.3 Summary 201

11 Conclusion **202**

Chapter 1

Introduction

The main goal of this dissertation is to explore the theory and application of quantitative specifications in formal techniques for system development. To this end, we develop a quantitative framework for specification of reactive systems, as well as quantitative analogues of various standard boolean techniques for verification and synthesis for systems.

1.1 Motivation

Formal techniques in system development have had significant success over the past several decades. In most formal techniques, the central object is the specification of the system. Formal verification, validation, and automated synthesis of systems would be impossible without quality specifications to define the correct behaviours. Further, specifications often play other roles in system development such as aiding in generation of test cases, and in many cases, acting as documentation. Often, significant advances in formal verification and synthesis have been a result of new specification formalisms. For example, the advent of efficient and practical model checking techniques started with the use of temporal logics for specifications [132, 59], the verification of real time systems and hybrid systems were enabled by the introduction of timed automata, real-time logics, and hybrid automata for specifying them [6, 9, 10, 7], and the proliferation of synthesis techniques for software started with the use of partial-programs (sketches) as specifications [153, 154, 155]. Hence, it is of paramount importance that in addition to the study of novel algorithms and methodologies for existing specification formalisms, the specification formalisms themselves be examined periodically to learn from their short-comings.

We briefly describe the current state of the art for specifying discrete event systems. In the domain of hardware systems, most specification frameworks can be roughly classified into two categories — automata-based and logic-based. Specifications in this domain are usually full-functional specifications, i.e., the specification expresses all the requirements on the system. In logic-based frameworks, the good temporal behaviours of a system are expressed using either a formula in temporal logic, or a program in a high-level language that compiles down to temporal logic. These specifications are usually easier to handle — for example, adding a requirement to the specification can be done by just adding

a conjunct to the formula. However, most automated techniques that handles logic-based specifications must first convert them into an automata-based specification. On the other hand, automata-based specifications (where the specification is expressed either a (possibly symbolic) automata [58, 17] or a program that compiles down to an automata) are much easier to use in algorithmic techniques, but are harder to manipulate and modify. In the software domain, there are very few full-fledged specification frameworks to specify the complete functionality of programs (exceptions include interactive theorem-prover and code-generation frameworks such as Coq [19] and Isabelle/HOL [129]). Most automated formal techniques either handle standard "specification-free" requirements (such as memory safety or absence of integer overflow) or handle partial specifications written into the program by the developer (such as assertions or pre- and post-conditions). These specifications do not specify the full functionality of the system, but instead only some necessary conditions for correctness. A popular recent paradigm for specification of programs is partial-programs. Here, most of the program is written by the developer. However, certain non-deterministic choices for expressions (called holes) are present in the partial program. The rest of the specification is specified either using assertions of pre- and post-conditions. A program satisfies the partial-program if it instantiates the holes with expressions such that the assertions or the pre- and post-conditions are satisfied.

Most standard specification frameworks come with a host of automated and semi-automated algorithmic techniques that can be used for verification, synthesis and testing, making them extremely useful for system development. However, while these classical specification frameworks are extremely successful, there are several common problems that arise when using these specifications in system development. Full functional specifications, like those of hardware designs, have a tendency of becoming large and cumbersome to write or maintain. As a result, a bug found during verification is as often a bug in the specification as in the system. Further, requirements change midway during the process of system development, and often auxiliary specifications derived from these requirements need to be changed. Systems synthesized from specifications are rarely as good as hand crafted ones due to missing soft requirements (such as the requirement of good performance or acceptable quality of service). On the other hand, even if the developer is aware of these soft requirements, the specification frameworks often cannot express them and the accompanying automated algorithmic techniques cannot handle them.

Our main hypothesis is that quantitative specifications can help alleviate several of these problems. While classical (boolean) specifications classify systems into good and bad ones, quantitative specifications instead rate the systems using a quantitative metric to assign values to systems. These metrics may either correspond to a physical property of the system (such as execution time or energy consumption) or to the relation of the system to another. In this dissertation, we uniformly adopt the convention that a system which is assigned a lower value by the quantitative specification is preferred over a system that is assigned a higher value. There are several reasons that motivate our hypothesis about quantitative specifications.

- Firstly, a large fraction of the soft specifications that are inexpressible in the classical specification frameworks are "inherently quantitative", i.e.,

they often correspond to a physical, quantitative property of the system such as execution time, response time, or power consumption.

- Secondly, even when the corresponding requirement is expressible in a classical specification framework, quantitative specifications allow the developer to express exactly the "what" of the requirement than the "how". For example, a requirement that a system respond to a request within some fixed time could be translated into a classical boolean specification by specifying restrictions on the behaviour of various components within the system. However, such a specification expresses "how" to achieve the requirement—if the design of the various components of the system changed, one would need to rewrite the specification. Instead a quantitative framework would allow the specification to directly express the "what" of the requirement, i.e., that the response time is to be minimized or bounded.

- Thirdly, quantitative specifications allow the developer to naturally express trade-offs. A large number of design choices that arise during development are intuitively trade-offs between various resources or system properties (for example, CPU usage versus network usage). Quantitative specifications can be used to specify systems as points on the Pareto curves of these trade-offs — just by varying the parameters, one can change the system being specified. In a classical specification framework, there is no way of switching between different points in the trade-off curve, other than rewriting the specification from scratch.

While we believe that quantitative specifications can solve various problems with the classical specifications, there are several hurdles to their adoption. Classical specification frameworks have been studied for a long time and a host of algorithmic techniques and development methodologies have been constructed around them. These include abstraction based techniques to handle large system, automatic synthesis of tricky parts of the system, test-case generation in the cases where full verification is not possible, etc. On the other hand, quantitative specifications are novel enough that not many such techniques exist. We strive to rectify this situation in this work, and present algorithmic techniques to enable various development methodologies.

1.2 Paradigms of Quantitative Analysis

In this work, we distinguish between two different paradigms of quantitative specifications. These are depicted in Figure 1.1[Pg. 9].

- *Behavioural Metrics.* In the first paradigm, the quantities in the analysis are not an inherent property of the system being analyzed but come from the relation (or the degree of fit) of the system to an ideal Boolean specification. This is analogous to the class of specification-based requirements in the classical setting – for example, the requirement that all the behaviours of a reactive system strictly conform to a temporal specification given by an automaton. Similarly, in the quantitative setting, we can define behavioural metrics that measure how closely the behaviour of a

Figure 1.1: Paradigms of Quantitative Analysis

reactive system conforms to a temporal specification given by an automaton.

- *Quantitative Properties.* In the second paradigm, the quantities in the analysis arise inherently as a measurement of some property of the system itself, and not a distance to any specification. These properties are analogous to *specification-free* properties in the boolean world – for example, the requirement that a program never dereference a null-pointer. Two common quantitative properties that fall into this paradigm are execution time and energy consumption.

As in the boolean world, the distinction between behavioural metrics (specification-based) and quantitative properties (specification-free) paradigms is neither hard and fast, nor well-defined. However, it is a convenient high-level categorization to separate the techniques and algorithms required for their analysis. Part I[Pg. 27] and Part II[Pg. 87] of this dissertation mainly deal with the behavioural metric paradigm and quantitative property paradigm respectively. In Section 1.2.1[Pg. 9] and Section 1.2.2[Pg. 12], we describe the major contributions of the two parts.

1.2.1 Quantities as Preference: Specifying Reactive Systems

Formal methods have seen significant success in the field of hardware verification and synthesis where most systems can be modelled as reactive systems. In this domain, the specifications are usually given as either as formulae in temporal logics or as automata over the set of input and output actions. These specifications serve as the starting point in a large number of steps in the process of system design such as verification, test generation, and code synthesis.

However, there are several problems with the use of formal temporal specifications for reactive system design, verification, and synthesis. These problems usually arise from the quality and source of the specifications. Real specifications are large and composed of multiple requirements arising from different sources; and they rarely capture the full intent of the designer. The multiple requirements in a specification are often incompatible, i.e., they contradict each other in corner cases; the designer writes more detailed low-level specifications resolving all the high level incompatibilities. Further, it is very common that additional requirements are added or existing requirements are modified as the

design process proceeds. Due to these changes, the detailed specifications written by the designer may possibly need to be rewritten. While synthesizing from such specifications, these problems lead to a significant drop in the quality of synthesized systems. As specifications rarely capture the full intent of the designer, the quality of the synthesized systems is often much worse than that of the corresponding hand-crafted systems. We explain some of these difficulties with two illustrative examples.

Illustrative Examples. Consider a reactive request-grant system with the following specification: every request r must be eventually granted with a grant g. Let us call this specification R_0. Now, there are an infinite number of implementation systems all of which satisfy the above requirement. One way of preferring some implementations over others is by adding the additional requirement of minimizing "spurious grants" (say requirement R_1). Translating this into a standard classical specification framework entails adding the following two specifications:

- S_1: There are no grants before the first request; and
- S_2: After every grant, there are no more further grants until another request is seen.

These additional specifications exactly express the no spurious grants requirement. However, one caveat is that they express the "how" of the requirement, rather than the "what" of it.

Another more important caveat is that S_1 and S_2 are not very robust towards additional requirements — if other requirements are added, S_1 and S_2 will need to be changed. For example, let us add an additional requirement that there must be at least one grant every 100 steps, to keep the system "live" (call this requirement R_2). Now, specifications S_1 and S_2 need to be changed to account for this by rewriting them into S_1' and S_2' as follows:

- S_1': There are no grants before the first request or 100 time steps have passed; and
- S_2': After every grant, there are no more further grants until another request is seen or 100 time steps have passed.

While the rewriting was simple in this case, often, the addition specifications depend non-trivially on more than one requirement and the process becomes much harder.

We consider another example to illustrate the problem of incompatible specifications. Consider reactive request-grant systems with two kinds of requests (r_1 and r_2) and grants (g_1 and g_2). The requirements we place on these systems are:

- R_1: Every request r_1 is granted in the same step with grant g_1;
- R_2: Every request r_2 is granted in the same step with grant g_2; and
- R_3: Grants g_1 and g_2 cannot occur simultaneously.

These requirements are mutually incompatible, i.e., there is no common implementation. This is because when both request r_1 and r_2 arrive at the same step, the system cannot output both g_1 and g_2. Such incompatibilities are usually resolved by the designed by writing more detailed specifications. Here, one resolution S_1 can be that when both r_1 and r_2 arrive at the same time, the system alternates between granting with g_1 and g_2. Another resolution S_2 is that if both r_1 and r_2 arrive in the same step infinitely often, then the system

grants with both g_1 infinitely often and with g_2 infinitely often. However, if the intention of the designer is that both r_1 and r_2 are treated with the same priority, then the resolution S_2 is too weak and the resolution S_1 is too strong.

The resolution S_2 allows systems which grant conflicting requests with g_1 more often than with g_2, while resolution S_1 does not allow systems which grant alternatingly with g_1 for a burst of n steps and then with g_2 for a burst of n steps.

Further, as in the previous example, if either of the original requirements R_1 or R_2 were to change, the chosen resolution (S_1 or S_2) needs to be rewritten. For example, if R_1 were to be changed to each request r_1 is to be granted either in the same step or in the next step, the resolution S_1 needs to be changed significantly.

The Simulation Distances Framework. Our approach to solving the problems with classical specifications described above is to use quantitative specifications to better express the intent of the designer. We introduce the simulation distances framework for specifying reactive systems. Simulation distances are a quantitative extension of the classically used correctness condition, the simulation relation [127]. Simulation distances are defined using a quantitative extension of the simulation game. In a classical simulation game, two players (say Player 1 and Player 2) alternatively choose transitions from the implementation and specification respectively. If Player 2 can always match the implementation transition chosen by Player 1 with a specification transition having the same output, the specification simulates the implementation. In a quantitative simulation game, we allow Player 2 to pick mis-matching transitions from the specification, but such mismatches are penalized according to an *error model*. The simulation distance from the specification to the implementation with respect to the error model is the lowest penalty value achievable by Player 2 in the quantitative simulation game. We show that using variations of the simulation game, we can measure various properties (such as correctness, coverage, and robustness) of the implementation with respect to the specification.

We argue that specifying that the desired system is the one that minimizes the simulation distance to an ideal specification with respect to a given error model allows the designer to capture her intent better as many specifications are "inherently quantitative". For example, the requirement of minimizing spurious grants in the above illustrative example could be achieved by adding an ideal specification that does not grant at all, and an error model that penalized every grant. Minimizing the distance to this ideal specification would minimize spurious grants. Further, this specification and error model need not be changed when the additional requirement R_1 is added.

Quantitative specifications in general, and simulation distances in particular, also provide a different approach to the problem of incompatible specifications. We can resolve multiple incompatible requirements by stating that the desired system is one that minimizes the maximal simulation distance to each of these specifications. For example, in the illustrative example above with conflicting requests, the preferred systems are those that minimize the maximal simulation distance to requirement R_1 and R_2. Further, changing one of these requirements does not require changing any of the further resolutions.

We present an algorithm for synthesizing optimal implementations that min-

imize the maximal simulation distances to a number of incompatible specifications. The algorithm is based on finding optimal strategies in multi-dimensional quantitative games.

Comparisons to other quantitative specification techniques. While we have discussed the advantages of the quantitative simulation distances framework over Boolean specifications in many settings, the simulation distances framework also has several advantages compared to other quantitative specification techniques (based on weighted automata or weighted logics) [22, 77]. Most important of these being that the simulation distances framework avoids many of the computational complexity and undecidability issues that arise from the use of weighted automata or weighted logics. Formally, the weighted containment problem (quantitative equivalent of Boolean language inclusion), the problems of checking if there exists a behaviour whose value is greater or lesser than a given threshold value (quantitative equivalents of Boolean emptiness and universality problems), and other important language theoretic problems are undecidable for weighted automata (see, for example, [73]). On the other hand, for simulation distances, these problems are decidable, and further, decidable in polynomial time for the common case of bounded weights. Intuitively, this difference is due to the fact that simulation distances are inherently "branching time" specifications, while weighted automata are inherently "linear time" specifications (see [165, 166] for definition of linear time and branching time specifications). As in the Boolean case, problems arising from linear time specifications (for example, computing language inclusion) are computationally more expensive than problems arising from branching time specifications such as computing the simulation relation. Hence, the simulation distances framework lets us use quantitative specifications without paying the penalty of computational complexity or undecidability as with many other quantitative specification techniques. Further, the simulation distances framework also enables many standard system design paradigms such as hierarchical and compositional design. For a discussion of these and further comparison to other quantitative specification techniques, see Part I.

1.2.2 Quantities as Measurement: Analyzing Quantitative Properties

In the second part of the dissertation, we mainly deal with extending classical verification and synthesis techniques to systems with quantitative properties. The three classical techniques we focus on are counterexample-guided inductive synthesis, abstraction and abstraction refinement, and model checking of infinite-state systems.

Performance-aware Synthesis. Partial-program synthesis is a development methodology where a developer writes only a part of the program, and specifies the rest of her intent declaratively. The incomplete program (often called a partial-program) may have holes corresponding to various parameters. In partial-program synthesis for concurrency, these holes usually correspond to synchronization choices. In most previous work (see, for example, [153, 53]) either the performance penalty of the synthesized synchronization is ignored,

or heuristics (such as minimizing the size of synthesized atomic sections, or the granularity of locks) are used. However, in many cases, these heuristics give the wrong answer or do not apply at all as in the case of optimistic concurrency. For example, it is easy to write programs where fine-grained locks performs much worse than coarse-grained locks (due to the cost of the locking operations themselves), and the heuristic of minimizing critical sections does not apply at all to the case of optimistic concurrency as the size of the critical sections are fixed.

We introduce a counterexample-guided synthesis algorithm that does not use such heuristics, but instead synthesizes the optimal correct program for a given architecture. In addition to the standard inputs (the partial program and correctness condition), the performance characteristics of the architecture is provided to the algorithm in the form of a *performance model*, which is a weighted automaton over various synchronization related actions. The output is a specialization of the partial program which is not only correct, but is also optimal with respect to the performance model.

Quantitative Abstraction Refinement. Most formal techniques that are able to handle large systems and state-spaces are based on abstraction [65, 66] and automated abstraction refinement [60]. In such techniques, given a (concrete) system, a smaller abstract system is built such that the set of behaviours exhibited by the abstract system is a superset of the set of behaviours exhibited by the concrete system. Hence, if all the behaviours of the abstract system satisfy a given property, then all the behaviours of the corresponding concrete system satisfy the property. Automated abstraction refinement is a technique where given an abstract system and a spurious (not present in the concrete system) behaviour of the abstract system that does not satisfy the given property, automatically produces another abstract system which does not contain the behaviour.

Most abstraction and abstraction refinement techniques are state-based, i.e., the abstract objects being reasoned about are sets of concrete states. Most quantitative properties are path-based, i.e., they are a property of whole traces instead of single states. Path-based or trace-based abstraction has been used mainly for analysis of termination of programs (See, for example, [135, 62]). Hence, trace-based abstraction is most suited for quantitative properties. However, unlike in termination analysis, the techniques need to reason quantitatively about parts of traces. To this end, we introduce segment-based abstractions for quantitative properties. In a segment-based abstraction, the abstract objects are sets of segments (parts of execution traces) and along with each abstract segment, relevant quantitative properties (for example, the maximum and minimum length of a segment) are stored. These quantitative properties of abstract segments can then be used to compute an abstract system value.

Our quantitative abstraction techniques provide the quantitative version of the standard guarantee, i.e., that the value of the abstract system is always an over-approximation of the value of the concrete system. However, in addition, our abstraction refinement algorithm has the *anytime property*, by which we mean that the abstraction refinement algorithm can be stopped at any point of time during its execution, and it will return a valid over-approximation of the quantitative property. Further, the algorithm will produce tighter over-

approximations given more time to execute.

We apply this algorithm to the problem of estimating the worst-case execution time of programs. We show that our techniques provide a significant improvement over standard worst-case execution time analysis.

Model Checking of Battery Systems. Systems interacting with energy sources have been modelled classically using weighted transition systems and energy games. In all these modelling frameworks, the state of the energy source is represented using one number, i.e., the amount of energy remaining in the source. However, real batteries exhibit behaviour that is not feasible in any of these ideal models. A major class of these behaviours is dubbed the *recovery effect*. In the recovery effect, a battery which is apparently dead and out of energy recovers a small amount of energy after a certain period of time. Another non-ideal effect that is observed in real batteries is the *rate-limit effect*. Here, not all energy of the battery is immediately available for use, i.e., there is a limit on the rate at which energy can be used.

We introduce battery transition systems which are able to model such non-ideal behaviour. Battery transition systems are a discretization of the KiBAM model, a commonly used physical model of batteries [123]. Intuitively, in a battery transition system, the total charge in the battery is divided into two parts – known as tanks. The *available charge tank* contains energy that is immediately available for use, while the *bound charge tank* contains energy that is attached internally to the battery. As time passes, there is diffusion between the two tanks of the battery.

Battery transition systems do not fall into any previously known class of infinite state systems for which standard model checking problems are decidable. For example, battery transition systems do not fall into the class of well-structured transition systems. We present novel model checking algorithms for battery transitions systems. These algorithms are based on building forward reachability trees a la Karp-Miller [110], and using specific conditions to stop exploration before a certain depth is reached.

1.3 Outline

We first start with a preliminary introduction and definitions of the basic concepts and techniques used in this report in Chapter 2[Pg. 16]. Further, this report is divided into two parts – Part I[Pg. 27] dealing with behavioural metrics for specification of reactive systems and Part II[Pg. 87] dealing with analysis for quantitative properties of systems. Other than the preliminary definitions, each part is as self-contained as possible and contains an introduction motivating the work described in the part, and a conclusion discussing the related work.

Quantities as Preference: Part I[Pg. 27]. In Chapter 3[Pg. 30], we introduce simulation distances, a quantitative framework for specifying reactive systems. In Chapter 4[Pg. 67], we discuss how to extend the simulation distances framework to handle multiple, incompatible requirements.

Quantities as Measurement: Part II[Pg. 87]. In Chapter 6[Pg. 90], we extend the standard counterexample-guided inductive synthesis algorithm to account for the performance of the synthesized programs in the domain of concurrent programs. Chapter 7[Pg. 110] deals with the theory of abstraction and abstraction refinement for quantitative properties. In Chapter 8[Pg. 144], we apply the techniques developed in Chapter 7[Pg. 110] to the problem of estimating the worst-case execution time of programs. In Chapter 9[Pg. 168], we introduce a new model for systems that interact with an energy source, and provide model checking algorithms for such systems.

Chapter 2

Preliminaries: Systems and Specifications

This report deals mainly with the analysis and synthesis of discrete event computational systems. In this chapter, we introduce the formalisms we use to model and specify these kinds of systems.

2.1 Modelling Discrete Computational Systems

2.1.1 Labelled Transition Systems

Labelled transition systems have been used to model computational systems since they were introduced. Labelled transition systems are an elegant way of separating the operations that can be performed by the computational system (actions) from the computational state of the system.

Formally, a *labelled transition system* L is a tuple $\langle S, \Sigma, \Delta, s_\iota \rangle$ where:

- S is a set of states;
- Σ is a finite set actions;
- $\Delta \subseteq S \times \Sigma \times S$ is a set of transitions; and
- $s_\iota \in S$ is an initial state.

Further, we require that for every state $s \in S$, there exist $\sigma \in \Sigma$ and $s' \in S$ such that $(s, \sigma, s') \in \Delta$. Intuitively, S represent the state of a computational system, and Σ represent the class of operations that the computational system may perform. The transition set Δ contains the set of single computational steps the system may perform, i.e., $(s, \sigma, s') \in \Delta$ implies that the system may go from state s to state s' on performing an action σ. The set of all labelled transition systems sharing the same set of actions Σ is denoted by $\mathcal{S}(\Sigma)$.

A labelled transition system is *deterministic* if for each state $s \in S$ and action $\sigma \in \Sigma$, the next state is unique, i.e., $|\{s' \mid (s, \sigma, s') \in \Delta\}| = 1$ for each s and σ. If a labelled transition system is not deterministic, it is *non-deterministic*.

When a computational system is modelled using a labelled transition system, the behaviours of the corresponding system are traces of the labelled transition

system. Formally, a sequence $s_0\sigma_0 s_1\sigma_1\ldots$ is a *infinite trace* of the labelled transition system $L = \langle S, \Sigma, \Delta, s_\iota\rangle$ if and only if: (a) for all $i \in \mathbb{N}$, $(s_i, \sigma_i, s_{i+1}) \in \Delta$; and (b) $s_0 = s_\iota$. Similarly, a sequence $\pi = s_0\sigma_0 s_1\sigma_1\ldots s_n$ is a *finite trace* of L if: (a) for all $0 \le i < n$, $(s_i, \sigma_i, s_{i+1}) \in \Delta$; and (b) $s_0 = s_\iota$. We represent the set of all infinite traces of a labelled transition system L by $\mathit{Traces}(L)$, and the set of all finite traces by $\mathit{FiniteTraces}(L)$.

Alternating Transitions Systems We model computational systems that are controlled by two separate agents, each controlling the "input" and "output" actions of the system, using alternating transition system [11]. Formally, an *alternating transition system* is a tuple $\langle S, (S_{in}, S_{out}), \Sigma, (\Sigma_{in}, \Sigma_{out}), \Delta, s_\iota\rangle$ where $\langle S, \Sigma, \Delta, s_\iota\rangle$ is a labelled transition system and:

- S_{in} and S_{out} form a partition of S, i.e., $S_{in} \cup S_{out} = S$ and $S_{in} \cap S_{out} = \emptyset$ – S_{in} and S_{out} are called the *input states* and *output states* respectively;
- Σ_{in} and Σ_{out} form a partition of Σ, i.e., $\Sigma_{in} \cup \Sigma_{out} = \Sigma$ and $\Sigma_{in} \cap \Sigma_{out} = \emptyset$ – Σ_{in} and Σ_{out} are called the *input actions* and *output actions* respectively; and
- The transitions $\Delta \subseteq S_{in} \times \Sigma_{in} \times S \cup S_{out} \times \Sigma_{out} \times S$, i.e., the transitions from input states are on input actions and the transitions from output states are on output actions.

Intuitively, in an alternating transition system, there are two types of actions – the input and output actions. The actions Σ_{in} represent the inputs from the environment to the system, and the actions Σ_{out} represent the outputs from the system to the environment. Correspondingly, the input states represent states where the system waits for input from the environment, and the output states represent states where the systems provides output to the environment.

An alternating transition system the transitions from Player 1 states go only to Player 2 states and vice-versa are called *synchronous reactive systems*.

Fairness constraints A *Büchi (weak fairness) condition* for a labelled transition system having state space S is specified as a set of Büchi states $B \subseteq S$. Given a Büchi condition B and an infinite trace $\pi = s_0\sigma_0 s_1\sigma_1\ldots$ of a transition system, the trace π is *fair* if and only if $\forall n \ge 0 : (\exists i > n : \rho_i \in B)$.

A *Streett (strong fairness) condition* for a labelled transition system with state space S is a set of request-response pairs $F = \{\langle R_1, G_1\rangle, \langle R_2, G_2\rangle, \ldots, \langle R_d, G_d\rangle\}$ where each $R_i \subseteq S$ and each $G_i \subseteq S$. Given the above Streett condition, a trace $\pi = s_0\sigma_0 s_1\sigma_1\ldots$ is *fair* if and only if $\forall 1 \le k \le d : \big((|\{i \mid s_i \in R_k\}| = \infty) \Rightarrow (|\{i \mid s_i \in G_k\}| = \infty)\big)$.

Weighted Transition Systems A *weighted transition system* $W = \langle S, \Sigma, \Delta, s_\iota, v\rangle$ is a transition system $\langle S, \Sigma, \Delta, s_\iota\rangle$ along with a weight function $v : \Delta \to \mathbb{N}$ that maps transitions of the system to weights given by natural numbers. The weights of a transition system are used to represent some quantitative metric about the transition (for example, the energy cost associated with it).

Weighted probabilistic transition system. A *probabilistic transition system* (PTS) is a generalization of a transition system with a probabilistic transition function. Formally, let $\mathcal{D}(S)$ denote the set of probability distributions over S. A PTS consists of a tuple $\langle S, \Sigma, \Delta, s_\iota \rangle$ where S, Σ, s_ι are as for transition systems, and $\Delta : S \times \Sigma \to \mathcal{D}(S)$ is probabilistic, i.e., given a state and an action, it returns a probability distribution over successor states. A *weighted probabilistic transition system* (WPTS) consists of a PTS and a weight function $v : S \times \Sigma \times S \to \mathbb{Q} \cup \{\infty\}$ that assigns weights to transitions. An *trace* of a weighted probabilistic transition system is an infinite sequence of the form $(s_0\sigma_0 s_1\sigma_2 \ldots)$ where $s_i \in S$, $\sigma_i \in \Sigma$, and $\Delta(s_i, \sigma_i)(s_{i+1}) > 0$, for all $i \geq 0$.

2.2 Automata over Infinite Words

Automata are a standard formalism for modelling properties of discrete computational systems – they are used to abstract away from implementation details and instead model properties of the observable input/output behaviours, i.e., the actions of the system.

Formally, a *finite automaton* $\mathcal{A}$ is a tuple $\langle S, \Sigma, \Delta, s_\iota \rangle$ where each of S, Σ, Δ, and s_ι are as in a labelled transition system with the additional restriction that the set of states S is finite.

Remark 2.1. *In the context of finite automata, the set of actions Σ is called the* alphabet*, and $\mathcal{A}$ is known as an automaton* over the alphabet Σ*. Further, in this setting, finite and infinite traces of an automaton $\mathcal{A}$ are known as finite and infinite* runs*.*

Though the definition of finite automata and labelled transition systems is very similar, we differentiate between them due to the way each is used. Labelled transition systems are generally used to model a single computational system while automata are used to represent sets of behaviours of related systems sharing the same set of actions (or alphabet).

Languages and Accepting Conditions Given an alphabet (or equivalently a set of actions) Σ a *finite word* w over Σ is a finite sequence $\sigma_0\sigma_1 \ldots \sigma_n$ and an *infinite word* w over Σ is an infinite sequence $\sigma_0\sigma_1 \ldots$ where each $\sigma_i \in \Sigma$ for all $i \in \mathbb{N}$. The set of all finite and infinite words over Σ are represented by Σ^* and Σ^ω respectively.

An automata $\mathcal{A}$ is used to represent a set $Lang(\mathcal{A})$ of (finite or infinite) words over Σ – this set of words is called the *language of the automaton.* Each word in the language of the automaton is *accepted* by the automaton – i.e., the language of the automaton is the set of all words accepted by it. The language of the automaton is defined using *accepting conditions* over the set of runs of the automaton. We list some commonly used accepting conditions:

- *Büchi accepting condition.* The Büchi acceptance condition is specified by a subset of automaton states $B \subseteq S$ (known as Büchi states). A word $w = \sigma_0\sigma_1 \ldots \in \Sigma^\omega$ is accepted by $\mathcal{A}$ with a Büchi accepting condition given by B if and only if there exists a run of the automaton $s_0\sigma_0 s_1\sigma_0 \ldots$ such that $s_i \in B$ for infinitely many i, i.e., $\forall j : \exists i : s_i \in B$. An automaton $\mathcal{A}$ along with a Büchi accepting condition is called a *Büchi automaton.* For a Büchi automaton $\mathcal{A}$, the language $Lang(\mathcal{A}) \subseteq \Sigma^\omega$.

- *Finite accepting condition.* The finite acceptance condition is specified by a set of safe states $P \subseteq S$. A word $w = \sigma_0\sigma_1 \dots \sigma_{n-1}$ is accepted by an automaton with a finite accepting condition if there exists a run of the automaton $s_0\sigma_0 s_1\sigma_1 \dots \sigma_{n-1}s_n$ such that $\forall 0 \leq i \leq n : s_i \in F$. An automaton $\mathcal{A}$ along with a finite accepting condition is called a *finite automaton*. For a finite automaton $\mathcal{A}$, the language $Lang(\mathcal{A}) \subseteq \Sigma^*$.

- *Safety accepting condition.* The Safety acceptance condition is specified by a subset of automaton states $P \subseteq S$ (known as safe states). A word $w = \sigma_0\sigma_1 \dots \in \Sigma^\omega$ is accepted by $\mathcal{A}$ with a safety accepting condition given by P if and only if there exists a run of the automaton $s_0\sigma_0 s_1\sigma_0 \dots$ such that $s_i \in P$ for infinitely many i, i.e., $\forall j : \exists i : s_i \in P$. An automaton $\mathcal{A}$ along with a safety accepting condition is called a *safety automaton*.

Weighted Automata and Objective Functions A *weighted automaton* is a tuple $\langle S, \Sigma, \Delta, s_\iota, v\rangle$ where $\langle S, \Sigma, \Delta, s_\iota\rangle$ is a finite automaton, and $v : \Delta \to \mathbb{N}$ is function mapping transitions of the automaton to weights given by natural numbers.

Weight functions in weighted automata are used to assign costs or weights to transitions. They are used to model various quantitative properties of transitions (for example, energy consumption of the action or execution time).

Given a weighted automaton, an objective function is used to map runs of the automaton to a single numerical value. Formally, an *objective function* is a function mapping infinite sequence of real numbers to a single real number, i.e., $\nu : \mathbb{R}^\omega \to \mathbb{R}$. Given an objective function ν, the *value of a run* $\pi = s_0\sigma_0 s_1\sigma_1 \dots$ is $\nu(v((s_0, \sigma_0, s_1))v((s_1, \sigma_1, s_2)) \dots)$. We abuse notation and write $\nu(\pi)$ for the value fo the run π. Given an objective function ν and a weighted automaton $\mathcal{A}$, the *value of a word* $w \in \Sigma^\omega$ is defined to be $\inf\{\nu(\pi) \mid \pi \text{ is a run of } w \text{ in } \mathcal{A}\}$.

Now, we define the two major objective functions we use in this dissertation:

- *Limit-average objective.* The limit-average objective is used to measure the long run average of weights in a sequence. Formally, $LimAvg(v_0v_1 \dots) = \liminf_n \frac{1}{n} \cdot \sum_{i=0}^{i<n} w_i$.
- *Discounted-Sum objectives.* The discounted-sum objectives are a family of objective functions parameterized by a parameter λ (where $0 \leq \lambda \leq 1$). Intuitively, the discounted-sum objectives compute the sum of a sequence of weights where the weights appearing later in the sequence are given lesser priority than the weights appearing in the initial part. Formally, $Disc_\lambda(v_0v_1 \dots) = \sum_{i=0}^{i=\infty} w_i \cdot \lambda^i$.

2.3 $2\frac{1}{2}$-Player Games

2.3.1 Game Graphs

A 2-*player game graph* G is a tuple $\langle S, (S_1, S_2), \Sigma, \Delta, s_\iota\rangle$ where:

- S is a set of states;
- (S_1, S_2) is a partition of the states S, i.e., $S_1 \cup S_2 = S$;
- Σ is a finite set actions;

- $\Delta \subseteq S \times \Sigma \times S$ is a set of transitions; and
- $s_\iota \in S$ is an initial state.

The states in S_1 and S_2 are known as *Player 1 states* and *Player 2 states*, respectively. As with labelled transition systems, we require that there exists at least one transition from each state, i.e., $\forall s \in S : \exists s' \in S, \sigma \in \Sigma : (s, \sigma, s') \in \Delta$. Intuitively, a 2-player game graph represents a computational system being acted on by two different agents – Player 1 and Player 2. When the system is in a state $s \in S_i$, the next action to be performed is chosen by Player i, and the state of the system changes according to the set of transitions Δ (as in a labelled transition system).

We assume all 2-player game graphs are deterministic – each state and action lead to exactly one different next state, i.e., $(s, \sigma, s') \in \Delta \wedge (s, \sigma, s'') \in \Delta \implies s' = s''$.

When the two players represent the choices internal to a system, we call the corresponding 2-player game graph an *alternating transition system.* Note that the difference between 2-player game graphs and alternating transition systems is only the terminology.

A $2\frac{1}{2}$*-player game graph* G is a tuple $\langle S, (S_1, S_2), \Sigma, \Delta, \delta, s_\iota \rangle$ where S, S_1, S_2, Σ, Δ, and s_ι are as in 2-player game graphs; and $\delta : S \times \Sigma \to \mathcal{D}(S)$ is a *probabilistic transition function* such that:

- Non-negativity over support. $\delta(s, \sigma)(s') \geq 0$ for all $s, s' \in S$ and $\sigma \in \Sigma$, and further, $\delta(s, \sigma)(s') > 0$ if and only if $(s, \sigma, s') \in \Delta$; and
- Support unitarity. $\sum_{s' \in S} \delta(s, \sigma)(s') \in \{0, 1\}$.

We say that an action $\sigma \in \Sigma$ is *enabled* in a state $s \in S$ if and only if $\sum_{s' \in S} \delta(s, \sigma)(s') = 1$. As we are interested in infinite computations, we place an additional restriction that at least one action $\sigma \in \Sigma$ is enabled in every state $s \in S$.

Intuitively, $2\frac{1}{2}$-player game graphs represent computational systems where in addition to actions chosen by the two agents, the next state of the system depends on a probability distribution given by the function δ.

Note that 2-player game graphs can be considered a special case of $2\frac{1}{2}$-player game graphs by setting $\delta(s, \sigma)(s') = 1$ if $(s, \sigma, s') \notin \Delta$ and 0 otherwise.

2.3.2 Strategies and Observations

Histories and Plays. A *history* of a game graph G with initial state s_ι and transitions Δ is a sequence $history = s_0 \sigma_0 s_1 \sigma_1 \ldots s_n$ such that $s_0 = s_\iota$ and for each $0 \leq i < n$, we have that $(s_i, \sigma_i, s_{i+1}) \in \Delta$. The set of all histories of the game graph G are denoted by *Histories*(G), or by *Histories* when G is clear from the context. We define the subset of *Player i histories* $Histories_i \subseteq Histories$ (for $i \in \{1, 2\}$) as the set of histories where the last state is a Player i state, i.e., $Histories_i = \{s_0 \sigma_0 \ldots s_n \mid s_0 \sigma_0 \ldots s_n \in Histories \wedge s_n \in S_i\}$. Intuitively, a history of a game graph is a finite sequence of computational steps resulting due to the choices of the agents.

A *play* of a game graph G is an "infinite" history, i.e., an infinite sequence $\rho = s_0 \sigma_0 s_1 \sigma_1 \ldots$ where $s_0 = s_\iota$ and $(s_i, \sigma_i, s_{i+1}) \in \Delta$ for all $i \in \mathbb{N}$. The set of all plays of the game graph G are denoted by *Plays*(G), or by *Plays* when G is clear from the context. Intuitively, a play of a game graph is the infinite sequence of computational steps resulting from the choices of all the agents.

Strategies and Outcomes. A strategy for a Player i is a recipe for the player to choose actions in the game graph. Formally, a *pure strategy* for Player i in a game graph G is a function $\phi_i : \mathit{Histories}_i(G) \to \Sigma$ such that if $\phi_i(s_0\sigma_0 \dots s_n) = \sigma_n$, then $\exists s_{n+1} \in S : (s_n, \sigma_n, s_{n+1}) \in \Delta$. The set of all Player i strategies in a game graph G is denoted by $\Phi_i(G)$, or if G is clear from the context, by Φ_i.

Remark 2.2. *Note that the above definition is a restricted notion of strategies which is sufficient for all the situations in this monograph. More generally, a randomized strategy does not map each history to a single action, but instead to a probability distribution of actions. From this point on, we use the term strategy to refer to a pure strategy unless otherwise specified.*

A play $\rho = s_0\sigma_0 s_1\sigma_1 \dots$ of G *conforms* to a Player i strategy ϕ_i if for each Player i history $\mathit{history} = s_0\sigma_0 \dots s_n$ that is a prefix of ρ, we have $\phi_i(\mathit{history}) = \sigma_n$.

Given a Player 1 strategy ϕ_1 and a Player 2 strategy ϕ_2, the outcome of the *strategy-profile* (ϕ_1, ϕ_2) (denoted by $\mathit{Outcomes}(\phi_1, \phi_2)$) is the set of all plays that conform both to ϕ_1 and to ϕ_2.

We define two special classes of strategies which are important in the analysis of many types of games and are sufficient for most of the games that appear in this monograph.

- *Memoryless Strategies.* A Player i strategy is memoryless if for each pair of histories $\mathit{history} = s_0\sigma_0 \dots s_n$ and $\mathit{history}' = s'_0\sigma'_0 \dots s'_m$ in $\mathit{Histories}_i$, if $s_n = s'_m$, then $\phi_i(\mathit{history}) = \phi_i(\mathit{history}')$. The set of all memoryless Player i strategies of G is denoted by $\Phi_i^M(G)$.

  Intuitively, a memoryless strategy chooses the next action based only on the current state, i.e., the last state in the history. For a memoryless strategy ϕ_i, we abuse notation and write $\phi_i(s_n)$ instead of $\phi_i(s_0\sigma_0 \dots s_n)$.

- *Finite-Memory Strategies.* A Player i strategy is finite-memory if there exist a deterministic finite automata over the alphabet $\Sigma \times S$ with states M, and a next-action function $\mathit{NextAction} : M \times S \to \Sigma$ such that the following holds: given a Player i history $\mathit{history} = s_0\sigma_0 \dots s_n$, let $m_0(\sigma_0, s_1)m_1(\sigma_1, s_2) \dots (\sigma_{n-1}, s_n)m_n$ be the run of the memory automata corresponding to the word $(\sigma_0, s_1) \dots (\sigma_{n-1}, s_n)$ – then, $\phi_i(\mathit{history}) = \mathit{NextAction}(m_n, s_n)$.

  We call the state-space M of the automaton the *memory* of the strategy, and the transition relation of the automaton the *memory-update relation*.

  Intuitively, a finite-memory strategy chooses the next action for each history based only on a finite amount of information about the history. The set of all finite-memory Player i strategies of G is denoted by $\Phi_i^{FM}(G)$. Note that every memoryless strategy ϕ_i is a finite-memory strategy having a memory automaton with a single state (say m^*) and $\mathit{NextAction}(m^*, s) = \phi_i(s)$ for each $s \in S_i$.

Partial-Observability and Observation-Based Strategies. Given a game graph G with states S, a *observation mapping* is a function *Observe* mapping states to a set of *observations* O. Intuitively, the *Observe* function maps a game graph state to a corresponding observation that contains

only the information "visible" to a given player. We extend the *Observe* function to histories by defining $\mathit{Observe}(s_0\sigma_0 s_1 \dots s_n)$ to be the sequence of $\mathit{Observe}(s_0)\mathit{Observe}(s_1)\dots\mathit{Observe}(s_n)$ for each of the visited states.

Given an observation mapping *Observe*, a Player i strategy ϕ_i is *observation-based* if for every pair of histories *history* and *history'* from $\mathit{Histories}_i$ such that $\mathit{Observe}(\mathit{history}) = \mathit{Observe}(\mathit{history}')$, we have that $\phi_i(\mathit{history}) = \phi_i(\mathit{history}')$.

2.3.3 Games and Objectives

The major question in the analysis of games is whether each player can choose a strategy to enforce that no matter what strategy the opposing player chooses the resulting play has certain properties. Intuitively, a game is defined by a game graph along with an objective for each player. Further, a game may restrict the strategies of each player to observation-based strategies for some observation mappings. In this case, the game is called a *partial information game.* If the set of strategies allowed for the players is unrestricted, the game is called a *perfect information game.*

Objectives. For a game graph G,

- A *boolean objective* Φ is a subset of the set of all plays $\mathit{Plays}(G)$; and
- A *quantitative objective* ν is a function that maps plays to $\mathbb{R}$.

Intuitively, a boolean objective is a set of outcomes that are favourable to Player 1; and a quantitative objective measure the payoff to Player 1, i.e., higher the ν value of a play, the more favourable the play is to Player 1. For a given boolean objective Φ, we say that the play ρ is *winning for Player 1* if $\rho \in \Phi$, and say that the play is *losing for Player 1* if $\rho \notin \Phi$. Dually, ρ is winning for Player 2 if it is losing for Player 1, and ρ is losing for Player 2 if it is winning for Player 1.

We list some standard boolean and quantitative objectives.

- *Safety objective.* The safety objective is defined by a set of safe states T. A play is in $\mathit{Safe}(T)$ if and only if each state in the play belongs to T.
- *Büchi objective.* The Büchi objective is defined by a set of Büchi states B.A A play is in Büchi(B) if and only if there exists an infinite number of states in the play that belong to B.
- *Streett objective.* The Streett objective is defined by a number of request-response Streett pairs $\langle F = (R_0, G_0), (R_1, G_1), \dots, (R_n, G_n)\rangle$ where each R_i and G_i are sets of states. A play ρ is in $\texttt{Streett}(F)$ if and only if for each $0 \leq i \leq n$, we have that R_i is visited a finite number of times in ρ or G_i is visited an infinite number of times in ρ.
- *Parity objective.* The Parity objective is defined by a sequence of sets of states $\mathrm{PS} = \langle P_0, P_1, P_2, \dots P_n\rangle$. A play ρ is in Parity(PS) if and only if the minimum i such that P_i is visited infinitely often in ρ is even.
- *Limit-average objective.* The quantitative limit-average objective measures the long-run average weight of the play is defined by the sequence of

weights of the transitions in the play. Formally, if $w_0 w_1 \ldots$ is the sequence of weights in the play ρ, we have

$$LimAvg(\rho) = \liminf_{n\to\infty} \frac{1}{n} \cdot \sum_{i=0}^{i=n} w_i$$

- *Discounted-sum objective.* The quantitative objective discounted-sum is a family of objectives parameterized by a parameter λ (such that $0 < \lambda < 1$) and measure the sum of the weights of the play such that the earlier weights count more than than the later ones. Formally, if $w_0 w_1 \ldots$ is the sequence of weights in the play ρ, we have

$$\mathrm{Disc}_\lambda(\rho) = \sum_{i=0}^{i=\infty} \lambda^i \cdot w_i$$

- *Supremum and Infimum objectives.* The quantitative objectives supremum and infimum measure the maximum and minimum weight that occurs in a play. Formally, if $w_0 w_1 \ldots$ is the sequence of weights in the play ρ, we have that $\sup(\rho) = \sup_i w_i$ and $\inf(\rho) = \inf_i w_i$.

- *Multi-dimensional limit-average threshold objective.* The boolean objective *MLimAvg* applies in the case where the weight of a transition is not a scalar rational number, but instead a n dimensional vector of rational numbers, i.e., if the weight function has the signature $v : \Delta \to \mathbb{Q}^n$. Given a threshold vector $\mathbf{v}$, we have that a play ρ is in $MLimAvg(v)$ if and only if

$$\liminf_{n\to\infty} \frac{1}{n} \cdot \sum_{i=0}^{i=n} \mathbf{w_i} \leq \mathbf{v}$$

  Note that the weights $\mathbf{w_i}$ are n-dimensional vector here and the limit is taken component wise.

- *Limit-average safety objectives.* The limit-average safety objective is a quantitative objective specified by a separate limit-average objective and a safety objective. Formally, a play has the value assigned by the limit-average objective if it is also in the safety objective; otherwise, it has value ∞.

- *Limit-average parity objectives.* The limit-average safety objective is a quantitative objective specified by a separate limit-average objective and a parity objective. Formally, a play has the value assigned by the limit-average objective if it is also in the parity objective; otherwise, it has value ∞.

Winning Strategies and Optimal Strategies. Given a boolean objective Φ, Player 1 strategy ϕ_1 is a *winning strategy* if every ρ corresponding to ϕ_1 is winning for Player 1, i.e., if $\forall \phi_2 : Outcomes(\phi_1, \phi_2) \subseteq \Phi$. Similarly, a Player 2 strategy ϕ_2 is a *winning strategy* if $\forall \phi_1 : Outcomes(\phi_1, \phi_2) \cap \Phi = \emptyset$.

Given a quantitative objective ν, the *value of a Player 1 strategy* ϕ_1 is defined as $Val_1(\phi_1) = \inf_{\phi_2 \in \Phi_2} \mathbb{E}(\nu(\rho) \mid \rho \in Outcomes(\phi_1, \phi_2))$, i.e., the expected value

of the play arising from the two strategies. Dually, the *value of a Player 2 strategy* ϕ_2 is given by $Val_2(\phi_2) = \sup_{\phi_1 \in \Phi_1} \mathbb{E}(\nu(\rho) \mid \rho \in Outcomes(\phi_1, \phi_2))$. Note that we have not formally defined the probability measure over the outcome of a pair of strategies; see, for example, [42] for a formal definition.

The *Player 1 value* of a game $\mathcal{G}$ is the maximum value Player 1 can enforce for a resulting play no matter what the Player 2 strategy, i.e., it is equivalent to $\sup_{\phi_1 \in \Phi_1} Val_1(\phi_1)$. Similarly, the *Player 2 value* of a game $\mathcal{G}$ is the minimum value Player 2 can enforce for a resulting play, i.e., equivalent to $\inf_{\phi_2 \in \Phi_2} Val_2(\phi_2)$.

A strategy ϕ_i is *optimal for Player i* if it achieves the Player i value for the game, i.e., $Val_i(\phi_i) = Val_i(\mathcal{G})$. Similarly, a strategy ϕ_i is ϵ*-optimal* if $|Val_i(\phi_i) - Val_i(\mathcal{G})| \leq \epsilon$.

2.3.4 Standard Results on 2-player and $2\frac{1}{2}$-player Games

We end this chapter by presenting a number of classical results on 2-player and $2\frac{1}{2}$-player games.

Objective	Decision complexity	Player 1 str.	Player 2 str.
Safety	PTIME	Memoryless	Memoryless
Büchi	PTIME	Memoryless	Memoryless
Streett	EXPTIME	Finite memory	Memoryless
Parity	NP ∩ CO-NP	Memoryless	Memoryless
Limit-average	NP ∩ CO-NP	Memoryless	Memoryless
Discounted-sum	NP ∩ CO-NP	Memoryless	Memoryless
Multi-dimensional limit-average	CO-NP complete	–	Memoryless
Limit-average safety	NP ∩ CO-NP	Finite Memory	Memoryless
Limit-average parity	NP ∩ CO-NP	–	–

Table 2.1: Summary of results for 2-player perfect-information games. The first column is the complexity of deciding if Player 1 has a winning strategy for the Boolean objectives or if the Player 1 value for the game is greater than a given threshold for quantitative objectives. The second and third columns give the class of strategies that are suffice for winning (for Boolean objectives) or optimal play (quantitative objectives).

2-player games. Table 2.1[Pg. 24] summarizes the results for 2-player perfect information (i.e., where the strategies are not based on an observation mapping).

For 2-player partial information games (i.e., where the strategies are restricted to some observation-based strategies), the classical results come from [143]. Further extensions to other objectives are summarized in Table 2.2[Pg. 25].

Objective	Decision complexity	Player 1 str.	Player 2 str.
Safety	PTIME	Memoryless	Memoryless
Büchi	PTIME	Memoryless	Memoryless
Streett		Finite memory	Memoryless
Parity	NP ∩ co-NP	Memoryless	Memoryless
Limit-average	NP ∩ co-NP	Memoryless	Memoryless
Discounted-sum	NP ∩ co-NP	Memoryless	Memoryless
Multi-dimensional limit-average	co-NP complete	–	Memoryless
Limit-average safety	NP ∩ co-NP	Finite Memory	Memoryless
Limit-average parity	NP ∩ co-NP	–	–

Table 2.2: Summary of results for $2\frac{1}{2}$-player partial information games. The columns are as in Table 2.1

$2\frac{1}{2}$-player games. For $2\frac{1}{2}$-player games, the results are of two kinds—in the first category, the algorithmic problem is to compute if a player has a strategy to win with probability 1; and in the second, the algorithmic problem is to compute the exact value of the game for a player. For a comprehensive set of results on $2\frac{1}{2}$-player games related to complexity and class of strategies for each player, we refer the reader to [42].

Relevant results. Here, we explicitly state the results that are used in the remainder of this report.

Theorem 2.3 ([126])**.** *For 2-player games with safety, Büchi, and parity objectives, either Player 1 has a memoryless winning strategy or Player 2 has a memoryless winning strategy.*

Theorem 2.4 ([126])**.** *For 2-player games with Streett objectives, either Player 1 has a finite memory winning strategy or Player 2 has a memoryless winning strategy.*

Theorem 2.5 ([172])**.** *For 2-player games with limit-average, discounted-sum, and limit-average safety objectives, the Player 1 value of the game is equal to the Player 2 value of the game. Further, both players have memoryless optimal strategies.*

Theorem 2.6 ([50])**.** *For 2-player games with limit-average parity objectives, the Player 1 value of the game is equal to the Player 2 value of the game. Further, Player 2 has a memoryless optimal strategy, and Player 1 has finite-memory ϵ-optimal strategies.*

Theorem 2.7 ([51])**.** *For $2\frac{1}{2}$-player games with parity objectives, the Player 1 value of the game is equal to the Player 2 value of the game. Further, both players have optimal memoryless strategies.*

Theorem 2.8 ([73])**.** *For $2\frac{1}{2}$-player partial information games with limit-average objectives, it is undecidable to compute if Player 1 has a winning strategy.*

Part I

Quantities as Preference

In this part, we introduce the simulation distances framework for specifying reactive systems. Here, the classical notion of simulation of an implementation system $\mathcal{I}$ by a specification system $\mathcal{S}$ is replaced by a quantitative correctness distance $d_{\text{cor}}(\mathcal{I}, \mathcal{S})$. A system $\mathcal{I}_1$ is preferred over another $\mathcal{I}_2$ with respect to a specification $\mathcal{S}$ if $d_{\text{cor}}(\mathcal{I}_1, \mathcal{S}) \leq d_{\text{cor}}(\mathcal{I}_2, \mathcal{S})$.

We also show how the same framework can be used to define other useful notions of distances between systems. We define the coverage distance to measure the coverage of a specification by an implementation, i.e., how much of the specification behaviour is covered by the implementation. Further, we define the robustness distance to measure the robustness of an implementation to external errors with respect to a specification.

In Chapter 4[Pg. 67], we show how the simulation distances framework can be used to solve the problem of incompatible specifications. In practice, different parts of the specification come usually from different sources and are often incompatible in the corner cases, i.e., there is no common implementation. The designer then resolves these incompatible requirements and writes more detailed specifications. These detailed specifications are often cumbersome and hard to modify. On the other hand, using simulation distances, we can formalize the notion that the desired implementation is the one that comes closest to satisfying all parts of the specification. Formally, given multiple incompatible specifications, we define the problem of synthesizing from incompatible specifications as finding the implementation that minimizes the maximal simulation distance to these specifications. This is equivalent to stating that in the simulation distances framework, the equivalent of conjunction of two specifications is taking the maximum of the simulation distances to each (i.e., the classical $\mathcal{I} \models \mathcal{S}_1 \wedge \mathcal{I} \models \mathcal{S}_2$ is replaced by $\mathcal{I}$ minimizes $\max(d(\mathcal{I}, \mathcal{S}_1), d(\mathcal{I}, \mathcal{S}_2))$).

Properties of Simulation Distances

A good specification framework is not necessary only to distinguish good systems from bad systems, but it must also offer a host of additional properties and techniques to make standard system development workflows feasible. We discuss a number of properties of simulation distances and the components of the system development workflow they enable. In the classical setting, many of these properties are taken for granted as they are trivially satisfied by most of the specification frameworks. However, each of these properties need to be examined and shown for the quantitative specification frameworks.

Compositional Specifications. Systems are usually composed of multiple components and each of these components is specified separately. Verification usually takes a parallel route, where each component of the implementation system is verified against the corresponding component of the specification. In classical specification frameworks, the property that allows for this is compositionality, which states that is each component of the implementation is correct with respect to the corresponding component of the specification, then so is the implementation with respect to the full specification.

We prove the equivalent quantitative compositionality property for simulation distances for a class of error models. Intuitively, the quantitative compositionality property states that the simulation distance from the specification to the implementation is bounded by the sum of simulation distances from each

component of the specification to the corresponding component of the implementation.

Test-case Generation. In cases where full verification of a system is not possible either due to the size of the system or other constraints, testing is a commonly used technique of system validation. In the testing paradigm, a set of inputs to the system (called the test suite) is chosen and the system behaviour on these inputs is validated against the specification. The success of the testing paradigm in detecting bugs is largely dependent on the choice of the test-suite.

Model-based testing provides a structured and robust technique of choosing a test-suite given a specification and an implementation system. In this framework, a test-suite is chosen to maximize some notion of "coverage" of the specification by the test-suite (for example, state coverage). We show using a case study how the simulation distances framework can be used to generate test-suites in the model-based testing, and how the coverage distance can be used to generate better test-suites than using any classical notion of coverage.

Hierarchical Design. In workflows where a system is designed starting from a specification, the final system is not generated at once from the specification, but instead the development proceeds in steps. From the high level specification, a slightly lower level design is generated where certain, but not all, aspects of the implementation are fixed. In each further step, a lower level system is generated by refining more and more aspects of the design till the concrete implementation system is finalized.

A good specification framework allows the designer to reason about each of these refinement steps separately, i.e., by proving that each refined lower level system is correct with respect to the previous one, the designer should be able to show that the final concrete system is correct with respect to the initial specification. The relevent property for classical specification frameworks is transitivity, i.e., the property that if system $\mathcal{S}_1$ is correct with respect to system $\mathcal{S}_2$, and system $\mathcal{S}_2$ is correct with respect to system $\mathcal{S}_3$, then $\mathcal{S}_1$ is correct with respect to $\mathcal{S}_3$. The quantitative analogue of the classical transitivity property is the triangle inequality which states that $d(\mathcal{S}_1, \mathcal{S}_3) \leq d(\mathcal{S}_1, \mathcal{S}_2) + d(\mathcal{S}_2, \mathcal{S}_3)$. We show that simulation distances follow the triangle inequality for a large class of error models, enabling the same hierarchical development pattern used in the classical case.

Abstraction. A common technique to overcome large state spaces during verification is abstraction. Here, from a given concrete system, a smaller abstract system is constructed such that the abstract system has more behaviours than the concrete system. The guarantee is that if the abstract system is correct with respect to a specification, then so is the concrete system. As the abstract system is smaller than the concrete one, it is potentially easier to verify.

We show that the abstraction similar technique can be used even in the simulation distances framework. Formally, we prove that the distance from a specification to the abstract system is an upper bound on the distance from the specification to the concrete system.

Chapter 3

Simulation Distances

Boolean notions of correctness are formalized by preorders on systems. Quantitative measures of correctness can be formalized by real-valued distance functions between systems, where the distance between implementation and specification provides a measure of "fit" or "desirability." We extend the simulation preorder to the quantitative setting, by making each player of a simulation game pay a certain price for her choices. We use the resulting games with quantitative objectives to define three different simulation distances. The *correctness distance* measures how much the specification must be changed in order to be satisfied by the implementation. The *coverage distance* measures how much the implementation restricts the degrees of freedom offered by the specification. The *robustness distance* measures how much a system can deviate from the implementation description without violating the specification. We consider these distances for safety as well as liveness specifications. The distances can be computed in polynomial time for safety specifications, and for liveness specifications given by weak fairness (Büchi) constraints. We show that the distance functions satisfy the triangle inequality, that the distance between two systems does not increase under parallel composition with a third system, and that the distance between two systems can be bounded from above and below by distances between abstractions of the two systems. These properties suggest that our simulation distances provide an appropriate basis for a quantitative theory of discrete systems. We also demonstrate how the robustness distance can be used to measure how many transmission errors are tolerated by error correcting codes.

3.1 Motivation

Standard verification systems return a boolean answer that indicates whether a system satisfies its specification. However, not all correct implementations are equally good, and not all incorrect implementations are equally bad – there is a preference order among systems. There is thus a natural question whether it is possible to extend the standard specification frameworks and verification algorithms to capture a finer and more quantitative view of the relationship between specifications and systems.

We focus on extending the notion of simulation to the quantitative setting.

For reactive systems, the standard correctness requirement is that all executions of an implementation have to be allowed by the specification. Requiring that the specification simulates the implementation is a stricter condition, but it is computationally less expensive to check. The simulation relation defines a preorder on systems. We extend the simulation preorder to a distance function that given two systems, returns a real-valued distance between them.

Let us consider the definition of simulation of an implementation $\mathcal{I}$ by a specification $\mathcal{S}$ as a two-player game, where Player 1 (the implementation) chooses moves (transitions) and Player 2 (the specification) tries to match each move. The goal of Player 1 is to prove that simulation does not hold, by driving the game into a state from which Player 2 cannot match the chosen move; the goal of Player 2 is to prove that there exists a simulation relation, by playing the game forever. In order to extend this definition to capture how "good" (or how "bad") the simulation is, we make the players pay a certain price for their choices. The goal of Player 1 is then to maximize the cost of the game, and the goal of Player 2 is to minimize it. The cost is given by an objective function, such as the limit average of transition prices. For example, for incorrect implementations, i.e., those for which the specification $\mathcal{S}$ does not simulate the implementation $\mathcal{I}$, we might be interested in how often the specification (Player 2) cannot match an implementation move. We formalize this using a game with a limit-average objective between modified systems. The specification is allowed to "cheat," by choosing mis-matching transitions. However, for each such mismatching transition, the specification player (Player 2) gets a penalty. As Player 2 is trying to minimize the value of the game, she is motivated not to cheat. The value of the game measures how much the specification can be forced to cheat by the implementation. We call this distance function *correctness*.

Let us consider the examples in Figure 3.1[Pg. 33]. We take the system $\mathcal{S}_1$ as the specification. The specification allows at most two symbols b to be output in a row. Now let us consider the two incorrect implementations $\mathcal{I}_3$ and $\mathcal{I}_4$. The implementation $\mathcal{I}_3$ outputs an unbounded number of b's in a row, while the implementation $\mathcal{I}_4$ can output three b's in a row. The specification $\mathcal{S}_1$ will thus not be able to simulate either $\mathcal{I}_3$ or $\mathcal{I}_4$, but $\mathcal{I}_4$ is a "better" implementation in the sense that it violates the requirement to a smaller degree. We capture this by allowing $\mathcal{S}_1$ to cheat in the simulation game by simulating a b transition action by an a transition. Each match or mis-match is penalized according to the error model in Figure 3.6[Pg. 38]. Intuitively, the error model in Figure 3.6[Pg. 38] states that every mis-match gets a penalty of 1 while every match gets a penalty of 0. Hence, as per the error model, every third move will be penalized as the mis-matching transition will be taken every third move while simulating $\mathcal{I}_3$. The correctness distance from $\mathcal{S}_1$ to $\mathcal{I}_3$ will therefore be 1/3. When simulating $\mathcal{I}_4$, the specification $\mathcal{S}_1$ needs to cheat only one in four times—this is when $\mathcal{I}_4$ takes a transition from its state 2 to state 3. The distance from $\mathcal{S}_1$ to $\mathcal{I}_4$ will be 1/4.

Considering the implementation $\mathcal{I}_2$ from Figure 3.1[Pg. 33], it is easy to see that it is correct with respect to the specification $\mathcal{S}_1$. The correctness distance would thus be 0. However, it is also easy to see that $\mathcal{I}_2$ does not include all behaviors allowed by $\mathcal{S}_1$. Our second distance function, *coverage*, is the dual of the correctness distance. It measures how many of the behaviors allowed by the specification are actually implemented by the implementation. This distance is obtained as the value for the implementation in a game in which $\mathcal{I}$ is required to simulate $\mathcal{S}$, with the implementation being allowed to cheat. The coverage

distance can be used, for example, to measure how much of a specification is actually implemented by the system, or as a coverage metric to measure the quality of a test suite. Our third distance function is called *robustness*. It measures how robust the implementation $\mathcal{I}$ is with respect to the specification $\mathcal{S}$ in the following sense: we measure how often the implementation can make an unexpected error (i.e., it performs a transition not present in its transition relation), with the resulting behavior still being accepted by the specification. Unexpected errors could be caused, for example, by a hardware problem, by a wrong environment assumption, unreliable message channels, or by a malicious attack. Robustness measures how many such unexpected errors are tolerated.

In addition to safety specifications, we consider liveness specifications given by weak (Büchi) fairness constraints or strong (Streett) fairness constraints. In order to define distances to liveness specifications, the notion of quantitative simulation is extended to *fair* quantitative simulation. We study variations of the correctness, coverage, and robustness distances using limit-average and discounted objective functions. Limit-average objectives measure the long-run frequency of errors, whereas discounted objectives count the number of errors and give more weight to earlier errors than later ones.

The correctness, coverage, and robustness distances can be calculated by solving the value problem in the corresponding games. Without fairness requirements, we obtain limit-average games or discounted games with constant weights. The values of such games can be computed in polynomial time [172]. We obtain polynomial complexity also for distances between systems with weak-fairness constraints, whereas for strong-fairness constraints, the best known algorithms require exponential time.

We present composition and abstraction techniques that are useful for computing and approximating simulation distances between large systems. We prove that distance from a composite implementation $\mathcal{I}_1 \parallel \mathcal{I}_2$ to a composite specification $\mathcal{S}_1 \parallel \mathcal{S}_2$ is bounded by the sum of distances from $\mathcal{I}_1$ to $\mathcal{S}_1$ and from $\mathcal{I}_2$ to $\mathcal{S}_2$. Furthermore, we show that the distance between two systems can be bounded from above and below by distances between abstractions of the two systems.

Finally, we present case studies showing applications of the robustness distance and the coverage distance. In the first case study, we consider error correction systems for transmitting data over noisy channels and show that the robustness distance measures how many transmission errors can be tolerated by an implementation. Three implementations are analyzed, one based on the Hamming code, one based on triple modular redundancy, and an implementation without any error correction. In the second case study, a specification of a reactive system with inputs and outputs is considered, and we use the coverage metric to determine what part of the input words for which a specification defines an output is covered by different implementations. In the third case study, a specification of a drink vending machine as a reactive system is considered, and the quality of different test suites is measured as the coverage distance to the specification.

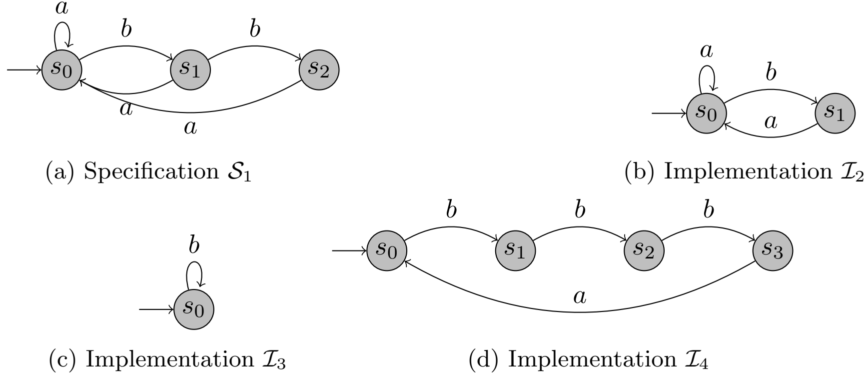


Figure 3.1: Example systems.

3.2 Simulation Relations, Simulation Games, and Quantitative Simulation Games

The simulation preorder [127] is a useful and polynomially computable relation to compare two transition systems. In [12] this relation was extended to alternating simulation between alternating transition systems. For systems with fairness conditions, the simulation relation was extended to fair simulation in [95]. These relations can be computed by solving games with boolean objectives (called simulation games). Here, we extend various types of simulation games with quantitative objectives.

3.2.1 Simulation and Alternating Simulation

Simulation Relations. Consider two labelled transition systems $L = \langle S, \Sigma, \Delta, s_\iota \rangle$, and $L' = \langle S', \Sigma, \Delta', s'_\iota \rangle$. The system L' *simulates* the system L if there exists a relation $\leq_{sim} \subseteq S \times S'$ having the following properties (note that we write $s \leq_{sim} s'$ instead of $(s, s') \in \leq_{sim}$ and say that s' simulates s):

- the initial state of L' simulates the initial state of L, i.e., $s_\iota \leq_{sim} s'_\iota$; and
- if s'_0 simulates s_0, then for each transition from s_0 to s'_0 on action σ, there exists a matching transition from s'_0 to some state s'_1 such that s'_1 simulates s_1. Formally, $\forall s_0, s_1 \in S, s'_0 \in S' : s_0 \leq_{sim} s'_0 \wedge (s_0, \sigma, s_1) \in \Delta \implies \exists s'_1 : (s'_0, \sigma, s'_1) \in \Delta' \wedge s_1 \leq_{sim} s'_1$. This requirement is captured by Figure 3.2[Pg. 34].

If L' simulates L, we abuse notation and write $L \leq_{sim} L'$. Further, if there exists at least one witness $\leq_{sim}$ such that $s \leq_{sim} s'$, we say that s' simulates s.

Intuitively, $s \leq_{sim} s'$ if the behaviour of s' subsumes the behaviour of s in a specific "local" manner, i.e., we want that every transition from s has a matching transition from s' and that the target states of these transitions are also related by $\leq_{sim}$. The following classical theorem shows that this definition of "local" subsumption of behaviour implies the "global" subsumption of behaviour, i.e., language inclusion.

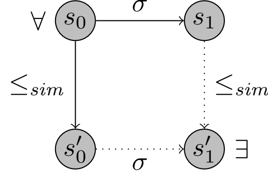


Figure 3.2: Requirements for simulation of s_0 by s'_0

Figure 3.3: Counter-example to converse of Theorem 3.1[Pg. 34]: $Lang(L) \subseteq Lang(L')$, but L' does not simulate L

Theorem 3.1 ([127])**.** *If L' simulates L, then $Lang(L) \subseteq Lang(L')$.*

However, the converse of the above theorem is not true – there are systems L and L' such that $Lang(L) \subseteq Lang(L')$, but L' does not simulate L. For an example of such systems, see Figure 3.3[Pg. 34].

Alternating Simulation. For two alternating transition systems $A = \langle S, (S_{in}, S_{out}), \Sigma, (\Sigma_{in}, \Sigma_{out}), \Delta, s_\iota \rangle$ and $A' = \langle S', (S'_{in}, S'_{out}), \Sigma, (\Sigma_{in}, \Sigma_{out}), \Delta', s'_\iota \rangle$, *alternating simulation* of A by A' (or A' *alternating simulates* A) if there exists a relation $\leq_{asim} \subseteq S_{in} \times S'_{in} \cup S_{out} \times S'_{out}$ such that:

- $s_\iota \leq_{asim} s'_\iota$; and
- for all $s_0 \in S$ and $s'_0 \in S'$ such that $s_0 \leq_{asim} s'_0$, we have that:
  - If $s_0 \in S_{out}$, for all $(s_0, \sigma, s_1) \in \Delta$, there exists $(s'_0, \sigma, s'_1) \in \Delta'$ such that $s_1 \leq_{asim} s'_1$; and
  - If $s_0 \in S_{in}$, for all $(s'_0, \sigma, s'_1) \in \Delta'$, there exists $(s_0, \sigma, s_1) \in \Delta$ such that $s_1 \leq_{asim} s'_1$.

  These requirements are summarized in Figure 3.4[Pg. 35]. If A' alternating simulates A, we abuse notation and write $A \leq_{asim} A'$.

The above definition closely follows the definition of simulation with respect to output actions. However, for input actions, the direction of the matching (simulation) is the other way, i.e., every input transition from the state in A' should be matched by a input transition of A. Intuitively, alternating simulation is the “local” interpretation of the requirement that every output produced by L is produced by L' and every input accepted by L' is accepted by L.

Note that there exist results showing that alternating simulation implies the alternating analogue of language inclusion (alternating trace containment) [12, 11].

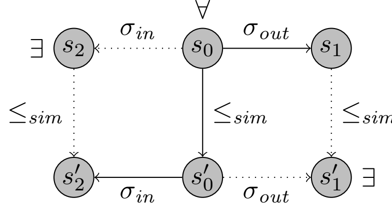


Figure 3.4: Requirements for alternating simulation of s_0 by s'_0

3.2.2 Qualitative Simulation Games

An alternative formulation of simulation and alternating simulation preorders is based on games. Formally, given two labelled transition systems, L and L', we construct a game $\mathcal{G}_{L,L'}$ such that simulation of L by L' holds if and only if Player 2 has a winning strategy in $\mathcal{G}_{L,L'}$. Similarly, for two alternating transition systems, A and A', we can construct a game $\mathcal{H}_{A,A'}$ such that alternating simulation of A by A' holds if and only if Player 2 has a winning strategy in $\mathcal{H}_{A,A'}$. We informally describe the simulation and alternating simulation games below. The formal descriptions follow.

Given two labelled transition systems $L = \langle S, \Sigma, \Delta, s_\iota \rangle$, and $L' = \langle S', \Sigma, \Delta', s'_\iota \rangle$, the simulation game $\mathcal{G}_{L,L'}$ proceeds in the following steps:

1. Initially, the current state of L and L' is set to the corresponding initial states s_ι and s'_ι respectively.
2. In each round, Player 1 picks a transition (say (s_i, σ, s_{i+1})) from the current state s_i of L.
3. Then, Player 2 picks a transition (say $(s'_i, \sigma', s'_{i+1})$) from the current state s'_i of L'.
4. The current states of L and L' are updated to s_{i+1} and s'_{i+1} respectively, and the game proceeds again with a new round (step 2).

Player 2 wins the game if $\sigma = \sigma'$ (i.e., the actions of the transitions match) in each round. Otherwise, Player 1 wins.

Similarly, given two alternating transition systems $A = \langle S, (S_{in}, S_{out}), \Sigma, (\Sigma_{in}, \Sigma_{out}), \Delta, s_\iota \rangle$ and $A' = \langle S', (S'_{in}, S'_{out}), \Sigma, (\Sigma_{in}, \Sigma_{out}), \Delta', s'_\iota \rangle$, the alternating simulation game $\mathcal{H}_{A,A'}$ is proceeds as follows:

1. Initially, the current state of A and A' is set to the corresponding initial states s_ι and s'_ι respectively.
2. In each round, we have:
   - if the current states are output states, as in a simulation game, Player 1 picks a transition (say (s_i, σ, s_{i+1})) from the current state s_i of L, and Player 2 picks a transition (say $(s'_i, \sigma', s'_{i+1})$) from the current state s'_i of L'.
   - if the current states are input states, Player 1 picks a transition (say $(s'_i, \sigma', s'_{i+1})$) from the current state s'_i of L', and Player 2 picks a transition (say (s_i, σ, s_{i+1})) from the current state s_i of L.
3. The current states of L and L' are then updated to s_{i+1} and s'_{i+1} respectively, and the game proceeds again with a new round (step 2).

As before, Player 2 wins the game if $\sigma = \sigma'$ (i.e., the actions of the transitions match) in each round. Otherwise, Player 1 wins.

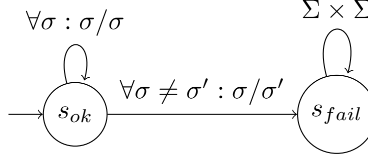


Figure 3.5: Standard Error Automaton

It can be seen that the simulation and alternating simulation games actually correspond to the simulation and alternating simulation preorders, i.e., Player 2 wins in the game if and only if the corresponding relation between the systems holds. Intuitively, if simulation (or alternating simulation) holds, Player 2 can ensure that the current states in each round (s_i and s'_i) are in simulation (or alternating simulation). This holds as s_ι and s'_ι are in simulation (or alternating simulation) initially, and in each step, Player 2 can choose the matching transition from the definition of the witness simulation relation $\leq_{sim}$ (or witness alternating simulation relation $\leq_{asim}$). Conversely, given a winning strategy for Player 2 in the game, a witness simulation (or alternating simulation) relation can be constructed by collecting all the pairs of current states reached in each round over all the strategies for Player 1.

Error Automata. For the formal definition of the simulation and alternating simulation games, we use error automata. Though this makes the definition more involved here, it simplifies further definitions in the next section (Section 3.2.3[Pg. 38]).

Given a set of actions Σ, an *error automata* is an automata $\langle S, \Sigma \times \Sigma, \Delta, s_\iota \rangle$. Note that in error automata, we write the elements (σ, σ') of the alphabet $\Sigma \times \Sigma$ as σ/σ' instead. We will use error automata to characterize the matching of the actions in each round of the simulation game, i.e., error automata are used to separate valid simulations from invalid ones.

The *standard error automaton* (See Figure 3.5[Pg. 36]) over actions Σ is an automaton $\langle \{s_{ok}, s_{fail}\}, \Sigma \times \Sigma, \Delta, s_{ok} \rangle$ where $\Delta = \{(s_{ok}, \sigma/\sigma, s_{ok}) \mid \sigma \in \Sigma\} \cup \{(s_{ok}, \sigma/\sigma', s_{fail}) \mid \sigma, \sigma' \in \Sigma \wedge \sigma \neq \sigma'\} \cup \{(s_{fail}, \sigma/\sigma', s_{fail}) \mid \sigma, \sigma' \in \Sigma\}$ along with the safety acceptance condition given by $\mathit{Safe}(\{s_{ok}\})$. Intuitively, the standard error automata accepts a word if and only if at each position of the word the two actions in the action pair are equal.

Simulation and Alternating Simulation Games Given two labelled transition systems $L = \langle S, \Sigma, \Delta, s_\iota \rangle$, and $L' = \langle S', \Sigma, \Delta', s'_\iota \rangle$ and an error automata $\mathcal{A} = \langle S^{\mathcal{A}}, \Sigma \times \Sigma, \Delta^{\mathcal{A}}, s_\iota^{\mathcal{A}} \rangle$ the *simulation game* $\mathcal{G}_{L,L',\mathcal{A}}$ is played over the *simulation game graph* $G = \langle S^G, (S_1^G, S_2^G), \Sigma, \Delta^G, s_\iota^G \rangle$ where:

- The state space $S^G = S \times (\Sigma \cup \{\#\}) \times S' \times S^{\mathcal{A}} \times \{1, 2\}$ consists of:
  - The current states of L and L' in the simulation game at the first and third components respectively;
  - The current player (either Player 1 or Player 2) at the last component;

- The action of the previous L transition chosen by Player 1 in the second component, or # if Player 1 is yet to choose a transition in this round; and
- The current state of the error automaton in the fourth component.

- The initial state $s_\iota^G = (s_\iota, \#, s'_\iota, s_\iota^{\mathcal{A}}, 1)$.
- A state in S^G is in S_i^G if and only if the last component is equal to i.
- The set of transitions Δ^G consists of two types of transitions:
  - *Testing transitions.* If (s_0, σ, s_1) is a transition of L in Δ, then $((s_0, \#, s'_0, s_0^{\mathcal{A}}, 1), \sigma, (s_1, \sigma, s'_0, s_0^{\mathcal{A}}, 2)) \in \Delta^G$; and
  - *Matching transitions.* If (s'_0, σ', s'_1) is a transition of L' in Δ' and $(s_0^{\mathcal{A}}, \sigma/\sigma', s_1^{\mathcal{A}})$ is transition of the error automaton, then $((s_1, \sigma, s'_0, s_0^{\mathcal{A}}, 2), \sigma', (s_1, \#, s'_1, s_1^{\mathcal{A}}, 1)) \in \Delta^G$.

A play in this game graph is winning for Player 2 if and only if the run of the error automaton that is embedded in the states of the play is accepting.

We have the following theorem.

Theorem 3.2. *Given two labelled transition systems L and L' over the actions and the standard error automaton $\mathcal{A}$ over Σ, Player 2 has a winning strategy in the corresponding simulation game $\mathcal{G}_{L,L',\mathcal{A}}$ if and only if L' simulates L.*

We skip the proof of the theorem and instead refer the reader to any standard text on model checking of discrete time systems.

Similarly, given two alternating transition systems $A = \langle S, (S_{in}, S_{out}), \Sigma, (\Sigma_{in}, \Sigma_{out}), \Delta, s_\iota \rangle$ and $A' = \langle S', (S'_{in}, S'_{out}), \Sigma, (\Sigma_{in}, \Sigma_{out}), \Delta', s'_\iota \rangle$ and an error automaton $\mathcal{A} = \langle S^{\mathcal{A}}, \Sigma \times \Sigma, \Delta^{\mathcal{A}}, s_\iota^{\mathcal{A}} \rangle$ the *alternating simulation game* $\mathcal{H}_{A,A',\mathcal{A}}$ is played over the *alternating simulation game graph* $H = \langle S^H, (S_1^H, S_2^H), \Sigma, \Delta^H, s_\iota^H \rangle$ where:

- The state space, Player 1 and Player 2 states, actions and the initial state of $H_{A,A',\mathcal{A}}$ are defined as in a simulation game; and
- The set of transitions Δ^H consists of four types of transitions:
  - *Output Testing transitions.* If (s_0, σ, s_1) is an output transition of A in Δ, then $((s_0, \#, s'_0, s_0^{\mathcal{A}}, 1), \sigma, (s_1, \sigma, s'_0, s_0^{\mathcal{A}}, 2)) \in \Delta^H$; and
  - *Output Matching transitions.* If (s'_0, σ', s'_1) is a transition of A' in Δ' and $(s_0^{\mathcal{A}}, \sigma/\sigma', s_1^{\mathcal{A}})$ is transition of the error automaton, then $((s_1, \sigma, s'_0, s_0^{\mathcal{A}}, 2), \sigma', (s_1, \#, s'_1, s_1^{\mathcal{A}}, 1)) \in \Delta^H$.
  - *Input Testing transitions.* If (s'_0, σ', s'_1) is an output transition of A' in Δ, then $((s_0, \#, s'_0, s_0^{\mathcal{A}}, 1), \sigma', (s_0, \sigma', s'_1, s_0^{\mathcal{A}}, 2)) \in \Delta^H$; and
  - *Input Matching transitions.* If (s_0, σ, s_1) is a transition of A in Δ and $(s_0^{\mathcal{A}}, \sigma'/\sigma, s_1^{\mathcal{A}})$ is transition of the error automaton, then $((s_0, \sigma', s'_1, s_0^{\mathcal{A}}, 2), \sigma, (s_1, \#, s'_1, s_1^{\mathcal{A}}, 1)) \in \Delta^H$

As for simulation games, we have the following theorem for alternating simulation games.

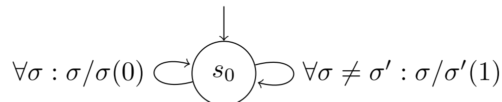


Figure 3.6: Standard Error Model

Theorem 3.3. *Given two labelled transition systems A and A' over the actions and the standard error automaton $\mathcal{A}$ over Σ, Player 2 has a winning strategy in the corresponding alternating simulation game $\mathcal{H}_{A,A',\mathcal{A}}$ if and only if A' alternating simulates A.*

Fair Simulation Games. Given two labelled transitions systems with fairness conditions L^F and $L'^{F'}$, the fair simulation game is played in the same game graph $G_{A,A'}$ as the simulation game. However, in addition to matching the symbol in each step, Player 2 has to ensure that if the trace produced by the sequence of transitions of L chosen by Player 1 satisfies is fair, then the trace produced by the sequence of L' transitions chosen is also fair.

3.2.3 Quantitative Simulation Games

We define a generalized notion of simulation games called quantitative simulation games where the simulation objectives of matching each transition are replaced by a quantitative objectives.

Error Models. In quantitative simulation games, we allow Player 2 to match transitions on one action with transitions on a different action. However, we assign a cost to such mismatches and Player 2 tries to minimize the cost of mismatches in the game. These costs are specified using error models, which are the quantitative analogue of error automata.

Given a set of actions Σ, an *error model* is a deterministic weighted automaton $\langle S, \Sigma \times \Sigma, \Delta, s_\iota, v\rangle$ with either a *LimAvg* or a $Disc_\lambda$ objective. As with error automata, we write the members of the alphabet of error models as σ/σ' instead of (σ, σ'). We also require that each word over symbols of the form σ/σ is assigned cost 0 to ensure that correct simulations incur no penalty.

For a set of actions Σ, the *standard error model* (see Figure 3.6) is the weighted automaton $\langle\{s\}, \Sigma \times \Sigma, \Delta, s\rangle$ with the weight function v where: (a) $\Delta = \{(s, \sigma/\sigma', s) \mid \sigma, \sigma' \in \Sigma\}$, and (b) $v((s, \sigma/\sigma', s)) = 0$ if $\sigma = \sigma'$ and $v((s, \sigma/\sigma', s)) = 1$ if $\sigma \neq \sigma'$.

Intuitively, the error models specify quantitatively the "quality" of match for each simulation run. Intuitively, a transition on label represents that a transition on label σ_1 in the simulated system can be simulated by a transition on label σ_2 in the simulating system with the accompanying cost. In a simulation game, the sequence of transitions chosen by Player 1 form the L trace $s_0\sigma_0 s_1\sigma_1 \ldots$, and the sequence of transitions chosen by Player 2 form the L' trace $s'_0\sigma'_0 s'_1\sigma'_1 \ldots$, the "quality" of match is given the value of the run of the error model on the word $(\sigma_0/\sigma'_0)(\sigma_1/\sigma'_1)\ldots$.

We present a few natural error models here.

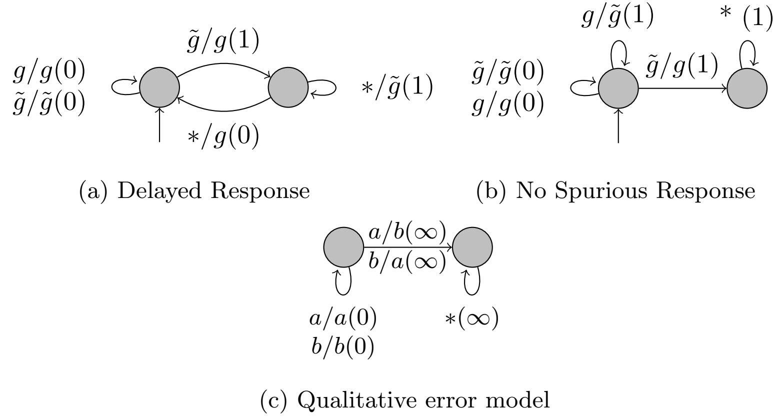


Figure 3.7: Sample error models: the first component of the symbol on each transition is the output symbol expected by the specification, while the second component is the output symbol actually produced by the implementation.

- *Standard Error Model.* (Figure 3.6[Pg. 38]) Every replacement can occur during simulation with a constant cost and can be used to model "one-off" errors like bit-flips.
- *Delayed Response Model.* (Figure 3.7a[Pg. 39]) This model measures the timeliness of responses (g) to requests (r). Here, when the implementation outputs $\tilde{g}$ when a grant g is expected, all transitions have a penalty until the missing grant g is seen. Under this model, the cost of the simulation is the fraction of time spent waiting for the grant.
- *No Spurious Response Model.* (Figure 3.7b[Pg. 39]) This model is meant to ensure that no spurious grants are produced. If an implementation produces a grant not required by the specification, all subsequent transitions get a penalty.
- *Qualitative Model.* (Figure 3.7c[Pg. 39]) This model recovers the boolean simulation games. The distance is 0 if and only if the simulation relation holds.

Remark 3.4. *In [36], we introduced the notion of modification schemes to characterize the kinds of errors or mismatches permitted during the simulation game. However, the modification schemes used in [36] do not cover some natural kinds of error schemes. Of the above models, the standard model and the qualitative model can be expressed as modification schemes from [36], whereas the delayed response and the spurious response model cannot be cast as modification schemes.*

Quantitative Simulation Games. Quantitative simulation games are defined in exactly the same way as standard simulation games, the only difference being that the boolean objective obtained from the error automaton is replaced with the quantitative objective obtained from the error model.

Formally, given two labelled transition systems $L = \langle S, \Sigma, \Delta, s_\iota \rangle$, and $L' = \langle S', \Sigma, \Delta', s'_\iota \rangle$ and an error model $\mathcal{M} = \langle S^{\mathcal{M}}, \Sigma \times \Sigma, \Delta^{\mathcal{M}}, s_\iota^{\mathcal{M}}, v^{\mathcal{M}} \rangle$ the

corresponding *quantitative simulation game* is played on the weighted game graph $Q = \langle S^Q, (S_1^Q, S_2^Q), \Sigma, \Delta^Q, s_\iota^Q, v\rangle$ where:

- $\langle S^Q, (S_1^Q, S_2^Q), \Sigma, \Delta^Q, s_\iota^Q, v\rangle$ is the same as the standard simulation game graph $G_{L,L',\mathcal{A}}$ where $\mathcal{A} = \langle S^{\mathcal{M}}, \Sigma \times \Sigma, \Delta^{\mathcal{M}}, s_\iota^{\mathcal{M}}$.
- The weight function v is specified as follows for the different cases:
  - *LimAvg* case. If $\mathcal{M}$ has a limit-average objective, we define the weight functions as follows:
    (a) For testing transitions $((s_0, \#, s_0', s_0^{\mathcal{A}}, 1), \sigma, (s_1, \sigma, s_0', s_0^{\mathcal{A}}, 2)) \in \Delta^Q$, we set $v(((s_0, \#, s_0', s_0^{\mathcal{A}}, 1), \sigma, (s_1, \sigma, s_0', s_0^{\mathcal{A}}, 2))) = 0$; and
    (b) For matching transitions $((s_1, \sigma, s_0', s_0^{\mathcal{A}}, 2), \sigma', (s_1, \#, s_1', s_1^{\mathcal{A}}, 1)) \in \Delta^Q$, the set $v(((s_1, \sigma, s_0', s_0^{\mathcal{A}}, 2), \sigma', (s_1, \#, s_1', s_1^{\mathcal{A}}, 1))) = 2 \cdot v^{\mathcal{M}}((s_0^{\mathcal{A}}, \sigma/\sigma', s_1^{\mathcal{A}}))$, i.e., we set the weight to be twice the weight of the transition in the corresponding error model.
  - *Disc*$_\lambda$ case. If $\mathcal{M}$ has a discounted sum objective with discount factor λ, we define the weight functions as follows:
    (a) As for the limit-average case, the testing transitions have weight 0; and
    (b) For matching transitions, we set the weight to be a factor of $\frac{2}{\sqrt{\lambda}}$ of the weight of the corresponding error model transition.

The objective of the quantitative simulation game is the limit-average objective if the error model $\mathcal{M}$ was a limit-average automaton, and it is a $Disc_{\sqrt{\lambda}}$ objective if the error model was a $Disc_\lambda$ automaton. The scaling factors for the weights and the change from $Disc_\lambda$ to $Disc_{\sqrt{\lambda}}$ while going from the error model to the game graph is to normalize the values of the game.

Similarly, for two alternating transition systems, we can define the *quantitative alternating simulation game* based on the standard alternating simulation game. The weight function for this game is defined in the same way as for quantitative simulation games. We do not write the full definition here, but instead just mention that for alternating systems A and A', and error model $\mathcal{M}$, we denote the corresponding quantitative simulation as $\mathcal{P}_{A,A',\mathcal{M}}$ and the quantitative simulation game graph as $R_{A,A',\mathcal{M}}$.

Quantitative Fair Simulation Games. Given two transition systems L and L' with fairness constraints (and an error model), as with standard fair simulation games, the quantitative fair simulation games are played on the same graph as the corresponding quantitative simulation game. The objective of the game is modified to be the hybrid objective consisting of quantitative objective from the error model and the boolean objective that if the L trace produced by the transitions chosen by Player 1 is fair, then the L' trace produced by the transitions chosen by Player 2 must be fair – in other words, the value of a play is ∞ if the L trace is fair and the L' trace is not fair, and the value of the trace is given by the weight function otherwise.

Note that we do not use discounted objective ($Disc_\lambda$) along with fairness conditions as the two objectives are independent. The *Disc* objectives mainly consider the finite prefix of a play, whereas fairness conditions consider only the infinite suffix. Whenever a quantitative (alternating) simulation game with *Disc*

objectives is mentioned, it is understood that there are no fairness conditions on the systems.

3.3 Simulation Distances

We present examples of distances that can be defined using quantitative simulation games. The first two presented here are modifications of the classical simulation game in which the simulation relation is not perfect. The games are set up in such a way that the amount of imperfection in the simulation measures a property of the relation between the systems. The third game presented measures the robustness of a system in a novel way.

Remark 3.5 (A note about system nomenclature)**.** *Although quantitative simulation games can be constructed between any two systems over the same set of actions, in the following discussion, it is usually convenient to think of one of the systems involved as the* specification *and the other as a candidate* implementation. *This nomenclature is justified as the simulation relation and the simulation pre-order is generally used as a notion of refinement of systems.*

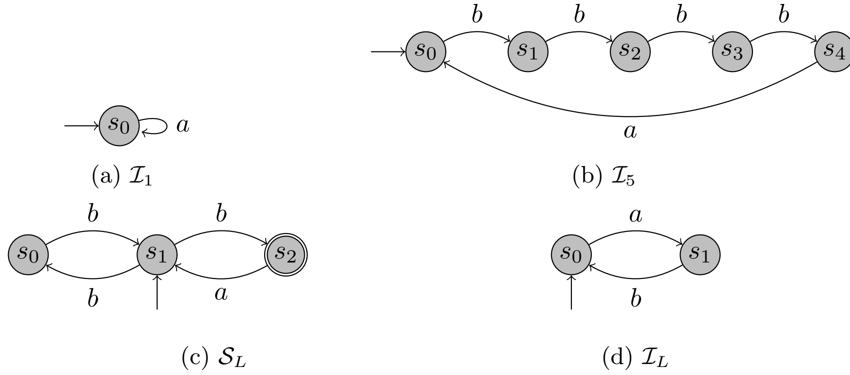
3.3.1 Correctness

Given a specification $\mathcal{S}$ and an implementation $\mathcal{I}$, such that $\mathcal{I}$ is incorrect with respect to $\mathcal{S}$, the correctness distance measures the degree of "incorrectness" of $\mathcal{I}$. The boolean (fair) simulation relation is very strict in a certain way. Even a single nonconformant behavior can destroy this relation. Here we present a game which is not as strict and measures the minimal number of required errors, i.e. the minimal number of times the specification has to use nonmatching symbols when simulating the implementation.

Let $\mathcal{I}$ and $\mathcal{S}$ be two labelled (resp. alternating) transition systems with the same set of actions. The *correctness distance* ${d_{\text{cor}}}^{\mathcal{M}}(\mathcal{I}, \mathcal{S})$ from system $\mathcal{I}$ to system $\mathcal{S}$ with respect to an error model $\mathcal{M}$ is the value of the quantitative simulation game ${\mathcal{G}_{\text{cor}}}^{\mathcal{M}}(\mathcal{I}, \mathcal{S})$ where one of the following holds:

- The systems $\mathcal{I}$ and $\mathcal{S}$ are labelled transition systems without fairness constraints, and ${\mathcal{G}_{\text{cor}}}^{\mathcal{M}}(\mathcal{I}, \mathcal{S})$ is the quantitative simulation game $\mathcal{Q}_{\mathcal{I},\mathcal{S},\mathcal{M}}$; or
- The systems $\mathcal{I}$ and $\mathcal{S}$ are alternating transition systems without fairness constraints, and ${\mathcal{G}_{\text{cor}}}^{\mathcal{M}}(\mathcal{I}, \mathcal{S})$ is the alternating quantitative simulation game $\mathcal{P}_{\mathcal{I},\mathcal{S},\mathcal{M}}$; or
- The systems $\mathcal{I}$ and $\mathcal{S}$ are labelled transition systems with fairness constraints, and ${\mathcal{G}_{\text{cor}}}^{\mathcal{M}}(\mathcal{I}, \mathcal{S})$ is the fair quantitative simulation game ${\mathcal{Q}^{fair}}_{\mathcal{I},\mathcal{S},\mathcal{M}}$; or
- The systems $\mathcal{I}$ and $\mathcal{S}$ are alternating transition systems without fairness constraints, and ${\mathcal{G}_{\text{cor}}}^{\mathcal{M}}(\mathcal{I}, \mathcal{S})$ is the fair quantitative alternating simulation game ${\mathcal{P}^{fair}}_{\mathcal{I},\mathcal{S},\mathcal{M}}$.

The game $\mathcal{G}_{\text{cor}}$ can be intuitively understood as follows. Given two systems $\mathcal{I}$ and $\mathcal{S}$, we are trying to simulate the system $\mathcal{I}$ by $\mathcal{S}$, but the specification $\mathcal{S}$ is allowed to make errors during simulation, i.e., to "cheat", but she has to pay a price for such a choice. As the simulating player is trying to minimize the value of the game, she is motivated not to cheat. The value of the game can thus be



(a) $\mathcal{I}_1$ (b) $\mathcal{I}_5$

(c) $\mathcal{S}_L$ (d) $\mathcal{I}_L$

Figure 3.8: Example Systems

seen as measuring how much she can be forced to cheat, that is, how much the implementation commits an error.

We interpret the correctness distance as a measure of how much an implementation deviates from an ideal specification. For example, if the implementation is correct ($\mathcal{S}$ simulates $\mathcal{I}$), then the correctness distance is 0. Otherwise, the correctness distance measures the "farthest" implementation behaviour from the given specification. When using the standard error model, the value of the game is either the limit-average or the discounted-sum of the number of errors, i.e., if the objective in the error model is *LimAvg*, then the value is the long run average of the errors, whereas if the objective is $Disc_\lambda$, the errors which occur earlier are given more importance and the value is the discounted sum of the positions of the errors.

Example Systems and Distances We present a few example systems and their distances here to demonstrate the fact that the above game measures distances that correspond to intuition. In Figure 3.8[Pg. 42] and Figure 3.1[Pg. 33], $\mathcal{S}_1$ is the specification system against which we want to measure the systems $\mathcal{I}_1$ through $\mathcal{I}_5$ using the standard error model. In this case, the specification says that there cannot be more than two b's in a row. Also, we have a specification with a liveness condition $\mathcal{S}_L$ against which we want to measure the implementation $\mathcal{I}_L$. The distances between these systems according to the *LimAvg* correctness game with the standard error model (Figure 3.6[Pg. 38]) are summarized in Table 3.1[Pg. 43].

Among the systems which do not satisfy the specification $\mathcal{S}_1$, i.e. $\mathcal{I}_3$, $\mathcal{I}_4$ and $\mathcal{I}_5$, we showed in the introduction that the distance from $\mathcal{I}_3$ to $\mathcal{S}_1$ is $1/3$, while the distance from $\mathcal{I}_4$ to $\mathcal{S}_1$ is $1/4$. However, surprisingly the distance from $\mathcal{I}_5$ to $\mathcal{S}_1$ is less than the distance from $\mathcal{I}_4$. In fact, the distances reflect on the long run the number of times the specification has to err to simulate the implementation.

In case of the specification $\mathcal{S}_L$ and implementation $\mathcal{I}_L$ with liveness conditions, the specification can take the left branch to state s_0 to get a penalty of $\frac{1}{2}$ or take the right branch to state s_2 to get a penalty of 1. However, it needs to take the right branch infinitely often to satisfy the liveness condition. To achieve the distance of $\frac{1}{2}$, the specification needs infinite memory so that it can take the right branch lesser and lesser number of times. In fact, if the specification has

T_1	T_2	$d_{\text{cor}}{}^{LimAvg}(T_1,T_2)$	$d_{\text{cov}}{}^{LimAvg}(T_1,T_2)$	$d_{\text{rob}}{}^{LimAvg}(T_1,T_2)$
$\mathcal{S}_1$	$\mathcal{S}_1$	0	0	1
$\mathcal{I}_1$	$\mathcal{S}_1$	0	2/3	1/3
$\mathcal{I}_2$	$\mathcal{S}_1$	0	1/3	2/3
$\mathcal{I}_3$	$\mathcal{S}_1$	1/3	1	1
$\mathcal{I}_4$	$\mathcal{S}_1$	1/4	1	1
$\mathcal{I}_5$	$\mathcal{S}_1$	1/5	1	1
$\mathcal{I}_L$	$\mathcal{S}_L$	1/2	1	1

Table 3.1: Distances according to the correctness, coverage, and robustness games.

a strategy with finite-memory of size m, it can achieve a distance of $\frac{1}{2} + \frac{1}{2m}$.

Further, we present an example of how using simulation distances can simplify the specification of certain requirements. In many cases in practice, the real requirement on some system is the minimization or maximization of certain events or conditions. However, due to the use of boolean specification frameworks, these requirements are extremely hard to express.

Example 3.6. *Consider a request-grant requirement which states that every request r has to be finally granted with g (seen in Figure 3.9a[Pg. 44] – in the figure, state s_0 is a Büchi state). In [130], the authors analyze the additional specifications that need to be added to ensure that there are no spurious grants, i.e., g is never output when there is no pending request r. To this goal, the authors add the following two additional specification requirements: (a) there are no grants before the first request; and (b) there are no two grants without a request in between. Although, these additional requirements achieve the goal, they are neither trivial to come up with, nor do they state exactly the required condition – the authors develop a tool to help construct and debug such additional requirements in [130].*

Another significant problem is that these additional specifications need to be changed whenever the original requirement changes. For example, if the original requirement changes to every request has to be finally granted, and there should be at least one g every 10 time steps (say to keep the system "alive"), the additional requirements need to be rewritten completely.

However, using correctness distances, the same effect can be achieved easily through adding an ideal specification that states that g never be output (Figure 3.9b[Pg. 44], along with an error model that penalizes every replacement of $\neg g$ with g (Figure 3.9c[Pg. 44]) and choosing implementations that minimize this correctness distance while satisfying the original specification. Therefore, implementations that do not grant spuriously while still granting every request will be preferred. Also, note that changing the original specification in this setting does not require changing the additional minimization requirement either.

3.3.2 Coverage

We present the dual of the correctness distance which measures the behaviors present in specification $\mathcal{S}$ system but not in the implementation $\mathcal{I}$. Intuitively, the coverage distance measures "how much" of the specification is actually im-

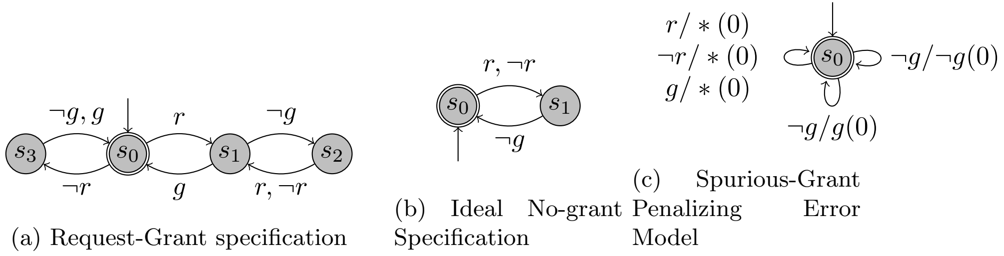


Figure 3.9: Minimizing spurious grants

plemented in the implementation. Hence, we have that the coverage distance from $\mathcal{I}$ to $\mathcal{S}$ is the correctness distance from $\mathcal{S}$ to $\mathcal{I}$.

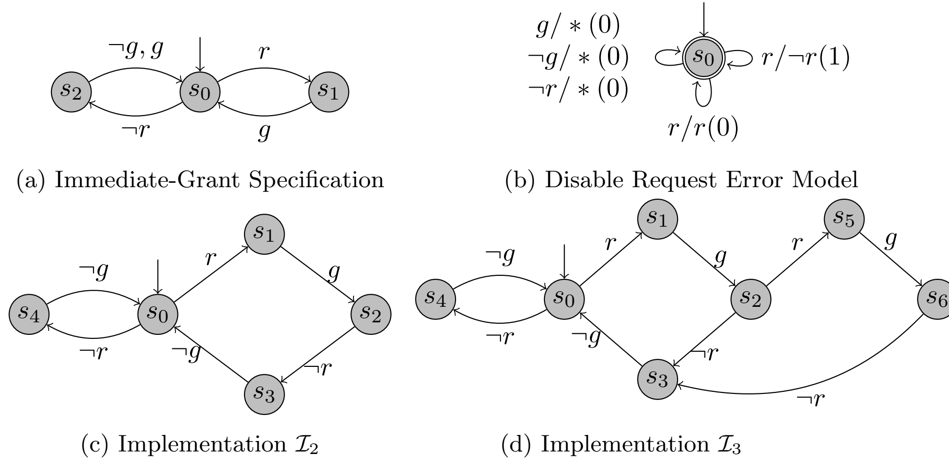
Given systems $\mathcal{S}$ and $\mathcal{I}$ and an error model $\mathcal{M}$, the *coverage distance* $d_{\text{cov}}(\mathcal{I}, \mathcal{S})$ from system $\mathcal{I}$ to system $\mathcal{S}$ is the value of the quantitative simulation game $\mathcal{G}_{\text{cov}\mathcal{I},\mathcal{S},\mathcal{M}}$ where if $\mathcal{S}$ and $\mathcal{I}$ are:

- Labelled transition systems without fairness constraints, ${\mathcal{G}_{\text{cov}}}^{\mathcal{M}}(\mathcal{I}, \mathcal{S})$ is the quantitative simulation game $\mathcal{Q}_{\mathcal{S},\mathcal{I},\mathcal{M}}$.
- Alternating transition systems without fairness constraints, ${\mathcal{G}_{\text{cov}}}^{\mathcal{M}}(\mathcal{I}, \mathcal{S})$ is the alternating quantitative simulation game $\mathcal{P}_{\mathcal{S},\mathcal{I},\mathcal{M}}$.
- Labelled transition systems with fairness constraints, ${\mathcal{G}_{\text{cov}}}^{\mathcal{M}}(\mathcal{I}, \mathcal{S})$ is the fair quantitative simulation game ${\mathcal{Q}^{fair}}_{\mathcal{S},\mathcal{I},\mathcal{M}}$.
- Alternating transition systems without fairness constraints, ${\mathcal{G}_{\text{cov}}}^{\mathcal{M}}(\mathcal{I}, \mathcal{S})$ is the fair quantitative alternating simulation game ${\mathcal{P}^{fair}}_{\mathcal{S},\mathcal{I},\mathcal{M}}$.

$\mathcal{G}_{\text{cov}}$ measures the minimal number of errors that have to be committed by $\mathcal{I}$ to cover all the behaviors of $\mathcal{S}$. We summarized some examples systems and their limit-average coverage distances with respect to the standard error model in Table 3.1[Pg. 43]. Further, we present an example showing how the coverage distance can be used to measure the "completeness" of an implementation.

Example 3.7. *Suppose we have a request-grant system where the specification states that every request has to be granted immediately (Figure 3.10a[Pg. 45]). Also, suppose due to physical constraints, implementing a system that successively granting multiple requests is difficult. Therefore, the implementation systems compensate by disallowing requests at certain points of time. We can now measure how complete the implementation systems are with respect to the complete specification using the coverage distance with respect to the error model shown in Figure 3.10b[Pg. 45]. The error model gives a penalty whenever a request r is ignored, i.e., replaced with a $\neg r$ – it does not care about the outputs g and $\neg g$.*

For example, the implementation $\mathcal{I}_2$ shown in Figure 3.10c[Pg. 45] has a coverage distance $\frac{1}{4}$ to the specification, while the implementation $\mathcal{I}_3$ shown in Figure 3.10d[Pg. 45] has a coverage distance of $\frac{1}{6}$. This intuitively corresponds to the idea that in the worst case, the $\mathcal{I}_2$ disables requests in every second step, while $\mathcal{I}_3$ disables requests in every third step.



(a) Immediate-Grant Specification (b) Disable Request Error Model

(c) Implementation $\mathcal{I}_2$ (d) Implementation $\mathcal{I}_3$

Figure 3.10: Minimizing disabled requests

3.3.3 Robustness

In a perfectly functioning system, errors may occur due to unpredictable events – for example, due to uncontrollable bit-flips during message passing. Given a specification system and a correct implementation of the specification, the notion of robustness presented here is a measure of the number of external errors that need to be introduced into the implementation behaviour that makes it nonconformant to the specification. The more such errors tolerated by the specification, the more robust the implementation is with respect to the specification. The lower the value of the robustness distance to a given specification, the more robust an implementation is. In case of an incorrect implementation, the simulation of the implementation does not hold irrespective of implementation errors. Hence, in that case, the robustness distance will be ∞.

Remark 3.8. *In this section, we identify every labelled transition system* $L = \langle S, \Sigma, \Delta, s_\iota \rangle$ *with the alternating transition system* $\langle S, (\emptyset, S), \Sigma, (\emptyset, \Sigma), \Delta, s_\iota$, *i.e., we consider all the actions and states of the labelled transition system to be output actions and output states respectively.*

Robustness Measuring Modification. To define the robustness distance, we need to modify the systems to explicitly encode both the occurrence of an external error, and the kind of external errors that might happen. To this effect, we introduce a *modification scheme ContErr* – a function that extends systems with additional behaviours relating to the modelling of errors. First, we describe the function *ContErr* informally. The *ContErr* function is parameterized by a set of pairs of output actions $\mathit{ErrPairs} \subseteq \Sigma_{out} \times \Sigma_{out}$. Intuitively, if $(\sigma_{out}, \sigma'_{out}) \in \mathit{ErrPairs}$, it means that an external error may force the the system to output σ'_{out} instead of σ_{out}. For example, an action to send the bit 0 over the network (say send_0) might get replaced by an action to send bit 1 over the network (say send_1) due to a bit-flip – we model this fact by letting $(\mathit{send}_0, \mathit{send}_1) \in \mathit{ErrPairs}$. However, an action to send the bit 0 over the network cannot get replaced by a completely unrelated action, say action

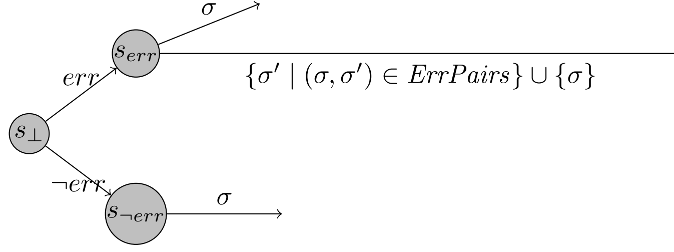


Figure 3.11: Gadget for *ContErr*

$zero(r_1)$ to set register r_1 to 0, due to any feasible external error – hence, $(send_0, zero(r1)) \notin ErrPairs$.

Given a system $\mathcal{S}$ and the set of possible errors *ErrPairs*, the system $ContErr(ErrPairs, \mathcal{S})$ is obtained by replacing every output state s with the gadget shown in Figure 3.11[Pg. 46]. Intuitively, before each output, the system $ContErr(ErrPairs, \mathcal{S})$ takes an additional input (given by either the actions *err* or $\neg err$). This input controls whether an external error happens in this step or not. If an external error may happen, the system $ContErr(ErrPairs, \mathcal{S})$ may choose to output not only the actions in the original system $\mathcal{S}$, but also the possible replacement actions given by *ContErr*.

Formally, given $ErrPairs \subseteq \Sigma_{out} \times \Sigma_{out}$ and an alternating transition system $\mathcal{S} = \langle S, (S_{in}, S_{out}), \Sigma, (\Sigma_{in}, \Sigma_{out}), \Delta, s_\iota\rangle$, the system $ContErr(ErrPairs, \mathcal{S})$ is given by $\langle S_{in} \cup (S_{out} \times \{\perp, err, \neg err\}), (S_{in}, S_{out} \times \{\perp, err, \neg err\}), \Sigma \cup \{err, \neg err\}, (\Sigma_{in} \cup \{err, \neg err\}, \Sigma_{out}), \Delta', s'_\iota\rangle$ where:

- If $s_\iota \in S_{in}$, then $s'_\iota = s_\iota$; otherwise, $s_\iota = (s_\iota, \perp)$.
- We have the following transitions in Δ':
  - If $(s, \sigma_{in}, s') \in \Delta$ and $\sigma_{in} \in \Sigma_{in}$, then $(s, \sigma_{in}, (s', \perp)) \in \Sigma_{out}$.
  - We have $((s, \perp), \neg err, (s, \neg err)) \in \Delta$ and $((s, \perp), err, (s, err)) \in \Delta$ for all $s \in S_{out}$.
  - If $(s, \sigma_{out}, s') \in \Delta$ and $\sigma_{out} \in \Sigma_{out}$, then $((s, \neg err), \sigma, s') \in \Delta'$ and $((s, err), \sigma, s') \in \Delta'$.
  - If $(s, \sigma_{out}, s') \in \Delta$ and $\sigma_{out} \in \Sigma_{out}$, then $((s, err), \sigma'_{out}, s') \in \Delta'$ for all σ'_{out} such that $(\sigma_{out}, \sigma'_{out}) \in ErrPairs$.

Remark 3.9. *Note that the kind of errors we model here are "one-off" errors such as bit-flips and message losses. In this kind of errors, the influence of the error is restricted to one step of the computation. The current model cannot handle errors that are have some kind of temporal correlation (for example, burst errors where one error leads to loss of a series of messages).*

For a given set of actions Σ, the *robustness error model* is a one state deterministic weighted automaton $\langle \{s\}, \Sigma' \times \Sigma', \{s\} \times (\Sigma' \times \Sigma') \times \{s\}, s\rangle$ where $\Sigma' = \Sigma \cup \{err, \neg err\}$. Further, we have:

$$v((s, \sigma/\sigma', s)) = \begin{cases} 0 & \text{if } \sigma = \sigma' \\ 2 & \text{if } \sigma = err \wedge \sigma' = \neg err \\ 2 & \text{if } \sigma = \neg err \wedge \sigma' = err \\ \infty & \text{otherwise} \end{cases}$$

Intuitively, the robustness error penalizes every mismatch in simulation by a cost of infinity except replacement of *err* by $\neg err$ and vice-versa.

Robustness distance. Given systems $\mathcal{S}$ and $\mathcal{I}$, and the possible error pairs *ErrPairs*, the *robustness distance* $d_{\text{rob}}{}^{ErrPairs}(\mathcal{I}, \mathcal{S})$ from system $\mathcal{I}$ to system $\mathcal{S}$ is the value of the quantitative alternating simulation game $\mathcal{G}_{\text{rob}\mathcal{I},\mathcal{S},ErrPairs} = \mathcal{P}^{fair}{}_{ContErr(ErrPairs,\mathcal{I}),ContErr(\emptyset,\mathcal{S})}$.

Intuitively, one round of the game $\mathcal{G}_{\text{rob}}{}^{ErrPairs}_{\mathcal{I},\mathcal{S},}$ is played in the following steps (we describe the round for an output transition of $\mathcal{I}$ and $\mathcal{S}$ – the round for an input transition is same as in standard simulation games): (a) Player 1 chooses either the *err* or the $\neg$*err* transition of in the $\mathcal{S}$. In this step, it is always better for Player 1 to choose *err* – hence, we assume that the *err* transition is chosen. (b) Player 2 chooses to match the transition either with the *err* or the $\neg$*err* transition from $\mathcal{I}$. If Player 2 chooses $\neg$*err*, she gets a penalty for the mismatch. Intuitively, Player 2 is choosing whether an uncontrollable error may occur in the implementation $\mathcal{I}$ in the current step; (c) Player 1 chooses a transition on the implementation system. She is allowed to choose one of the erroneous transitions (i.e., where an action σ is replaced with σ' based on *ErrPairs*) if Player 2 chose *err* in the previous step. Otherwise, Player 1 has to choose an existing transition from $\mathcal{I}$. (d) Player 2 chooses a matching transition from $ContErr(\emptyset, \mathcal{S})$ (or equivalently, from $\mathcal{S}$) to simulate the implementation. The matching has to be exact – otherwise, Player 2 gets a penalty of ∞. In brief, Player 2 chooses whether an error may occur in the current step in the implementation or not. If she chooses to disallow errors in the current step, she gets a penalty. However, if she chooses to allow errors, she has to exactly match the implementation transition (along with possible errors) with a specification transition.

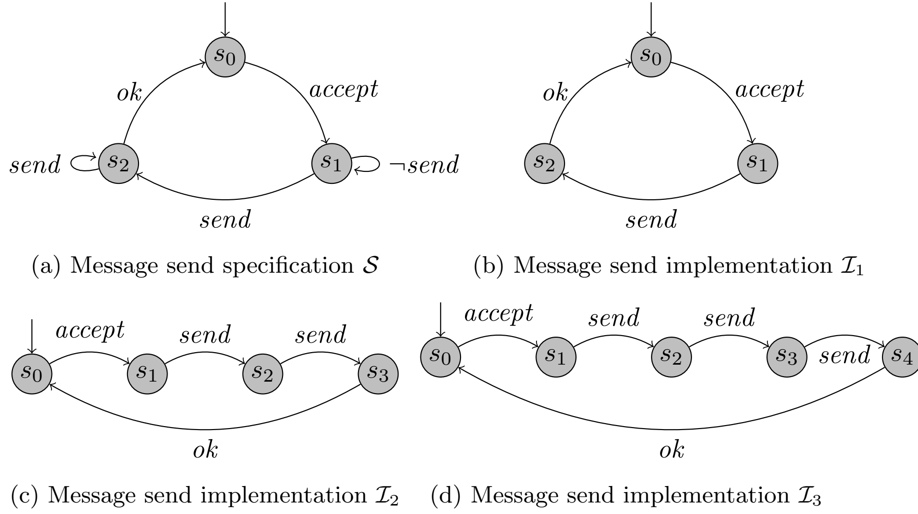
Player 2 tries to minimize the number of moves where it prohibits implementation errors (without destroying the simulation relation), whereas the implementation tries to maximize it. Intuitively, the positions where the specification cannot allow errors are the critical points for the implementation.

In Figures 3.8[Pg. 42] and 3.1[Pg. 33], in the robustness game for $\mathcal{S}_1$ and $\mathcal{S}_1$ with $ErrPairs = \{(a, b), (b, a)\}$, every position is critical. At each position, if an error is allowed, Player 1 can make the system output three b's in a row by using the error transition to return to state s_0 while outputting a b. The next two moves can be b's irrespective whether errors are allowed or not. This breaks the simulation. Now, consider $\mathcal{I}_1$. This system can be allowed to err every two out of three times without violating the specification. This shows that $\mathcal{I}_1$ is more robust than $\mathcal{S}_1$ for implementing $\mathcal{S}_1$. The list of distances is summarized in Table 3.1[Pg. 43].

Example 3.10. *Consider a simple example of sending a message across an unreliable channel. The specification is given in Figure 3.12a[Pg. 48] – in the ideal behaviour, the specification repeatedly accepts a message (symbol accept), ensures that it is sent successfully at least once (symbol send), and then returns ok. Now, the external error we are trying to model allows messages to be lost – therefore, a send action may be replaced by a $\neg$send action, i.e., we set $ErrPairs = \{(send, \neg send)\}$.*

Now, among the implementations $\mathcal{I}_1$, $\mathcal{I}_2$, and $\mathcal{I}_3$ in the Figure 3.12[Pg. 48], we have that $\mathcal{I}_1$ attempts to send each message only once, while $\mathcal{I}_2$ and $\mathcal{I}_3$ attempt to send each message twice and thrice respectively, in order to compensate for lost messages.

Intuitively, $\mathcal{I}_1$ is the least robust implementation, while $\mathcal{I}_3$ is the most robust



(a) Message send specification $\mathcal{S}$

(b) Message send implementation $\mathcal{I}_1$

(c) Message send implementation $\mathcal{I}_2$

(d) Message send implementation $\mathcal{I}_3$

Figure 3.12: Robustness of Message-Send protocols over an unreliable channel

as $\mathcal{I}_1$ does nothing to compensate for possible lost messages, while $\mathcal{I}_3$ does the most. The robustness distances also match this – we have that $d_{rob}(\mathcal{I}_1, \mathcal{S}) = \frac{1}{3}$, $d_{rob}(\mathcal{I}_2, \mathcal{S}) = \frac{1}{4}$, and $d_{rob}(\mathcal{I}_3, \mathcal{S}) = \frac{1}{5}$. Intuitively, to keep $\mathcal{I}_1$ from becoming erroneous, we ensure that the send is successful, i.e., prevent errors in 1 *step of its* 3 *step protocol. For $\mathcal{I}_2$ and $\mathcal{I}_3$, we need to ensure that at least one of the send is successful, i.e., prevent errors in* 1 *step of their* 4 *step and* 5 *step protocols, respectively.*

Component Robustness Distance. In many cases, a system is composed of two or more separate "components". An external error can not only cause the affected component to produce the wrong outputs, but also may make the other components behave badly. On the other hand, the components may be designed to work together robustly by compensating for errors in each others' behaviour. We extend the robustness distance to such systems with multiple components.

If $\mathcal{S}$ and $\mathcal{S}'$ are two transition systems, we define *asynchronous and synchronous composition* of the two systems, written as $\mathcal{S} \parallel \mathcal{S}'$ and $\mathcal{S} \times \mathcal{S}'$ respectively as follows:

- The state space is $\mathcal{S} \times \mathcal{S}'$;
- $((s_0, s'_0), \sigma, (s_1, s'_1))$ is a transition of $\mathcal{S} \parallel \mathcal{S}'$ iff (s_0, σ, s_1) is a transition of $\mathcal{S}$ and $s'_0 = s'_1$ or (s'_0, σ, s'_1) is a transition of $\mathcal{S}'$ and $s_0 = s_1$, and
- $((s_0, s'_0), \sigma, (s_1, s'_1))$ is a transition of $\mathcal{S} \times \mathcal{S}'$ iff (s_0, σ, s_1) is a transition of $\mathcal{S}$ and (s'_0, σ, s'_1) is a transition of $\mathcal{S}'$.

Given system $\mathcal{I}_1 \times \mathcal{I}_2$ having components $\mathcal{I}_1$ and $\mathcal{I}_2$, specification $\mathcal{S}$, and error pairs *ErrPairs*, the *component robustness distance* is defined to be the value of the game $\mathcal{Q}_{ContErr(ErrPairs, \mathcal{I}_1) \times \mathcal{I}_2, \mathcal{S}}$. Note that this definition is presented assuming that the first component is susceptible to external errors – however, it is easy to extend the definition to the other components as well as systems

with more than two components. We abuse notation and use the same notation as robustness distance, i.e., ${d_{\mathrm{rob}}}^{ErrPairs}(\mathcal{I}_1 \times \mathcal{I}_2, \mathcal{S})$ for component robustness distances.

Example 3.11. *Consider a message transmission protocol where in each round, the system receives either a request to either send a* 0 *or* 1*, indicated by* $r(0)$ *and* $r(1)$ *respectively. The system then executes a send protocol where the sender component transmits a series of* 0 *or* 1 *bits – indicated by* $t(0)$ *and* $t(1)$ *respectively. The receiver component then receives the transmitted bits and then outputs either* 0 *or* 1 *– indicated by* $o(0)$ *and* $o(1)$ *respectively.*

In Figure 3.13[Pg. 50], the specification $\mathcal{S}$ *states that when a* 0 *is requested, a* 0 *is output finally, and when a* 1 *is requested, a* 1 *is output finally. We measure robustness in the presence of bit-flips, i.e., a* $t(0)$ *may be interpreted as a* $t(1)$ *by the receiver or vice versa; a* $r(0)$ *may be interpreted as a* $r(1)$ *by the sender or vice versa; and a* $o(0)$ *may be replaced by a* $o(1)$ *by the receiver or vice versa. Formally, we set error pairs* $ErrPairs = \{(t(0), t(1)), (t(1), t(0)), (r(0), r(1)), (r(1), r(0)), (o(0), o(1)), (o(1), o(0))\}$.

The figure illustrates two pairs of sender and receiver components. Sender component $\mathcal{I}_1^{send}$ *transmits the bit which is requested for exactly once, while the sender component* $\mathcal{I}_2^{send}$ *transmits the bit which is requested for thrice. Similarly, the receiver component* $\mathcal{I}_1^{recv}$ *outputs the first bit it receives, while* $\mathcal{I}_2^{recv}$ *waits for three bits to be received and outputs the bit that is received two or more times.*

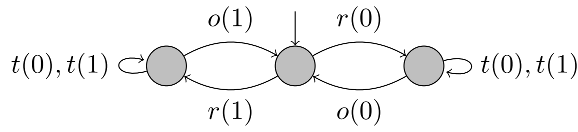
Intuitively, the system where the sender transmits three bits and receiver uses all the three bits for deciding the value to output (i.e., $\mathcal{I}_2^{send} \times \mathcal{I}_2^{recv}$ *is more robust than the system where only one bit is used (i.e.,* $\mathcal{I}_1^{recv} \times \mathcal{I}_1^{recv}$*).*

In the component robustness game, in $\mathcal{I}_1^{send} \times \mathcal{I}_1^{recv}$*, we need every action in the protocol to be correct, i.e., there are no external errors. Otherwise, the wrong bit may be output. On the other hand, in* $\mathcal{I}_2^{send} \times \mathcal{I}_2^{recv}$*, one error (during transmission) is tolerated. This is because the receiver waits for three bits and then outputs the majority bit. Hence, we have that* $d_{\mathrm{rob}}(\mathcal{I}_1^{send} \times \mathcal{I}_1^{recv}, \mathcal{S}) = \frac{3}{3}$ *and* $d_{\mathrm{rob}}(\mathcal{I}_2^{send} \times \mathcal{I}_2^{recv}, \mathcal{S}) = \frac{4}{5}$*. The component robustness distances reflect the intuition that the system which sends redundant messages is more robust (smaller robustness distance) than a system that does not.*

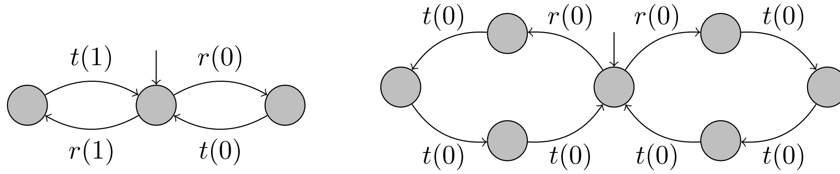
3.3.4 Computation of Simulation Distances

The computational complexity of computing the three distances defined here is the same as solving the value problem for the respective games. Firstly, note that given two systems with state spaces of size n and n' and transitions sets of size m and m' the product game graph has $O(nn')$ states and $O(nm' + mn)$ transitions for a fixed error model.

For systems without fairness conditions, the d_{cor}, d_{cov} and d_{rob} games are simple graph games with limit-average (*LimAvg*) or discounted-sum (*Disc*) objectives. The decision problem (deciding whether the value is greater than a given value) for these games is in NP ∩ co-NP [172], but no PTIME algorithm is known. However, for *LimAvg* objectives the existence of a pseudo-polynomial algorithm, i.e., polynomial for unary encoded weights, implies that the computation of the distances can be achieved in polynomial time. This is due to the fact that we use constant weights. Using the algorithm of [172], in the case

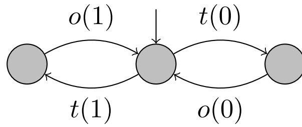


(a) Specification $\mathcal{S}$

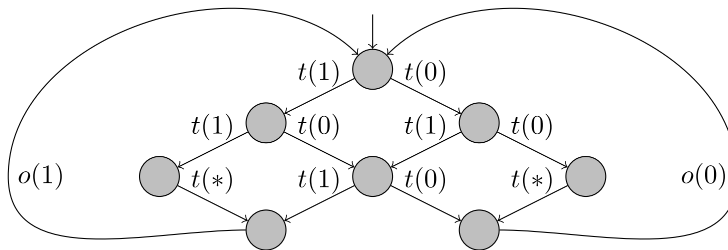


(b) Component send $\mathcal{I}_1^{send}$

(c) Component send $\mathcal{I}_2^{send}$



(d) Component receive $\mathcal{I}_1^{recv}$



(e) Component receive $\mathcal{I}_2^{recv}$

Figure 3.13: Message Sending Protocol

without fairness conditions d_{cor}, d_{cov} and d_{rob} distances can be computed in time $O((nn')^3 \cdot (nm' + mn))$. A variation of the algorithm in [172] gives a PTIME algorithm for the *Disc* objectives (given a fixed discounting factor).

For systems with Büchi (weak fairness) conditions, the corresponding games are graph games with *LimAvg* parity games, for which the decision problem is in NP ∩ co-NP. However, the use of constant weights and the fact that the implication of two Büchi conditions can be expressed as a parity condition with no more than 3 priorities leads to a polynomial algorithm. Using the algorithm presented in [50], we get a $O((nn')^3 \cdot (nm' + n'm))$ algorithm.

For systems with Streett (strong fairness) conditions, the corresponding games are graph games with *LimAvg* ω-regular conditions. For an ω-regular *LimAvg* game of N states, we can use the latest appearance records to convert into an equivalent parity game of $2^{O(N \log(N))}$ states and edges; and N priorities. The algorithm of [50] gives a $2^{O(N \log(N))}$ algorithm where $N = nn'$.

3.4 Properties of Simulation Distances

We present quantitative analogues of boolean properties of the simulation preorders. However, first we prove the following theorem that states that the correctness simulation distance is a refinement of the classical simulation relation.

Theorem 3.12. *If system $\mathcal{S}$ simulates $\mathcal{I}$, then we have that $d_{\text{cor}}{}^{\mathcal{M}}(\mathcal{I}, \mathcal{S}) = 0$ for all error models.*

Proof. We prove the result for labelled transition systems with no fairness constraints. The case with fairness constraints and the case of alternating transition systems is very similar.

We construct a Player 2 strategy ϕ_2 in $\mathcal{G}_{\text{cor}\mathcal{M},\mathcal{I},\mathcal{S}}$ such that $Val_2(\phi_2) = 0$ proving that $d_{\text{cor}}{}^{\mathcal{M}}(\mathcal{I}, \mathcal{S}) \leq 0$. Since, $d_{\text{cor}}{}^{\mathcal{M}}$ values are always non-negative, this completes the proof.

Suppose the simulation between $\mathcal{I}$ and $\mathcal{S}$ is witnessed by the simulation relation $\leq_{sim}$. The main idea behind the construction of ϕ_2 is to ensure that the $\mathcal{I}$ state and $\mathcal{S}$ state in the quantitative simulation game at the beginning of each round are in simulation. In other words, for every Player 1 state $(s^{\mathcal{I}}, \#, s^{\mathcal{S}}, s^{\mathcal{M}}, 1)$ visited in any play conforming to ϕ_2 ensures that $s^{\mathcal{I}} \leq_{sim} s^{\mathcal{S}}$. This holds in the initial state of the game as the initial states of $\mathcal{I}$ and $\mathcal{S}$ are in simulation.

Now, in the first step of a round, say Player 1 picks a testing transition $(s_0^{\mathcal{I}}, \sigma, s_1^{\mathcal{I}})$, i.e, moves from game state $(s_0^{\mathcal{I}}, \#, s_0^{\mathcal{S}}, s_0^{\mathcal{M}}, 1)$ to $(s_1^{\mathcal{I}}, \sigma, s_0^{\mathcal{S}}, s_0^{\mathcal{M}}, 2)$. In reply, the strategy ϕ_2 picks the matching transition $(s_0^{\mathcal{S}}, \sigma, s_1^{\mathcal{S}})$, i.e., moves to the state $(s_1^{\mathcal{I}}, \#, s_1^{\mathcal{S}}, s_1^{\mathcal{M}}, 1)$ such that $s_1^{\mathcal{I}} \leq_{sim} s_1^{\mathcal{S}}$. Such a state $s_1^{\mathcal{S}}$ and the transition $(s_0^{\mathcal{S}}, \sigma, s_1^{\mathcal{S}})$ are guaranteed to exist due to the definition of the simulation relation.

In any such play conforming to ϕ_2, as the matching of the transitions in the quantitative simulation game is exact, the error model has to assign cost 0 to the play. Hence, we have $Val_2(\phi_2) = 0$. □

3.4.1 Directed Metrics

Classical simulation relations satisfy the reflexivity and transitivity property which makes them preorders. In an analogous way, we show that the correct-

ness and coverage distances satisfy the quantitative reflexivity and the triangle inequality properties for a certain class error models defined below. This makes them directed metrics [71]. Further, we show that the robustness distance satisfies the triangle inequality, but not quantitative reflexivity.

Transitive Error Models. It can be seen easily that the simulation distances are not directed metrics for all error models as the triangle inequality fails to hold. Instead, here we provide a necessary and sufficient condition on the error models, for the triangle inequality to hold.

An error model $\mathcal{M}$ is *transitive* if the following holds: For every triple of infinite lasso (i.e., ultimately periodic) words generated by the error model M of the form $\alpha = \frac{a_0}{b_0}\frac{a_1}{b_1}\ldots$, $\beta = \frac{b_0}{c_0}\frac{b_1}{c_1}\ldots$ and $\gamma = \frac{a_0}{c_0}\frac{a_1}{c_1}\ldots$, the $\mathcal{M}(\alpha) + \mathcal{M}(\beta) \geq \mathcal{M}(\gamma)$.

For a non-transitive error model, the three systems whose only behavior outputs the lasso-words which witness the non-transitivity violate the triangle inequality. All the error models introduced in Section 3.2.3[Pg. 38] (see Figure 3.7[Pg. 39]) are transitive.

The transitiveness of an error model can be checked in polynomial time.

Proposition 3.13. *It is decidable in polynomial time whether an error model* $\mathcal{M} = \langle S^{\mathcal{M}}, \Sigma_{out} \times \Sigma_{out}, \Delta^{\mathcal{M}}, s_\iota^{\mathcal{M}}, v\rangle$ *is transitive.*

Proof. The result follows by constructing the product $\mathcal{M} \times \mathcal{M} \times \mathcal{M}$ with:

- State-space $S^{\mathcal{M}} \times S^{\mathcal{M}} \times S^{\mathcal{M}}$; and
- A transition between (s_0, s_1, s_2) to (s_0', s_1', s_2') on action $\frac{\sigma_1}{\sigma_3}$ having weight $w_1 + w_2 - w_3$ if and only if there exists transitions $(s_0, \sigma_1/\sigma_2, s_0')$, $(s_1, \sigma_2/\sigma_3, s_1')$ and $(s_3, \sigma_1/\sigma_3, s_3')$ having weights w_1, w_2 and w_3 respectively.

The model M is transitive iff there is no negative cycle in the product (checkable in polynomial time). □

Now, we are ready to prove the directed metric properties of the simulation distances.

Theorem 3.14. *Given a set of actions* Σ*, the function* ${d_{cor}}^{\mathcal{M}}$ *is a directed metric on the space of systems over* Σ *for all transitive error models* $\mathcal{M}$*. In other words,*

- *for all systems* $\mathcal{S}$*, we have that reflexivity holds for the correctness distance, i.e.,* ${d_{cor}}^{\mathcal{M}}(\mathcal{S}, \mathcal{S}) = 0$*; and*
- *for all systems* $\mathcal{S}_1$*,* $\mathcal{S}_2$*, and* $\mathcal{S}_3$ *over* Σ *we have the triangle inequality for the correctness distance, i.e.,* ${d_{cor}}^{\mathcal{M}}(\mathcal{S}_1, \mathcal{S}_3) \leq {d_{cor}}^{\mathcal{M}}(\mathcal{S}_1, \mathcal{S}_2) + {d_{cor}}^{\mathcal{M}}(\mathcal{S}_2, \mathcal{S}_3)$

Proof. We will prove the result for alternating systems with fairness conditions with limit-average objectives. The case without fairness conditions and the case of discounted-sum objectives are analogous.

Reflexivity. As we have that $\mathcal{S}$ simulates $\mathcal{S}$, we have from Theorem 3.12[Pg. 51] that ${d_{cor}}^{\mathcal{M}}(\mathcal{S}, \mathcal{S}) = 0$.

Triangle Inequality. Consider any $\epsilon > 0$. Let ϕ_2^2 and ϕ_2^3 be $\frac{\epsilon}{4}$-optimal finite memory strategies for Player 2 in the games $\mathcal{G}_{\text{cor}\mathcal{S}_1,\mathcal{S}_2}$ and $\mathcal{G}_{\text{cor}\mathcal{S}_2,\mathcal{S}_3}$ respectively. The existence of such finite-memory $\frac{\epsilon}{4}$-optimal strategies are guaranteed by standard results on limit-average parity games (See Chapter 2[Pg. 16]). Using

ϕ_2^2 and ϕ_2^3, we construct a finite-memory strategy ϕ_2^* for Player 2 in $\mathcal{G}_{\mathrm{cor}\mathcal{S}_1,\mathcal{S}_3}$. Let $S^{1,2}$ and $S^{2,3}$ be the state-spaces of the game graphs of $\mathcal{G}_{\mathrm{cor}\mathcal{S}_1,\mathcal{S}_2}$ and $\mathcal{G}_{\mathrm{cor}\mathcal{S}_2,\mathcal{S}_3}$, and let M_2 and M_3 are the memories of ϕ_2^2 and ϕ_2^3 respectively. The memory of ϕ_2^* will be $S^{1,2} \times M_2 \times S^{2,3} \times M_3$.

The main idea behind the construction of ϕ_2^* is as follows: during each round of $\mathcal{G}_{\mathrm{cor}\mathcal{S}_1,\mathcal{S}_3}$, the strategy simulates rounds of $\mathcal{G}_{\mathrm{cor}\mathcal{S}_1,\mathcal{S}_2}$ and $\mathcal{G}_{\mathrm{cor}\mathcal{S}_2,\mathcal{S}_3}$ in its memory.

Formally, let the state of the game $\mathcal{G}_{\mathrm{cor}\mathcal{S}_1,\mathcal{S}_3}$ at the beginning of a round be $(s_1, \#s_3, s_{1,3}^{\mathcal{M}}, 1)$. We will construct ϕ_2^* such that the memory of the strategy when arriving at this state will be $((s_1, \#, s_2, s_{1,2}^{\mathcal{M}}), m_2, (s_2, \#, s_3, s_{2,3}^{\mathcal{M}}), m_3)$ for some states s_2, m_2, and m_3 of the $\mathcal{S}_2$, M_2 and M_3 respectively and some states of the error model $s_{1,2}^{\mathcal{M}}$ and $s_{2,3}^{\mathcal{M}}$.

Now, the round proceeds as follows. Suppose Player 1 chooses the $\mathcal{S}_1$ transition (s_1, σ_1, s_1') in $\mathcal{G}_{\mathrm{cor}\mathcal{S}_1,\mathcal{S}_3}$, i.e., moves to $(s_1', \sigma_1, s_3, s_{1,3}^{\mathcal{M}}, 2)$. Then, the matching transition chosen by ϕ_2^* is (s_3, σ_3, s_3'), i.e., moving to $(s_1', \#, s_3', s_{1,3}^{\mathcal{M}'}, 1)$ where we have the following:

- $\mathcal{G}_{\mathrm{cor}\mathcal{S}_1,\mathcal{S}_2}$ round: Suppose in $\mathcal{G}_{\mathrm{cor}\mathcal{S}_1,\mathcal{S}_2}$, the current state is $(s_1, \#, s_2, s_{1,2}^{\mathcal{M}}, 1)$ and the current state of the memory of Player 2 strategy ϕ_2^2 is m_2. Let Player 1 choose the transition $\mathcal{S}_1$ transition (s_1, σ_1, s_1') and Player 2 choose the transition (s_2, σ_2, s_2') according to strategy ϕ_2^2, and let the memory of ϕ_2^2 be updated to m_2'.
- $\mathcal{G}_{\mathrm{cor}\mathcal{S}_2,\mathcal{S}_3}$ round: Suppose in $\mathcal{G}_{\mathrm{cor}\mathcal{S}_2,\mathcal{S}_3}$, the current state is $(s_2, \#, s_3, s_{2,3}^{\mathcal{M}}, 1)$ and the current state of the memory of Player 2 strategy ϕ_2^3 is m_3. Let Player 1 choose the transition $\mathcal{S}_2$ transition (s_2, σ_2, s_2') and Player 2 choose the transition (s_3, σ_3, s_3') according to strategy ϕ_2^3, and let the memory of ϕ_2^3 be updated to m_3'.
- $\mathcal{M}$ transition: $s_{1,3}^{\mathcal{M}'}$ is the unique $\mathcal{M}$ state such that $(s_{1,3}^{\mathcal{M}}, \sigma_1/\sigma_3, s_{1,3}^{\mathcal{M}'})$ is a $\mathcal{M}$ transition.

Further, the memory of ϕ_2^* is updated to $((s_1', \#, s_2', s_{1,2}^{\mathcal{M}'}, 1), m_2', (s_2', \#, s_3', s_{2,3}^{\mathcal{M}'}, 1), m_3')$.

Now, fix an $\frac{\epsilon}{2}$-optimal Player 1 strategy ϕ_1^* in $\mathcal{G}_{\mathrm{cor}\mathcal{S}_1,\mathcal{S}_3}$ – again, it can be assumed that this is a finite-memory strategy. Now, suppose play $\rho_{1,3} = \mathit{Outcomes}(\phi_1^*, \phi_2^*)$ in $\mathcal{G}_{\mathrm{cor}\mathcal{S}_1,\mathcal{S}_3}$. From the above construction, from each round of the play $\rho_{1,3}$ and the corresponding memory of ϕ_2^*, we can extract transitions of $\mathcal{G}_{\mathrm{cor}\mathcal{S}_1,\mathcal{S}_2}$ and $\mathcal{G}_{\mathrm{cor}\mathcal{S}_2,\mathcal{S}_3}$ to construct play $\rho_{1,2}$ and $\rho_{2,3}$ of $\mathcal{G}_{\mathrm{cor}\mathcal{S}_1,\mathcal{S}_2}$ and $\mathcal{G}_{\mathrm{cor}\mathcal{S}_2,\mathcal{S}_3}$, respectively. Further, $\rho_{1,2}$ and $\rho_{2,3}$ conform to the $\frac{\epsilon}{4}$-optimal Player 2 strategies ϕ_2^2 and ϕ_2^3, respectively. Therefore, we have

$$\begin{aligned} \mathit{Val}(\rho_{1,2}) + \mathit{Val}(\rho_{2,3}) &\leq \mathit{Val}(\phi_2^2) + \mathit{Val}(\phi_2^3) \\ &\leq d_{\mathrm{cor}}{}^{\mathcal{M}}(\mathcal{S}_1, \mathcal{S}_2) + \frac{\epsilon}{4} + d_{\mathrm{cor}}{}^{\mathcal{M}}(\mathcal{S}_2, \mathcal{S}_3) + \frac{\epsilon}{4} \\ &\leq d_{\mathrm{cor}}{}^{\mathcal{M}}(\mathcal{S}_1, \mathcal{S}_2) d_{\mathrm{cor}}{}^{\mathcal{M}}(\mathcal{S}_2, \mathcal{S}_3) + \frac{\epsilon}{2} \end{aligned}$$

Further, by the construction of ϕ_2^*, we have that the values of plays $\rho_{1,3}$, $\rho_{1,2}$ and $\rho_{2,3}$ are equal to values of words of the form $(\sigma_1^0/\sigma_3^0)(\sigma_1^1/\sigma_3^1)\ldots$, $(\sigma_1^0/\sigma_2^0)(\sigma_1^1/\sigma_2^1)\ldots$, and $(\sigma_2^0/\sigma_3^0)(\sigma_2^1/\sigma_3^1)\ldots$, respectively. Therefore, by the transitivity of the error model $\mathcal{M}$, we have

$$\mathit{Val}(\rho_{1,3}) \leq \mathit{Val}(\rho_{1,2}) + \mathit{Val}(\rho_{2,3})$$

Combining the above equations, we have that

$$\begin{aligned}
{d_{\text{cor}}}^{\mathcal{M}}(\mathcal{S}_1, \mathcal{S}_3) &= \sup_{\phi_1 \in \Phi_1} \inf_{\phi_2 \in \Phi_2} \mathit{Val}(\mathit{Outcomes}(\phi_1, \phi_2)) \\
&\leq \inf_{\phi_2 \in \Phi_2} \mathit{Val}(\mathit{Outcomes}(\phi_1^*, \phi_2)) + \frac{\epsilon}{2} \qquad \phi_1^* \text{ is } \tfrac{\epsilon}{2}\text{-optimal} \\
&\leq \mathit{Val}(\mathit{Outcomes}(\phi_1^*, \phi_2^*)) + \frac{\epsilon}{2} \\
&= \mathit{Val}(\rho_{1,3}) \\
&= \mathit{Val}(\rho_{1,2}) + \mathit{Val}(\rho_{2,3}) \\
&\leq {d_{\text{cor}}}^{\mathcal{M}}(\mathcal{S}_1, \mathcal{S}_2) + {d_{\text{cor}}}^{\mathcal{M}}(\mathcal{S}_2, \mathcal{S}_3) + \epsilon
\end{aligned}$$

As the choice of ϵ was arbitrary, we have

$${d_{\text{cor}}}^{\mathcal{M}}(\mathcal{S}_1, \mathcal{S}_3) \leq {d_{\text{cor}}}^{\mathcal{M}}(\mathcal{S}_1, \mathcal{S}_2) + {d_{\text{cor}}}^{\mathcal{M}}(\mathcal{S}_2, \mathcal{S}_3)$$

This gives us the triangle inequality in the case without fairness conditions.

In the case with fairness conditions, the same arguments apply as long as both $\mathcal{S}_1$ and $\mathcal{S}_3$ perform fair computations in the play $\rho_{1,3}$. In the case that $\mathcal{S}_1$ computation is fair and $\mathcal{S}_3$ computation is not fair, the value of the play will be ∞. However, by construction the value of either $\rho_{1,2}$ or $\rho_{2,3}$ will also be ∞ and hence the inequality still holds. □

Theorem 3.15. *${d_{\text{cov}}}^{\mathcal{M}}$ is a directed metric when $\mathcal{M}$ is the standard error model for both limit-average and discounted-sum objective, i.e.,*

- *for all systems $\mathcal{S}$, we have that ${d_{\text{cov}}}^{\mathcal{M}}(\mathcal{S}, \mathcal{S}) = 0$; and*
- *for all systems $\mathcal{S}_1$, $\mathcal{S}_2$, $\mathcal{S}_3$, we have that ${d_{\text{cov}}}^{\mathcal{M}}(\mathcal{S}_1, \mathcal{S}_3) \leq {d_{\text{cov}}}^{\mathcal{M}}(\mathcal{S}_1, \mathcal{S}_2) + {d_{\text{cov}}}^{\mathcal{M}}(\mathcal{S}_2, \mathcal{S}_3)$.*

Proof. The proof of this proposition follows from the fact that for any two systems $\mathcal{S}_1$ and $\mathcal{S}_2$, we have that ${d_{\text{cov}}}^{f}(\mathcal{S}_1, \mathcal{S}_2) = {d_{\text{cor}}}^{f}(\mathcal{S}_2, \mathcal{S}_1)$ and Theorem 3.14[Pg. 52]. □

The robustness distance satisfies the triangle inequality, but not the quantitative reflexivity. For example, the system $\mathcal{S}_1$ in Figure 3.1[Pg. 33] is a witness system that violates reflexivity. In fact, for *LimAvg* objectives and any rational value $v \in [0, 1]$, it is easy to construct a system $\mathcal{S}_v$ such that $d_{\text{rob}}(\mathcal{S}_v, \mathcal{S}_v) = v$.

Theorem 3.16. *${d_{\text{rob}}}^{\mathit{ErrPairs}}$ conforms to the triangle inequality for all ErrPairs – for all systems $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$, we have that ${d_{\text{rob}}}^{\mathit{ErrPairs}}(\mathcal{S}_1, \mathcal{S}_3) \leq {d_{\text{rob}}}^{\mathit{ErrPairs}}(\mathcal{S}_1, \mathcal{S}_2) + {d_{\text{rob}}}^{\mathit{ErrPairs}}(\mathcal{S}_2, \mathcal{S}_3)$.*

Proof. Our proof mainly relies on the fact that ${d_{\text{rob}}}^{\mathit{ErrPairs}}(\mathcal{S}, \mathcal{S}') = {d_{\text{cor}}}^{\mathcal{M}}(\mathit{ContErr}(\mathit{ErrPairs}, \mathcal{S}), \mathit{ContErr}(\emptyset, \mathcal{S}'))$ for an appropriate error model. Therefore, by repeated application of the triangle inequality for d_{cor} (Theo-

rem 3.14[Pg. 52]) and the above fact, we can write:

$$
\begin{aligned}
{d_{\mathrm{rob}}}^{ErrPairs}(\mathcal{S}_1,\mathcal{S}_3) &= {d_{\mathrm{cor}}}^{\mathcal{M}}(ContErr(ErrPairs,\mathcal{S}_1), ContErr(\emptyset,\mathcal{S}_3)) \\
&\leq {d_{\mathrm{cor}}}^{\mathcal{M}}(ContErr(ErrPairs,\mathcal{S}_1), ContErr(\emptyset,\mathcal{S}_2)) \\
&\qquad + {d_{\mathrm{cor}}}^{\mathcal{M}}(ContErr(\emptyset,\mathcal{S}_2), ContErr(\emptyset,\mathcal{S}_3)) \\
&\leq {d_{\mathrm{cor}}}^{\mathcal{M}}(ContErr(ErrPairs,\mathcal{S}_1), ContErr(\emptyset,\mathcal{S}_2)) \\
&\qquad + {d_{\mathrm{cor}}}^{\mathcal{M}}(ContErr(\emptyset,\mathcal{S}_2), ContErr(ErrPairs,\mathcal{S}_2)) \\
&\qquad + {d_{\mathrm{cor}}}^{\mathcal{M}}(ContErr(ErrPairs,\mathcal{S}_2), ContErr(\emptyset,\mathcal{S}_3)) \\
&= {d_{\mathrm{rob}}}^{ErrPairs}(\mathcal{S}_1,\mathcal{S}_2) \\
&\qquad + {d_{\mathrm{cor}}}^{\mathcal{M}}(ContErr(\emptyset,\mathcal{S}_2), ContErr(ErrPairs,\mathcal{S}_2)) \\
&\qquad + {d_{\mathrm{rob}}}^{ErrPairs}(\mathcal{S}_2,\mathcal{S}_3)
\end{aligned}
$$

However, it can be easily seen that $ContErr(ErrPairs,\mathcal{S}_2)$ simulates $ContErr(\emptyset,\mathcal{S}_2)$. In fact, the identity relation of the set of states of each of these systems is a witness simulation relation. Therefore, by Theorem 3.12[Pg. 51] we have that ${d_{\mathrm{cor}}}^{\mathcal{M}}(ContErr(\emptyset,\mathcal{S}_2), ContErr(ErrPairs,\mathcal{S}_2)) = 0$.

Substituting this into the above equation, we get the required result. □

3.4.2 Compositionality

In the qualitative case, compositionality theorems help analyze large systems by decomposing them into smaller components. For example, if $\mathcal{S}_1$ simulates $\mathcal{I}_1$ and $\mathcal{S}_2$ simulates $\mathcal{I}_2$, we have that the composition of $\mathcal{S}_1$ and $\mathcal{S}_2$ simulates the composition of $\mathcal{I}_1$ and $\mathcal{I}_2$. We show that in the quantitative case, the distance between the composed systems is bounded by the sum of the distances between individual systems. Recall the definition of asynchronous composition of systems from Section 3.3.3[Pg. 45]. We only consider the case of fully asynchronous composition here. The general case of partially synchronous composition is harder to deal with.

The following theorems show that the simulation distances between whole systems is no more than the sum of the distances between the individual components for the standard error model. The theorems can be extended to any error model which has a single state. However, it can be shown that compositionality does not hold for error models with more than one state for asynchronous systems. For example, in the Figure 3.14[Pg. 56] and the delayed response error model $\mathcal{M}$ from Figure 3.7a[Pg. 39], the distance ${d_{\mathrm{cor}}}^{\mathcal{M}}(\mathcal{I}_1,\mathcal{S}_1) = 1/2$ (as the system spends 2 out of 4 steps waiting for the missing grant in the worst case) and ${d_{\mathrm{cor}}}^{\mathcal{M}}(\mathcal{I}_2,\mathcal{S}_2) = 0$. However, we have that ${d_{\mathrm{cor}}}^{\mathcal{M}}(\mathcal{I}_1 \parallel \mathcal{I}_2,\mathcal{S}_1 \parallel \mathcal{S}_2) = 1$. This is due to the fact that in the composed system $\mathcal{I}_1 \parallel \mathcal{I}_2$ (unlike the system $\mathcal{I}_1$), the request and the matching grant in the first component can be separated by an unbounded number of steps of the second component.

Theorem 3.17. *The correctness, coverage, and robustness distances satisfy the following property for limit-average simulation distances and the standard error model $\mathcal{M}$:*

$$\forall \mathcal{S}_1,\mathcal{S}_2,\mathcal{I}_1,\mathcal{I}_2 : d^{\mathcal{M}}(\mathcal{S}_1 \parallel \mathcal{S}_2,\mathcal{I}_1 \parallel \mathcal{I}_2) \leq \alpha \cdot d^{\mathcal{M}}(\mathcal{S}_1,\mathcal{I}_1) + (1-\alpha) \cdot d^{\mathcal{M}}(\mathcal{S}_2,\mathcal{I}_2)$$

where α is the fraction of times $\mathcal{S}_1$ is scheduled in $\mathcal{S}_1 \parallel \mathcal{S}_2$ in the long run, assuming that the fraction has a limit in the long run.

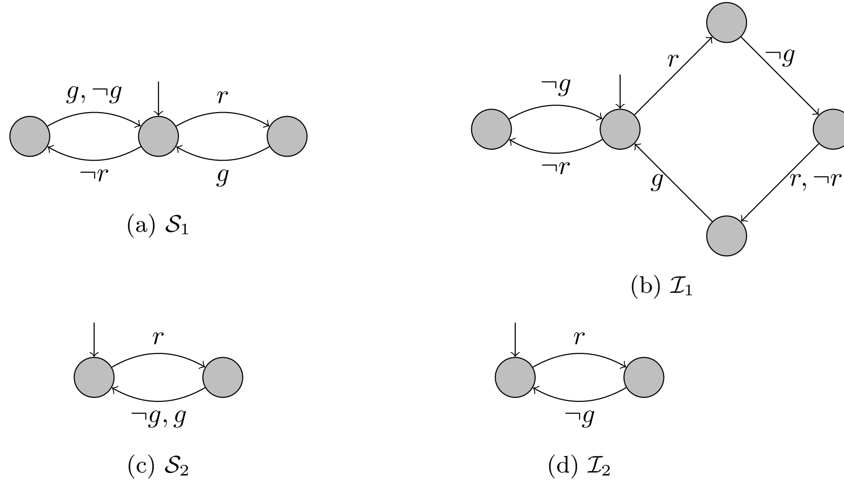


Figure 3.14: Non-compositionality for multi-state error models

Proof. The proof works for all cases by constructing a Player 2 strategy τ^* from the $\frac{\epsilon}{2}$-optimal strategies τ_1 and τ_2 in the games for computing $d(\mathcal{S}_1, \mathcal{I}_1)$ and $d(\mathcal{S}_2, \mathcal{I}_2)$ respectively. Let τ_1 and τ_2 be $\frac{\epsilon}{2}$-optimal strategies for Player 2 in the $\mathcal{G}_{\text{cor}\mathcal{S}_1,\mathcal{I}_1}$ and $\mathcal{G}_{\text{cor}\mathcal{S}_2,\mathcal{I}_2}$, with memory M_1 and M_2 respectively. We define a strategy τ^* for Player 2 in $\mathcal{G}_{\text{cor}\mathcal{S}_1\times\mathcal{S}_2,\mathcal{I}_1\times\mathcal{I}_2}$ with memory $M_1 \times M_2$. τ^* works by playing τ_1 and τ_2 component-wise.

Correctness. For the correctness game, we define τ^* as follows: If Player 1 moves from $((s_1, s_2), \#, (t_1, t_2), 1)$ to $((s_1', s_2), \sigma, (t_1, t_2), 2)$ according to the transition (s_1, σ, s_2) of the first component, and τ^* has the memory (m_1, m_2), it responds by playing the τ_1 strategy in the first component, i.e. if from the game position (s_1', σ, t_1) and memory m_1, τ_1 moves to $(s_1', \#, t_1')$ and updates memory to m_1', τ^* chooses to move to $((s_1', s_2), \#, (t_1', t_2), 1)$ and updates its memory to (m_1', m_2). The response to a Player 1 move in the second component of the system is similar.

We can prove that τ^* is a witness to the required inequality as follows: let ρ be any play conformant to τ_*. Let $I_1 \subseteq \mathbb{N}$ be the indices where the move is in the first component and let $I_2 = \mathbb{N} \backslash I_1$. Now, let ρ_i be the $\mathcal{G}_{\text{cor}\mathcal{S}_i,\mathcal{I}_i}$ play obtained from ρ by taking on the positions in $\mathcal{I}_i$ and projecting it into component i. By construction, we have ρ_i conformant to τ_i. Hence, we get for the limit-average

case:

$$
\begin{aligned}
LimAvg(\rho) &= \liminf_{n\to\infty} \frac{1}{n}\sum_{i=0}^{n} v(\rho^i) = \liminf_{n\to\infty} \frac{1}{n}\left(\sum_{i\in\mathcal{I}_1}^{i\le n} v(\rho_1^i) + \sum_{i\in\mathcal{I}_2}^{i\le n} v(\rho_2^i)\right) \\
&= \lim_{n\to\infty} \frac{1}{n}\left(\sum_{i\in\mathcal{I}_1}^{i\le n} v(\rho_1^i) + \sum_{i\in\mathcal{I}_2}^{i\le n} v(\rho_2^i)\right) \\
&= \lim_{n\to\infty} \frac{n_1}{n}\cdot\left(\frac{1}{n_1}\cdot\sum_{i\in\mathcal{I}_1}^{i\le n} v(\rho_1^i)\right) + \frac{n_2}{n}\cdot\left(\frac{1}{n_2}.\sum_{i\in\mathcal{I}_2}^{i\le n} v(\rho_2^i)\right) \\
&\qquad \text{where } n_i = |\{k \mid k\in\mathcal{I}_i \wedge k\le n\}| \\
&= \alpha\cdot\left(\lim_{n_1\to\infty}\frac{1}{n_1}\cdot\sum_{i\in\mathcal{I}_1}^{i\le n} v(\rho_1^i)\right) + (1-\alpha)\cdot\left(\lim_{n_2\to\infty}\frac{1}{n_2}\cdot\sum_{i\in\mathcal{I}_2}^{i\le n} v(\rho_2^i)\right) \\
&\qquad \text{as } \lim_{n\to\infty}\tfrac{n_1}{n} = \alpha \text{ and } \lim_{n\to\infty}\tfrac{n_2}{n} = 1-\alpha \\
&\le \alpha\cdot(d(\mathcal{S}_1,\mathcal{I}_1) + \frac{\epsilon}{2}) + (1-\alpha).(d(\mathcal{S}_2,\mathcal{I}_2) + \frac{\epsilon}{2}) \\
&\qquad \text{as } \rho_1 \text{ conforms to } \tau_1 \text{ and } \rho_2 \text{ conforms to } \tau_2 \\
&= \alpha\cdot d(\mathcal{S}_1,\mathcal{I}_1) + (1-\alpha)\cdot d(\mathcal{S}_2,\mathcal{I}_2) + \frac{\epsilon}{2}
\end{aligned}
$$

Hence, τ^* is a witness strategy that shows the required inequality.

Coverage. The proof for the coverage distance follows as the coverage distance is the dual of the correctness distance.

Robustness. For the robustness game, the proof follows by rewriting the robustness distance from $\mathcal{S}$ to $\mathcal{I}$ in terms of the correctness distance from $\mathit{ContErr}(\emptyset, \mathcal{S})$ to $\mathit{ContErr}(\mathit{ErrPairs}, \mathcal{I})$.

□

3.4.3 Abstraction

In the boolean case, properties of systems can be studied by studying the properties of over-approximations and under-approximations. In an analogous way, we prove that the distances between two systems is bounded from above and below by distances between abstractions of the two systems. We first define over-approximations and under-approximations of systems.

Given a transition system $\mathcal{S} = \langle S, \Sigma, \Delta, s_\iota\rangle$, a system $\mathcal{S}^\exists = \langle S^\exists, \Sigma, \Delta^\exists, s_\iota^\exists\rangle$ is a *existential abstraction* of $\mathcal{S}$ if there exists an equivalence relation $\equiv$ on S such that

1. $S^\exists$ is the set of equivalence classes of $\equiv$;
2. $s_\iota^\exists$ is the equivalence class of $\equiv$ containing s_ι; and
3. $(s_0^\exists, \sigma, s_1^\exists) \in \Delta^\exists$ if there exist s_0 and s_1 such that $(s_0, \sigma, s_1) \in \Delta$ and $s_0^\exists$ and $s_1^\exists$ are equivalence classes of s_0 and s_1 respectively.

Similarly, given a transition system $\mathcal{S} = \langle S, \Sigma, \Delta, s_\iota\rangle$, a system $\mathcal{S}^\forall = \langle S^\forall, \Sigma, \Delta^\forall, s_\iota^\forall\rangle$ is a *existential abstraction* of $\mathcal{S}$ if there exists an equivalence relation $\equiv$ on S such that

1. $S^\forall$ is the set of equivalence classes of $\equiv$;

2. $s_\iota^\forall$ is the equivalence class of $\equiv$ containing s_ι; and
3. $(s_0^\forall, \sigma, s_1^\forall) \in \Delta^\forall$ if for all s_0 and s_1 such $s_0^\forall$ and $s_1^\forall$ are equivalence classes of s_0 and s_1 respectively, we have that $(s_0, \sigma, s_1) \in \Delta$.

Theorem 3.18. *Let $\mathcal{S}_2$ and $\mathcal{S}_1$ be systems. Let $\mathcal{S}_2^\exists$ and $\mathcal{S}_1^\exists$ be existential abstractions, and $\mathcal{S}_2^\forall$ and $\mathcal{S}_1^\forall$ be universal abstractions of $\mathcal{S}_2$ and $\mathcal{S}_1$, respectively. The correctness, coverage, and robustness distances satisfy the three following properties:*

- $d_{cor}{}^f(\mathcal{S}_1^\forall, \mathcal{S}_2^\exists) \leq d_{cor}{}^f(\mathcal{S}_1, \mathcal{S}_2) \leq d_{cor}{}^f(\mathcal{S}_1^\exists, \mathcal{S}_2^\forall)$
- $d_{cov}{}^f(\mathcal{S}_1^\exists, \mathcal{S}_2^\forall) \leq d_{cov}{}^f(\mathcal{S}_1, \mathcal{S}_2) \leq d_{cov}{}^f(\mathcal{S}_1^\forall, \mathcal{S}_2^\exists)$
- $d_{rob}{}^f(\mathcal{S}_1^\forall, \mathcal{S}_2^\exists) \leq d_{rob}{}^f(\mathcal{S}_1, \mathcal{S}_2) \leq d_{rob}{}^f(\mathcal{S}_1^\exists, \mathcal{S}_2^\forall)$

Proof. Our proof relies on the fact that $\mathcal{S}^\forall$ is simulated by $\mathcal{S}$ and $\mathcal{S}$ is simulated by $\mathcal{S}^\exists$. This is an easy to prove using the witness simulation relation with relates every state to the corresponding equivalence class, i.e., $s \leq_{sim} s^\exists$ and $s^\forall \leq_{sim} s$ where $s^\exists$ and $s^\forall$ are the equivalence classes containing s for the corresponding equivalence relations.

- Now, we have the following for the correctness distance by repeated application of the triangle inequality (Theorem 3.14[Pg. 52]):

$$\begin{aligned} d_{\text{cor}}(\mathcal{S}_1^\forall, \mathcal{S}_2^\exists) &\leq d_{\text{cor}}(\mathcal{S}_1^\forall, \mathcal{S}_1) + d_{\text{cor}}(\mathcal{S}_1, \mathcal{S}_2^\exists) \\ &\leq d_{\text{cor}}(\mathcal{S}_1^\forall, \mathcal{S}_1) + d_{\text{cor}}(\mathcal{S}_1, \mathcal{S}_2) + d_{\text{cor}}(\mathcal{S}_2, \mathcal{S}_2^\exists) \end{aligned}$$

  However, by the fact that $\mathcal{S}_1^\forall$ is simulated by $\mathcal{S}_1$ and $\mathcal{S}_1$ is simulated by $\mathcal{S}_1^\exists$, we have that $d_{\text{cor}}(\mathcal{S}_1^\forall, \mathcal{S}_1) = 0$ and $d_{\text{cor}}(\mathcal{S}_2, \mathcal{S}_2^\forall)$. Substituting in the above equation, we get

$$d_{\text{cor}}(\mathcal{S}_1^\forall, \mathcal{S}_2^\exists) \leq d_{\text{cor}}(\mathcal{S}_1, \mathcal{S}_2)$$

  Dually, we have the following inequality:

$$\begin{aligned} d_{\text{cor}}(\mathcal{S}_1, \mathcal{S}_2) &\leq d_{\text{cor}}(\mathcal{S}_1, \mathcal{S}_1^\exists) + d_{\text{cor}}(\mathcal{S}_1^\exists, \mathcal{S}_2) \\ &\leq d_{\text{cor}}(\mathcal{S}_1, \mathcal{S}_1^\exists) + d_{\text{cor}}(\mathcal{S}_1^\exists, \mathcal{S}_2^\forall) + d_{\text{cor}}(\mathcal{S}_2^\forall, \mathcal{S}_2) \\ &\leq d_{\text{cor}}(\mathcal{S}_1^\exists, \mathcal{S}_2^\forall) \end{aligned}$$

- The result for the coverage distance follows as the coverage distance is the dual of the correctness distance.

- For the robustness distance, we have:

$$\begin{aligned} d_{\text{rob}}(\mathcal{S}_1^\forall, \mathcal{S}_2^\exists) &= d_{\text{cor}}(\mathit{ContErr}(\mathit{ErrPairs}, \mathcal{S}_1^\forall), \mathit{ContErr}(\emptyset, \mathcal{S}_2^\exists)) \\ &\leq d_{\text{cor}}(\mathit{ContErr}(\mathit{ErrPairs}, \mathcal{S}_1^\forall), \mathit{ContErr}(\mathit{ErrPairs}, \mathcal{S}_1)) \\ &\qquad + d_{\text{cor}}(\mathit{ContErr}(\mathit{ErrPairs}, \mathcal{S}_1), \mathit{ContErr}(\emptyset, \mathcal{S}_2^\exists)) \\ &\leq d_{\text{cor}}(\mathit{ContErr}(\mathit{ErrPairs}, \mathcal{S}_1^\forall), \mathit{ContErr}(\mathit{ErrPairs}, \mathcal{S}_1)) \\ &\qquad + d_{\text{cor}}(\mathit{ContErr}(\mathit{ErrPairs}, \mathcal{S}_1), \mathit{ContErr}(\emptyset, \mathcal{S}_2)) \\ &\qquad + d_{\text{cor}}(\mathit{ContErr}(\emptyset, \mathcal{S}_2), \mathit{ContErr}(\emptyset, \mathcal{S}_2^\exists)) \\ &= d_{\text{cor}}(\mathit{ContErr}(\mathit{ErrPairs}, \mathcal{S}_1^\forall), \mathit{ContErr}(\mathit{ErrPairs}, \mathcal{S}_1)) \\ &\qquad + d_{\text{rob}}(\mathcal{S}_1, \mathcal{S}_2) \\ &\qquad + d_{\text{cor}}(\mathit{ContErr}(\emptyset, \mathcal{S}_2), \mathit{ContErr}(\emptyset, \mathcal{S}_2^\exists)) \end{aligned}$$

Now, it is easy to see that $\mathit{ContErr}(\mathit{ErrPairs}, \mathcal{S}_1^{\forall})$ is simulated by $\mathit{ContErr}(\mathit{ErrPairs}, \mathcal{S}_1)$ and $\mathit{ContErr}(\emptyset, \mathcal{S}_2)$ is simulated by $\mathit{ContErr}(\emptyset, \mathcal{S}_2^{\exists})$. Hence, we have $d_{\text{rob}}(\mathcal{S}_1^{\forall}, \mathcal{S}_2^{\exists}) \leq d_{\text{rob}}(\mathcal{S}_1, \mathcal{S}_2)$.

Dually, we have

$$\begin{aligned}
d_{\text{rob}}(\mathcal{S}_1, \mathcal{S}_2) &= d_{\text{cor}}(\mathit{ContErr}(\mathit{ErrPairs}, \mathcal{S}_1), \mathit{ContErr}(\emptyset, \mathcal{S}_2)) \\
&\leq d_{\text{cor}}(\mathit{ContErr}(\mathit{ErrPairs}, \mathcal{S}_1), \mathit{ContErr}(\mathit{ErrPairs}, \mathcal{S}_1^{\exists})) \\
&\qquad + d_{\text{cor}}(\mathit{ContErr}(\mathit{ErrPairs}, \mathcal{S}_1^{\exists}), \mathit{ContErr}(\emptyset, \mathcal{S}_2^{\forall})) \\
&\leq d_{\text{cor}}(\mathit{ContErr}(\mathit{ErrPairs}, \mathcal{S}_1), \mathit{ContErr}(\mathit{ErrPairs}, \mathcal{S}_1^{\exists})) \\
&\qquad + d_{\text{cor}}(\mathit{ContErr}(\mathit{ErrPairs}, \mathcal{S}_1^{\exists}), \mathit{ContErr}(\emptyset, \mathcal{S}_2^{\forall})) \\
&\qquad + d_{\text{cor}}(\mathit{ContErr}(\emptyset, \mathcal{S}_2^{\forall}), \mathit{ContErr}(\emptyset, \mathcal{S}_2)) \\
&= d_{\text{cor}}(\mathit{ContErr}(\mathit{ErrPairs}, \mathcal{S}_1), \mathit{ContErr}(\mathit{ErrPairs}, \mathcal{S}_1^{\exists})) \\
&\qquad + d_{\text{rob}}(\mathcal{S}_1^{\exists}, \mathcal{S}_2^{\forall}) \\
&\qquad + d_{\text{cor}}(\mathit{ContErr}(\emptyset, \mathcal{S}_2^{\forall}), \mathit{ContErr}(\emptyset, \mathcal{S}_2))
\end{aligned}$$

As before, $\mathit{ContErr}(\mathit{ErrPairs}, \mathcal{S}_1)$ is simulated by $\mathit{ContErr}(\mathit{ErrPairs}, \mathcal{S}_1^{\exists})$ and $\mathit{ContErr}(\emptyset, \mathcal{S}_2^{\forall})$ is simulated by $\mathit{ContErr}(\emptyset, \mathcal{S}_2)$. Hence, we have $d_{\text{rob}}(\mathcal{S}_1, \mathcal{S}_2) \leq d_{\text{rob}}(\mathcal{S}_1^{\exists}, \mathcal{S}_2^{\forall})$.

□

3.5 Applications of Simulation Distances

We present three examples of application of the distances defined in Section 3.3[Pg. 41] to measure interesting properties of larger systems.

3.5.1 Robustness of Forward Error-Correction Systems

Forward error-correction systems (FECS) are a mechanism of error control for data transmission on noisy channels. The *maximum tolerable bit-error rate* of these systems is the maximum number of errors the system can tolerate while still being able to successfully decode the message. We show that this property can be measured as the component robustness distance d_{rob} between a system and an ideal specification.

We examine three forward error correction systems: one with no error correction facilities, the Hamming(7,4) code [93], and triple modular redundancy [121]. Intuitively, each of these systems is at a different point in the trade-off between efficiency of transmission and the tolerable bit-error rate. By design, the system with no error correction can tolerate no errors and the Hamming(7,4) system can tolerate one error in seven bits and the triple modular redundancy system can tolerate one error in three bits (or more precisely, three independent errors in twelve bits). However, the overhead incurred increases with increasing error tolerance. The system with no error correction uses no extra bits while, the Hamming(7,4) system and the triple modular redundancy system use 3 and 8 extra bits for transmitting a four bit message. We compute the values of the error tolerance by measuring robustness with respect to an ideal system which

No error correction

```
proc sender(B0,B1,B2,B3)
    send(B0,B1,B2,B3);
proc receiver()
    receive(B0,B1,B2,B3);
    return (B0,B1,B2,B3)
```

Hamming(7,4) error correction

```
proc sender(B0,B1,B2,B3)
    P0 := B0 ⊕ B1 ⊕ B3
    P1 := B0 ⊕ B2 ⊕ B3
    P2 := B1 ⊕ B2 ⊕ B3
    send(P0,P1,B0,P2,B1,B2,B3);
proc receiver()
    receive(P0,P1,B0,P2,B1,B2,B3);
    P0 := P0 ⊕ B0 ⊕ B1 ⊕ B3;
    P1 := P1 ⊕ B0 ⊕ B2 ⊕ B3;
    P2 := P2 ⊕ B1 ⊕ B2 ⊕ B3;
    B0 := B0 ⊕ (¬ P0 . P1 . ¬ P2);
    B1 := B1 ⊕ (P0 . ¬ P1 . P2);
    B2 := B2 ⊕ (P0 . P1 . ¬ P2);
    B3 := B3 ⊕ (P0 . P1 . P2);
    return (B0,B1,B2,B3)
```

Triple modular redundancy

```
proc sender(B0,B1,B2,B3)
    send(B0,B0,B0);
    send(B1,B1,B1);
    send(B2,B2,B2);
    send(B3,B3,B3);
proc receiver()
    receive(B01,B02,B03);
    receive(B11,B12,B13);
    receive(B21,B22,B23);
    receive(B31,B32,B33);
    B0 := B01 . B02
       ∨ B02 . B03 ∨ B03 . B01;
    B1 := B11 . B12
       ∨ B12 . B13 ∨ B13 . B11;
    B2 := B21 . B22
       ∨ B22 . B23 ∨ B23 . B21;
    B3 := B31 . B32
       ∨ B32 . B33 ∨ B33 . B31;
    return (B0,B1,B2,B3)
```

Figure 3.15: Forward error correction algorithms.

T_1	T_2	$d_{\text{rob}}(T_1, T_2)$
None	Ideal	1
Hamming	Ideal	6/7
TMR	Ideal	2/3

Table 3.2: Robustness of FECS for the limit-average distance.

can tolerate an unbounded number of errors. The ideal system is modelled as a system which non-deterministically sends a number of bits and then outputs the correct message. The pseudo-code for the three systems is presented in Figure 3.15[Pg. 60]. The only errors we allow are bit flips during transmission.

The transition systems for these systems are constructed according to the following rules:

- The state space of the sender component is $\{0, 1, \#\}^n$ and the state-space of the receiver component is $\{0, 1, \#\}^n$ where n is a constant specific to the system. The state of the sender component is the list of bits to be transmitted by the sender in this round, and the state of the receiver component is the list of bits already received by the receiver in this round. The initial state is $(\natural^n, \natural^m)$.
- The input alphabet for the sender component consists of $\Sigma_{in} = \{0, 1\}^4 \cup \{\#\}$ and the output of the sender component $\Sigma_{out} = \{0, 1\}$. The input represents the bit-block to be transmitted while the output represents the

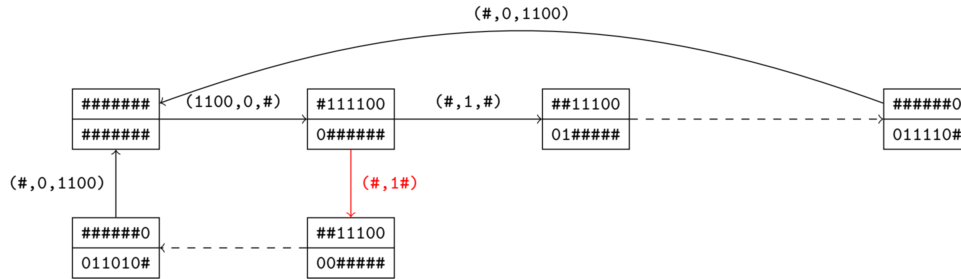


Figure 3.16: Part of the transition graph for Hamming(7,4) system.

individual bits transmitted. Similarly, the input to the receiver component is $\{0,1\}$ while the output is is $\{0,1\}^4 \cup \{\#\}$ where the input represents the individual bits received and the output represents the bit-block output finally.

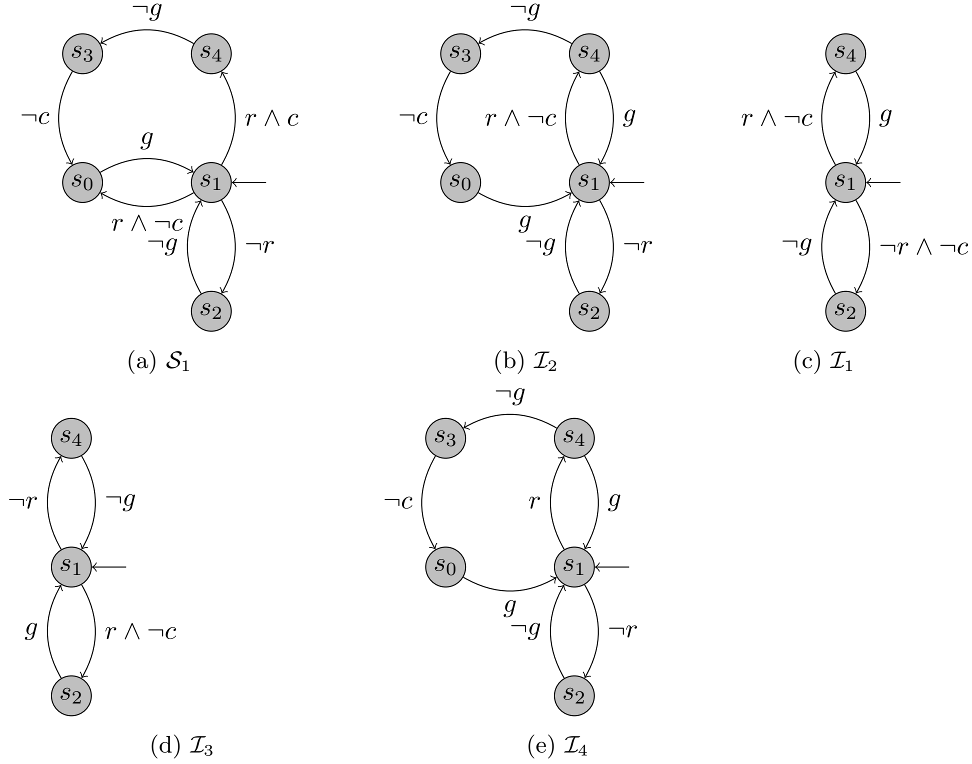
- Bit flips can occur during the transmission and the state is changed according to the bit received – i.e., if the sender sends a bit 0, and an external error flips it to 1, the receiver state changes as if a 1 were transmitted while the sender state changes as if a 0 were transmitted. These transitions which have a bit flip are considered as erroneous transmissions. To measure the robustness of the system, we will be setting error pairs *ErrPairs* to $\{(0,1),(1,0)\}$.

We explain the modelling with an example: we present the transmission of the bit block 1100 in the Hamming(7,4) system (Figure 3.16[Pg. 61]). The encoded bit string for this block is 0111100. In each state, the two components represent the sequence of bits to be sent and the sequence of bits to be received – i.e., the state of the sender and the state of the receiver. In each transition symbol, the first component represents the input message to the system; the second component represents the bit transmitted in the current step; and the third component represents the output message from the system. From the initial state $(\natural^7, \natural^7)$, on the input 1100, the transmitted bit is 0 (the first bit of the encoded string) and the state changes to $(\natural 111100, 0\natural\natural\natural\natural\natural\natural)$ on the transitions symbol $(1100, 0, \natural\natural\natural\natural)$ (assuming no errors). From this state, we go on the symbol $(\natural\natural\natural\natural, 1, \natural\natural\natural\natural)$ to the state $(\natural\natural 11100, 01\natural\natural\natural\natural\natural)$ and so on. An error transition from $(\natural 111100, 0\natural\natural\natural\natural\natural\natural)$ will lead to the state $(\natural\natural 11100, 00\natural\natural\natural\natural\natural)$.

The values of d_{rob} for these systems are summarized in Table 3.2[Pg. 60]. The robustness values clearly mirror the error tolerance values. In fact, each robustness value is equal to $1 - e$ where e is the corresponding error tolerance value.

3.5.2 Environment Restriction for Reactive Systems

In reactive systems, the transitions of the system are controlled by two agents, the system and the environment. While refining a specification for a reactive system, care has to be taken to ensure (a) all behaviors of the implementation are simulated by the specification, and (b) the behavior of the environment is not restricted more than in the specification. A number of extensions of the classical simulation relation have been suggested to include this requirement



(a) $\mathcal{S}_1$ (b) $\mathcal{I}_2$ (c) $\mathcal{I}_1$

(d) $\mathcal{I}_3$ (e) $\mathcal{I}_4$

Figure 3.17: Reactive systems.

such as ready simulation [24].

We propose a method to measure the amount of restriction the implementation system places on the environment over and above the restriction in the specification. The measure proposed here not only takes into consideration the languages of the two systems, but also the distance of the farthest unimplemented behavior in the implementation. For example, consider a specification that allows the environment behavior r_1^ω and two implementations $\mathcal{I}_1$ and $\mathcal{I}_2$ forbid it. However, say $\mathcal{I}_1$ allows the behavior $(r_1 r_2)^\omega$ whereas $\mathcal{I}_2$ allows only r_2^ω, $\mathcal{I}_1$ will be given a higher rating than $\mathcal{I}_2$.

We measure the amount of environment restriction is using the coverage distance (d_{cov}). We model a reactive system as a transition systems with the alphabet $\Sigma_{in} \cup \Sigma_{out}$ (where Σ_{in} and Σ_{out} are the environment actions (inputs) and system actions (outputs) respectively), and the transitions labeled with Σ_{in} and Σ_{out} alternate. To measure the excessive restriction on the environment, we choose an error model that assigns a constant cost 0 for every replacement of output actions Σ_{out}. We then compute the d_{cov} distance between the system and the implementation. We demonstrate that this method of measuring environment restriction by computing the distances for a *request-grant* system.

Consider the specification $\mathcal{S}_1$ and the implementations $\mathcal{I}_n$ in the Figure 3.17[Pg. 62]. All these systems are built so that every request r is granted by g in the same step or in the next step. However, if cancel c is high, there

T_1	T_2	$d_{\text{cov}}(T_1, T_2)$
$\mathcal{S}_1$	$\mathcal{S}_1$	0
$\mathcal{S}_1$	$\mathcal{I}_1$	1/2
$\mathcal{S}_1$	$\mathcal{I}_2$	1/4
$\mathcal{S}_1$	$\mathcal{I}_3$	1/4
$\mathcal{S}_1$	$\mathcal{I}_4$	0

Table 3.3: Restrictiveness of request-grant systems in Figure 3.17[Pg. 62]

should be no grant in that step. These requirements mandatorily forbid some environment behaviors, like the behavior with both r and c high all the time. The specification $\mathcal{S}_1$ restricts the environment most permissively: for every request r, cancel c is low in the current or the following step. Implementations $\mathcal{I}_1$, $\mathcal{I}_2$, $\mathcal{I}_3$ and $\mathcal{I}_4$ restrict the environment to various amounts by allowing no c's, allowing no c's for the relevant two steps, allowing no c's for the current step, and allowing no c's for the following step respectively. The restrictiveness values (d_{cov}) are summarized in Table 3.3[Pg. 63] and reflect the intuitive notion that $\mathcal{I}_1$ is the most restrictive, followed by $\mathcal{I}_2$ and $\mathcal{I}_3$, and then by the unrestrictive $\mathcal{I}_4$.

3.5.3 Test-Suite Coverage

In black-box testing, the specification of the system is used to generate a test-suite, i.e., a set of input sequences. The implementation is then tested with input sequences from the test-suite and it is ensured that the outputs produced by the implementation match the specification. For an introduction to model-based testing, see for example [68].

The quality of a test-suite is usually measured through a test-coverage metric – i.e., a metric that measures how much of the specification behaviour actually tested by the test-suite. We model a test-suite as a transition system which has no cycles, i.e., no infinite behaviours. We list several common test-coverage metrics commonly used in the literature. There have been many metrics proposed for measuring test-coverage in literature.

- *State coverage.* Given a test-suite, state coverage measures the set of all specification states visited in all the traces of the test-suite.
- *Transition coverage.* This metric measures the set of all specification transitions visited in all the traces of the test-suite.

Intuitively, in both these cases, a higher metric implies better coverage, i.e., a better quality test-suite.

We show how the coverage distance from Section 3.3.2[Pg. 43] can be used as a coverage metric and illustrate how it can be used to distinguish between test-suites that are equivalent according to the classical metrics.

Remark 3.19. *Note that the vending machine system designs used in the following discussion is for illustration only, and are not intended to be reflective of real vending machine designs.*

Consider the design for a simple vending machine from Figure 3.18a[Pg. 65]. The vending machine has four inputs, namely, $r(\textit{espresso})$, $r(\textit{macchiato})$,

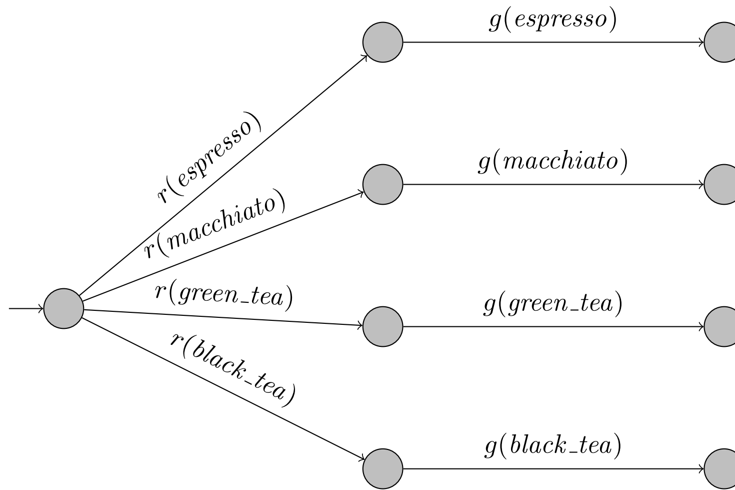
$r(green_tea)$, and $r(black_tea)$, each representing a request for the respective drinks. The outputs are $g(espresso)$, $g(macchiato)$, $g(green_tea)$, and $g(black_tea)$ – representing a dispensation of the respective drinks. The specification $\mathcal{S}$ says that every request for a drink is followed by a dispensation of the same drink.

For example, consider the following two test-suites $\mathcal{T}_1$ and $\mathcal{T}_2$ (shown in Figure 3.18b[Pg. 65] and Figure 3.18c[Pg. 65], respectively). Intuitively, $\mathcal{T}_1$ checks that the requests for espresso and macchiato are dispensed properly, while $\mathcal{T}_2$ checks the same for espresso and green tea.

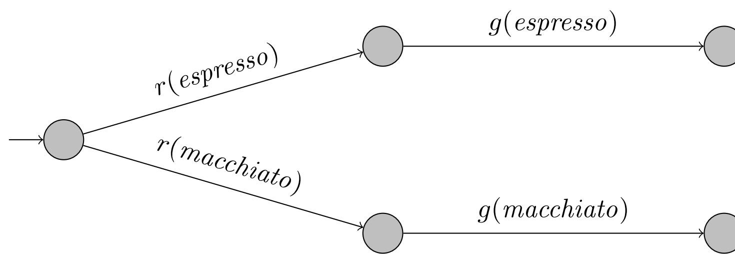
According to the standard test-coverage metrics both test-suite $\mathcal{T}_1$ and test-suite $\mathcal{T}_2$ have equivalent coverage, i.e., the test-suites both test an equal number of states (5) and transitions (4) of the specification $\mathcal{S}$. However, intuitively, there is a difference between the two test-suites. While espresso and macchiato are similar, green tea and black tea are similar drinks[1] Hence, it is likely that an implementation vending machine prepares espresso and macchiato drinks through similar internal processes, and green and black tea through similar processes. Therefore, test-suite $\mathcal{T}_2$ which tests the dispensation of espresso and green tea is likely to test more of the vending machine system than test-suite $\mathcal{T}_1$.

We can formalize this difference by specifying an error model $\mathcal{M}$ (Figure 3.18d[Pg. 65] that describes the similarity between different actions. Intuitively, error model $\mathcal{M}$ states that replacing $r(espresso)$ by $r(macchiato)$ and vice versa, and $r(green_tea)$ by $r(black_tea)$ and vice versa has a lower penalty than the other combinations of drinks. Now, coverage can be measured using the coverage distance ${d_{\text{cov}}}^{\mathcal{M}}$ with the discounted objective – we have ${d_{\text{cov}}}^{\mathcal{M}}(\mathcal{T}_1, \mathcal{S}) = 1$ while ${d_{\text{cov}}}^{\mathcal{M}}(\mathcal{T}_2, \mathcal{S}) = \frac{1}{2}$. This mirrors the intuition that $\mathcal{T}_2$ is "closer" to $\mathcal{S}$, i.e., that $\mathcal{T}_2$ covers more of $\mathcal{S}$. Note that unlike the classical coverage metrics, in the case of the coverage distance, a lower number is indicative of higher coverage.

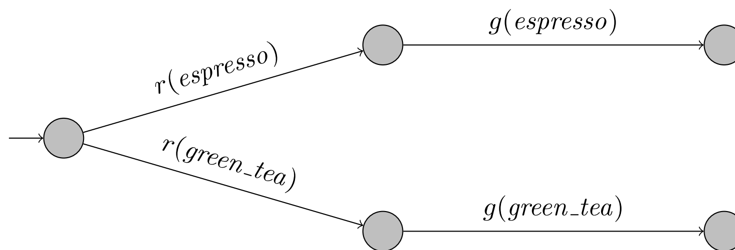
[1] Espresso and macchiato are brewed coffee-based drinks, while green tea and black tea are tea-based infusions.



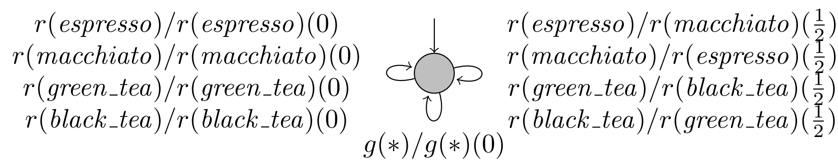
(a) Vending machine specification $\mathcal{S}$



(b) Vending machine test-suite $\mathcal{T}_1$



(c) Vending machine test-suite $\mathcal{T}_2$



(d) Vending machine error model $\mathcal{M}$

Figure 3.18: Coverage metrics for testing vending machine models

Chapter 4

Synthesis from Incompatible Specifications

Systems are often specified using multiple requirements on their behavior. In practice, these requirements can be contradictory. The classical approach to specification, verification, and synthesis demands more involved specifications that resolve any contradictions in the requirements. These resolved specifications are usually large, cumbersome, and hard to maintain or modify. In contrast, quantitative frameworks allow the formalization of the intuitive idea that what is desired is an implementation that comes "closest" to satisfying the mutually incompatible requirements, according to a measure of fit that can be defined by the requirements engineer. In the previous chapter (Chapter 3[Pg. 30]), we introduced one flexible framework for quantifying how "well" an implementation satisfies a specification—simulation distances that are parameterized by an error model.

We provide an algorithmic solution for the following quantitative synthesis question: given two (or more) behavioral requirements specified by possibly incompatible finite-state machines, and an error model, find the finite-state implementation that minimizes the maximum of the simulation distances to the individual requirements. We also demonstrate how quantitative specifications based on simulation distances might lead to smaller and easier to modify specifications. Finally, we illustrate our approach using case studies on error correcting codes and scheduler synthesis.

4.1 Motivation

A major problem for the wider adoption of techniques for the formal verification and synthesis of systems is the difficulty of writing quality specifications. As we have seen in the previous chapter, quantitative specifications have the potential to simplify the task of the designer, by enabling her to capture her intent better, and more simply. In this chapter, we focus on how quantitative specification and reasoning can be useful in cases when specifications are mutually incompatible. In practice, specifications of systems are often not monolithic. They are composed of parts that express different design requirements, possibly coming from different sources. Such high-level requirements can be therefore often

contradictory (see, for instance, [130, 21, 94] which provide methods for requirements analysis). Using the classic boolean approach, the solution would be to resolve conflicts by writing more detailed specifications that cover all possible cases of contradictions, and say how to resolve them. However, such specifications may become too large, and more importantly, the different requirements become entangled in the specification. The specifications are then much more difficult to maintain than the original requirements, as it is hard to modify one requirement without rewriting the rest of the specification. In contrast, simulation distances allow the formalization of the intuitive idea that what is desired is an implementation that comes "closest" to satisfying the requirements. More technically, we consider two questions: first, the (rigorously defined) distances from the implementation to (boolean) requirements are within given bounds, and second, the maximal distance to a requirement is minimized. It is simple to generalize the exposition and technical details that follow to minimizing any max-plus expression over the distances to the individual expressions. However, we instead choose not to complicate the presentation and restrict ourselves to simple case of the minimizing the maximal distance to a requirement.

Quantitative reasoning about systems is gaining importance with the spread of embedded systems with strict requirements on resource consumption and timeliness of response. The quantitative approach in this paper is fully compatible with quantitative resource (e.g. memory, energy) consumption requirements: the framework allows us to consider multiple specifications that model resource consumption, and it allows us to express the relative importance of resources. We can then ask the same two questions as above: first, we would like an implementation such that it consumes resources within given bounds; or, second, an implementation such that its maximal total consumption of a resource is minimized.

Synthesis from specifications [133, 131, 59] has been studied extensively as a technique to improve designer and programmer productivity. The possible utility of synthesis is higher if specifications remain at a high level. If designers are forced to write detailed low-level specifications to reconcile contradictory requirements, it decreases the usefulness of synthesis. First, it requires more effort from designers, requiring consideration of *how* a certain task will be performed, as opposed to *what* task should be performed. Second, the space of solutions to the synthesis problem is reduced, with possibly good implementations being ruled out. We therefore propose quantitative synthesis as a solution to the problem of synthesis from incompatible specifications.

Synthesis from incompatible specifications. The main technical problem we concentrate on in this paper is the problem of synthesis from incompatible specifications. The input to the synthesis problem consists of a set of (two or more) mutually incompatible specifications given by finite-state open reactive systems, and a simulation distance (given by an error model). The output should be an implementation, given by a deterministic open reactive system, that minimizes the maximal simulation distance to the given specifications.

Motivating example. Consider a system that grants exclusive access to a resource to two processes which periodically seek access to it. The input alphabet Σ_{in} consists of symbols r_1, r_2, r_1r_2 and nr representing that requests from either

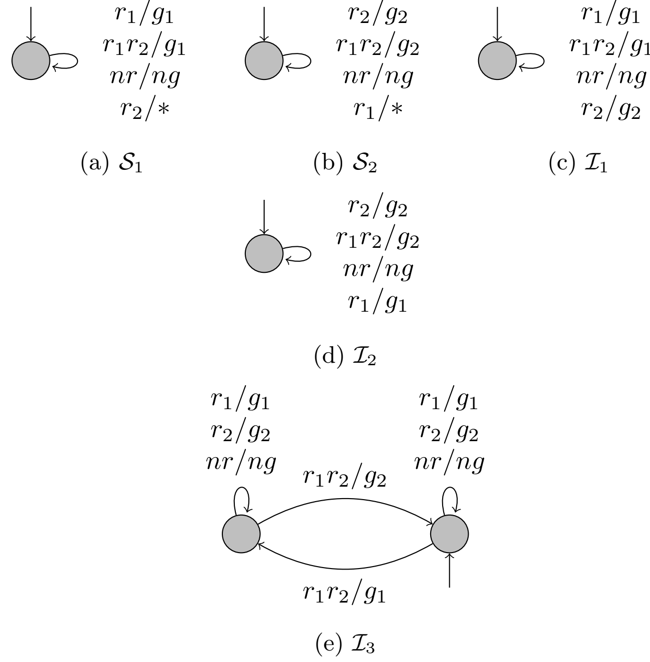


Figure 4.1: Example 1

Process 1, Process 2, both, or neither, respectively, coming in the current step. The output alphabet Σ_{out} consists of symbols g_1 (g_2) representing granting the resource to Process 1 (Process 2), and a special "don't care" symbol $*$. The specification of the system consists of two parts: the first part R_1 states that a request for the resource from Process 1 should be satisfied with a grant in the same step, and the second part R_2 states that a request for the resource from Process 2 should be satisfied with a grant in the same step. The two parts of the specification are shown in Figures 4.1a[Pg. 69] and 4.1b[Pg. 69]. The specifications are incompatible, because on input r_1r_2 the specification $\mathcal{S}_1$ allows only g_1, whereas specification $\mathcal{S}_2$ allows only g_2. Classically, the designer would have to manually resolve the conflict by, for example, constructing a specification that grants to Process 1 whenever both processes request in the same step (requirement R_3). However, the designer might not want to resolve the conflict at the specification level, but instead might want to state that she wants an implementation that comes close to satisfying the two mutually incompatible specifications, according to a measure of correctness. We use the correctness simulation distance as defined in the previous chapter for this measure.

Let us consider an error model that, intuitively, (i) assigns a cost 1 if the implementation does not grant a request that has arrived in the current step, and assigns the same cost for every step before the implementation grants the request, and (ii) assigns a cost 1 if the implementation grants a request that is not required in the current step. Let us now consider the three different implementations in Figures 4.1c[Pg. 69], 4.1d[Pg. 69], and 4.1e[Pg. 69], and their distances to the specifications $\mathcal{S}_1$ and $\mathcal{S}_2$. The implementation $\mathcal{I}_1$ always

prefers the request r_1 when the two requests arrive at the same time, while implementation $\mathcal{I}_2$ always prefers the request r_2 when two requests arrive at the same time.

The implementation $\mathcal{I}_1$ satisfies the specification $\mathcal{S}_1$, but on the input $(r_1 r_2)^\omega$, $\mathcal{I}_1$ makes a "mistake" at every step with respect to $\mathcal{S}_2$. The implementation $\mathcal{I}_1$ thus has simulation distance 0 from $\mathcal{S}_1$, and distance 1 from $\mathcal{S}_2$ for the limit-average objective. Similarly, implementation $\mathcal{I}_2$ has simulation distance 1 from $\mathcal{S}_1$ and distance 0 from $\mathcal{S}_2$ for the limit-average objective. The implementation $\mathcal{I}_3$ alternates grants in cases when the requests arrive at the same step. Its distance to both specifications would be $\frac{1}{2}$. This is because the worst-case input for this implementation is the sequence $(r_1 r_2)^\omega$ and on this input sequence, it makes a mistake in every other step, with respect to both $\mathcal{S}_1$ and $\mathcal{S}_2$.

The quantitative approach can be compared to the classical boolean approach to illustrate how it leads to specifications that are easier to modify:

- Consider an alternate requirement R_1' which says that every request by Process 1 should be granted in the next step (instead of the same step). In the boolean case, replacing requirement R_1 by R_1' also involves changing the requirement R_3 which resolves the conflict between R_1 and R_2. Requirement R_3 needs to be changed to R_3' which says that given that request r_1 happened in the previous step and r_2 happened in the current step, the output must be g_1 in the current step. However, in the quantitative case, no changes need to be done other than replacing R_1 with R_1'.
- Similarly, we can consider other ways of resolving the conflict between requirements R_1 and R_2, instead of using R_3 which prioritizes Process 1 over Process 2. We could have the requirement that we are equally tolerant to missed grants in each process (say requirement R_3') or that we tolerate twice as many missed grants in Process 1 than in Process 2, just by modifying the penalties in the error models. In the boolean case, the requirement R_3 is easily expressible, but the requirement R_3' is very hard to state without adding additional constraints to the specification. In the quantitative case, we can simply switch between R_3 and and R_3' just by changing the relative penalties for not granting r_1 or r_2.

An additional problem with writing detailed specifications in the classical boolean approach is that the different requirements become intertwined, i.e., it is very hard to modify one requirement without rewriting the rest of the specification. Changing a single requirement requires rewriting the parts of the specifications which deal with the resolution of any conflicts involving that requirement. This makes the specifications extremely hard to maintain or to change.

To illustrate how our framework can model resource consumption, we consider a system that sends messages over a network, as governed by a correctness specification. It costs a certain amount (in dollars) to send a kB of data, so it might be useful to compress data first. However, compression uses energy (in Joules). In our framework, we could add two boolean requirements saying that (a) data should not be sent on the network, and (b) compression should not be used. Then we can relax the requirement, by giving error models that have costs for sending data and using compression. In this way, the framework allows to synthesize a system where e.g. both total energy costs and total network

costs are within certain bounds. For further illustration of resource consumption modeling, we refer the reader to our case study on forward error correction codes, where the number of bits sent is the resource tracked.

Overview of results. The main result of this paper is an ϵ-optimal construction for the synthesis from incompatible specifications problem. We first consider the decision version of the problem: given k possibly mutually incompatible specifications, and a maximum distance to each specification, the problem is to decide whether there exists an implementation that satisfies these constraints. We show that the decision problem is coNP-complete (for a fixed k). The result is obtained by reduction to 2-player games with multiple limit average objectives [47]. We then present a construction of an ϵ-optimal strategy for the problem of synthesis for incompatible specifications. Furthermore, for the case of two specifications, and for the standard error model, we show that the result of our optimal synthesis procedure is always better (in a precise sense) than the result of classical synthesis from just one of the specifications. Finally, we demonstrate how our methods can enable simpler specifications, while allowing the synthesis of desirable solutions, using two case studies: on synthesis of custom forward error correction codes and on scheduler synthesis.

4.2 The Incompatible Specifications Problem

Recall the definition of the correctness distance from the previous chapter (Chapter 3[Pg. 30]). Here, we use the notation $d^{\mathcal{M}}(\mathcal{I}, \mathcal{S})$ for the correctness simulation distance from specification $\mathcal{S}$ to implementation $\mathcal{I}$ with respect to the error model $\mathcal{M}$ instead of the the notation ${d_{\text{cor}}}^{\mathcal{M}}(\mathcal{I}, \mathcal{S})$. Here, we only consider strictly synchronous reactive systems, i.e., input and output transitions strictly alternate. Further, without loss of generality, we assume that the initial state of a system is always an input state.

We specify that the error models in this chapter always assign a penalty of ∞ to mismatches of the sort σ_{in}/σ where $\sigma_{in} \neq \sigma$ and $\sigma_{in} \in \Sigma_{in}$, i.e., an input action. Intuitively, this restricts the implementation systems from re-interpreting the input actions. Further, we only consider the limit-average objective for the simulation games in this chapter.

Specifications $\mathcal{S}_i$ and error models $\mathcal{M}_i$ for $1 \leq i \leq k$ are said to be *incompatible* if $\neg\exists\mathcal{I} : \bigwedge_i d^{\mathcal{M}_i}(\mathcal{I}, \mathcal{S}_i) = 0$. Note that our definition may judge specifications compatible even if there is no common implementation which is simulated classically by each specification. This happens if there exists an implementation with the distance 0 to each of the specifications, which is possible if the specifications share long-term behavior, but differ in the short-term initial behavior.

Synthesis from incompatible specifications involves finding a "best-fit" implementation that minimizes the simulation distances to each specification. We formalize the synthesis from incompatible specifications problem as follows.

Given $\mathcal{S}_i$ and $\mathcal{M}_i$ for $1 \leq i \leq k$ as above, and a threshold vector $\mathbf{v} = \langle v_1, v_2, \ldots v_k \rangle \in \mathbb{Q}^k$, the *incompatible specifications decision problem* asks if $\exists\mathcal{I} : \forall 1 \leq i \leq k : d^{\mathcal{M}_i}(\mathcal{I}, \mathcal{S}_i) \leq v_i$.

Given specifications $\mathcal{S}_i$ and error models $\mathcal{M}_i$ for $1 \leq i \leq k$ and a bound $\epsilon > 0$, the *incompatible specifications optimization problem* is to find an implementation

$\mathcal{I}^*$ such that $\forall \mathcal{I} : \max_{i \in \{1,2,\ldots k\}} d^{\mathcal{M}_i}(\mathcal{I}^*, \mathcal{S}_i) \leq \max_{i \in \{1,2,\ldots k\}} d^{\mathcal{M}_i}(\mathcal{I}, \mathcal{S}_i) + \epsilon$. We call such an implementation $\mathcal{I}^*$ an ϵ-optimal implementation.

Theorem 4.1. *The incompatible specifications decision problem is* coNP-*complete for a fixed k.*

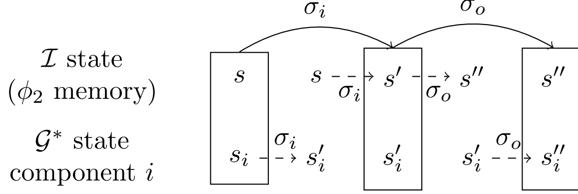


Figure 4.2: Working of ϕ_2: Solid edges are transitions in $\mathcal{G}^*$ and dashed edges are transitions in $d^{\mathcal{M}_i}(\mathcal{I}, \mathcal{S}_i)$

Proof. First, we prove that the incompatible specifications decision problem is in coNP. We reduce the problem to the decision problem in 2-player games with *MLimAvg* objectives.

Given specifications $\mathcal{S}_i$ and error models $\mathcal{M}_i$ for $1 \leq i \leq k$, consider the following game graph $\mathcal{G}^*$ with:

- Player 1 states $S_1 = (S^1_{in} \times S^{\mathcal{M}_1}) \times \ldots \times (S^k_{in} \times S^{\mathcal{M}_k})$ where S^i_{in} are the input states of $\mathcal{S}_i$ and $S^{\mathcal{M}_i}$ are the states of $\mathcal{M}_i$;
- Player 2 states $S_2 = (S^1_{out} \times S^{\mathcal{M}_1}) \times \ldots \times (S^k_{out} \times S^{\mathcal{M}_k})$ where S^i_{out} are the output states of $\mathcal{S}_i^{\mathcal{M}_i}$ and $S^{\mathcal{M}_i}$ are the states of $\mathcal{M}_i$; and
- A transition from state $((s_1, s^{\mathcal{M}}_1), \ldots, (s_k, s^{\mathcal{M}}_k))$ to $((s'_1, s^{\mathcal{M}'}_1), \ldots, (s'_k, s^{\mathcal{M}'}_k))$ on an input action σ_{in} exists if and only if each of $(s_i, \sigma_{in}, s'_i) \in \Delta^i$ where Δ^i is the transition set of $\mathcal{S}_i$ and $(s^{\mathcal{M}}_i, \sigma_{in}/\sigma_{in}, s^{\mathcal{M}}_i)$ is a transition of the $\mathcal{M}_i$. The weight function $v : \Delta \to \mathbb{Q}^n$ maps each such transition to a vector of weights with the ith component being the weight of the corresponding transition $\mathcal{M}_i$ transition $(s^{\mathcal{M}}_i, \sigma/\sigma, s^{\mathcal{M}}_i)$.
- A transition from state $((s_1, s^{\mathcal{M}}_1), \ldots, (s_k, s^{\mathcal{M}}_k))$ to $((s'_1, s^{\mathcal{M}'}_1), \ldots, (s'_k, s^{\mathcal{M}'}_k))$ on an output action σ_{out} exists if and only if for each i, there exists a $\mathcal{S}_i$ transition $(s_i, \sigma^i_{out}, s'_i)$ and an $\mathcal{M}_i$ transition $(s^{\mathcal{M}}_i, \sigma_{out}/\sigma^i_{out}, s^{\mathcal{M}'}_i)$. The weight function $v : \Delta \to \mathbb{Q}^n$ maps each such transition to a vector of weights with the ith component being the weight of the corresponding transition $\mathcal{M}_i$ transition $(s^{\mathcal{M}}_i, \sigma_{out}/\sigma^i_{out}, s^{\mathcal{M}}_i)$.

Intuitively, Player 1 chooses the input actions and Player 2 chooses output actions as well as transitions from $\mathcal{S}_i$ that should simulate the corresponding output action. We prove that a witness implementation exists if and only if there exists a finite memory Player 2 strategy in $\mathcal{G}^*$ for the *MLimAvg* objective.

(a) For any implementation $\mathcal{I}$, consider the games $\mathcal{Q}_{\mathcal{I},\mathcal{S}_i,\mathcal{M}_i}$ and the optimal Player 2 strategy ϕ^i_2 in each. By standard results on *LimAvg* games, we have that each ϕ^i_2 is memoryless. From these strategies, we construct a finite-memory Player 2 strategy ϕ_2 in $\mathcal{G}^*$ with the state space of $\mathcal{I}$ as the memory. The memory update function of ϕ_2 mimics the transition relation of $\mathcal{I}$. Let s be the current state of ϕ_2 memory and let $((s_1, s^{\mathcal{M}}_1), \ldots, (s_k, s^{\mathcal{M}}_k))$ be the current state in $\mathcal{G}^*$. By construction, s is Player 1 state in $\mathcal{I}$ iff $(s_1, \ldots, s_k)$ is Player 1 state in $\mathcal{G}^*$.

- If $((s_1, s_1^{\mathcal{M}}), \ldots, (s_k, s_k^{\mathcal{M}}))$ is a Player 1 state, Player 1 chooses an input symbol $\sigma_{in} \in \Sigma_{in}$ and updates the $\mathcal{G}^*$ state. The memory of ϕ_2 is updated to s' which is the unique successor of s on σ_i.
- Next, if the current state $((s'_1, s_1^{\mathcal{M}'}), \ldots, (s'_k, s_k^{\mathcal{M}'}))$ is a Player 2 state, the memory of ϕ_2 is updated to the unique successor s'' of s' in $\mathcal{I}$ (Player 2 states have unique successors in implementations). If (s', σ_{out}, s'') is the corresponding $\mathcal{I}$ transition, the chosen $\mathcal{G}^*$ state is $((s''_1, s_1^{\mathcal{M}''}), \ldots, (s''_k, s_k^{\mathcal{M}''}))$ where each ϕ_2^i chooses the $\mathcal{S}_i$ transition $(s'_i, \sigma^i_{out}, s''_i)$ in the state $(s'', \sigma_{out}, s'_i)$. The error model state is updated accordingly.

The construction of ϕ_2 is explained in Figure 4.2[Pg. 72].

For every path ρ conforming to ϕ_2, we can construct a path ρ^i in $\mathcal{Q}_{\mathcal{I},\mathcal{S}_i,ErrorModel_i}$ conforming to ϕ_2^i from the memory of ϕ_2 and the projection of ρ to i^{th} component (See Figure 4.2[Pg. 72]). Furthermore, the weights of the ith component of ρ have the same *LimAvg* as the weights of ρ^i. Therefore, the *LimAvg* value of the ith component of ρ is bound by $d^{\mathcal{M}_i}(\mathcal{I}, \mathcal{S}_i)$. This shows that the *MLimAvg* value of ϕ_2, $Val(\phi_2)$ is at most the maximum of $d^{\mathcal{M}_i}(\mathcal{I}, \mathcal{S}_i)$.
(b) For every finite-memory strategy ϕ_2 of Player 2 in $\mathcal{G}^*$, we can construct an implementation $\mathcal{I}$ such that $Val(\phi_2) \geq \max_{i\in\{1,2,\ldots k\}} d^{\mathcal{M}_i}(\mathcal{I}, \mathcal{S}_i)$, by considering the product of $\mathcal{G}^*$ and the memory of ϕ_2 and by removing all transitions originating from Player 2 states which are not chosen by ϕ_2.

From the results of [47], we have that solving *MLimAvg* games for the threshold $\{0\}^k$ for finite memory strategies is coNP-complete. However, we can reduce the problem of solving *MLimAvg* games for a threshold $\mathbf{v} \in \mathbb{Q}^k$ to a problem with threshold $\{0\}^k$ by subtracting $\mathbf{v}$ from each of the edge weights. This reduction is obviously polynomial for a fixed k. Therefore, the inconsistent specifications decision problem can be solved in coNP time in the size of $\mathcal{G}^*$, which in turn is polynomial in the size of the input for fixed k. To show the coNP hardness, we can use a modification of the proof of coNP hardness of *MLimAvg* games by reduction from the complement of 3-SAT. □

Now, we can find an ϵ-optimal implementation for the optimization problem by doing a binary search on the space of thresholds.

Corollary 4.2. *The incompatible specifications optimization problem can be solved in* EXPTIME *for a fixed k, ϵ and W, where W is the absolute value of the maximum cost in the error models.*

Proof. Without loss of generality, let $\epsilon = \frac{1}{q}$ for $q \in \mathbb{N}$. As the simulation distances are between 0 and W, we do a binary search on vectors of form $\{t\}^k$ to find $\{N/Wq\}^k$, the highest threshold for which an implementation exists. Since, the accuracy required is ϵ, the number of search steps is $O(\log(W/\epsilon)) = O(\log(Wq))$. We find an implementation (equivalently, a Player 2 finite memory strategy) with a value of at least this threshold. We reduce the problem to an equivalent threshold problem with integer weights and threshold $\{0\}^k$ by multiplying weights by Wq and subtracting $\{N\}^k$. From [47] and [52], we have that memory of size $O(|\mathcal{G}^*|^2 \cdot (|\mathcal{G}^*|qW)^k)$ is sufficient and further, this strategy can be computed in EXP time. Therefore, by guessing a strategy and checking for sufficiency, we have an EXP time algorithm. □

For the qualitative error model (Figure 3.7c[Pg. 39]) and any set of incompatible specifications, for all implementations the distance to at least one of the specifications is ∞. However, for the standard error model of [36], we show for the case of two specifications that it always is possible to do better.

Proposition 4.3. *For specifications $\mathcal{S}_1$ and $\mathcal{S}_2$ with the standard error model $\mathcal{M}$, let $\delta = \min(d^{\mathcal{M}}(\mathcal{S}_1, \mathcal{S}_2), d^{\mathcal{M}}(\mathcal{S}_2, \mathcal{S}_1))$. For every $\epsilon > 0$, there exists an implementation $\mathcal{I}^*$ with $d^{\mathcal{M}}(\mathcal{I}^*, \mathcal{S}_1) < \delta/2 + \epsilon$ and $d^{\mathcal{M}}(\mathcal{I}^*, \mathcal{S}_2) < \delta/2 + \epsilon$.*

Proof. Without loss of generality, let $d^{\mathcal{M}}(\mathcal{S}_1, \mathcal{S}_2) \leq d^{\mathcal{M}}(\mathcal{S}_2, \mathcal{S}_1)$. Consider the game graph of $\mathcal{Q}_{\mathcal{S}_1,\mathcal{S}_2,\mathcal{M}}$ and modify it by letting Player 2 choose the $\mathcal{S}_1$ output transitions, i.e., Player 1 chooses the inputs and Player 2 chooses both $\mathcal{S}_1$ and $\mathcal{S}_2$ outputs. Let ϕ_2^* be the optimal Player 2 strategy in this game. From ϕ_2^*, we construct two different implementations $\mathcal{I}_1$ and $\mathcal{I}_2$ having as the state space the product of the state spaces of $\mathcal{S}_1$ and $\mathcal{S}_2$. In the transition set,

- There exists an input transition from (s_1, s_2) to (s_1', s_2') on the input symbol σ_i if and only if (s_1, σ_i, s_1') and (s_2, σ_i, s_2') are input transitions of $\mathcal{S}_1$ and $\mathcal{S}_2$;
- There exists an output transition $((s_1, s_2), \sigma_o, (s_1', s_2'))$ in $\mathcal{I}_1$ iff ϕ_2^* chooses the $\mathcal{S}_1$ transition (s_1, σ_o, s_1') from state $(s_1, \#, s_2)$ and the $\mathcal{S}_2$ transition (s_2, σ_o, s_2') in state (s_1', σ_o', s_2); and
- There exists an output transition $((s_1, s_2), \sigma_o, (s_1', s_2'))$ on σ_o in $\mathcal{I}_2$ iff ϕ_2^* chooses the $\mathcal{S}_1$ transition (s_1, σ_o', s_1') from state $(s_1, \#, s_2)$ and the $\mathcal{S}_2$ transition (s_2, σ_o', s_2') in state (s_1', σ_o', s_2) and $\mathcal{S}_2$ transition corresponds to the $\mathcal{S}_2$ transition (s_2, σ_o, s_2').

Intuitively, ϕ_2^* chooses the most benevolent $\mathcal{S}_1$ behavior and $\mathcal{I}_1$ implements this $\mathcal{S}_1$ behavior, while $\mathcal{I}_2$ is the $\mathcal{S}_2$ behavior used to simulate this behaviour in the game.

Now, we construct $\mathcal{I}^*$ by alternating between $\mathcal{I}_1$ and $\mathcal{I}_2$. For each Player 1 state (s_1, s_2) in $\mathcal{I}_i$, let $TU((s_1, s_2))$ be the tree unrolling of $\mathcal{I}_i$ from (s_1, s_2) to a depth $N \in \mathbb{N}$ and let $\mathcal{T}(\mathcal{I}_i)$ be the disjoint union of such trees. Let $\mathcal{I}^*$ be the union of $\mathcal{T}(\mathcal{I}_1)$ and $\mathcal{T}(\mathcal{I}_2)$ where each transition to a leaf state (s_1, s_2) in $\mathcal{T}(\mathcal{I}_1)$ is redirected to the root of $TU((s_1, s_2))$ in $\mathcal{T}(\mathcal{I}_2)$, and vice versa.

We now show that $d^{\mathcal{M}}(\mathcal{I}^*, \mathcal{S}_i) < \delta/2$. Consider the Player 2 strategy ϕ_2 in $\mathcal{Q}_{\mathcal{I}^*,\mathcal{S}_2,\mathcal{M}}$: to simulate an $\mathcal{I}^*$ transition from (s_1, s_2) to (s_1', s_2') on σ_o, ϕ_2 chooses the $\mathcal{S}_2$ transition (s_2, σ_o, s_2'). If $((s_1, s_2), \sigma_o, (s_1', s_2'))$ was from $\mathcal{T}(\mathcal{I}_2)$, the cost of the simulation step is 0, and otherwise it is equal to the corresponding transition from $\mathcal{Q}_{\mathcal{S}_1,\mathcal{S}_2,\mathcal{M}}$. Now, fix ϕ_2 in $\mathcal{Q}_{\mathcal{I}^*,\mathcal{S}_2,\mathcal{M}}$ and let C be the cycle of the path obtained by fixing the optimal Player 1 strategy. Cycle C is composed of paths through $\mathcal{I}_1$ and $\mathcal{I}_2$ each of length N. The cost of the path through $\mathcal{I}_2$ is 0. The cost of the path through $\mathcal{I}_1$ is equal to the cost of the corresponding cycle in $\mathcal{Q}_{\mathcal{S}_1,\mathcal{S}_2,\mathcal{M}}$. If N is large enough, the path through $\mathcal{I}_1$ is composed of an acyclic part of length at most $n = 2 \cdot |\mathcal{Q}_{\mathcal{S}_1,\mathcal{S}_2,\mathcal{M}}|$ and of cyclic paths of average cost less than $d^{\mathcal{M}}(\mathcal{S}_1, \mathcal{S}_2) = \delta$. Therefore, for all $\epsilon > 0$ and $N > \frac{nW}{\epsilon}$ we have

$$d^{\mathcal{M}}(\mathcal{I}^*, \mathcal{S}_2) \leq Val(\phi_2) \leq \frac{(N - n) \cdot \delta + n \cdot W}{2N} \leq \frac{\delta}{2} + \epsilon < \delta$$

Similarly, we can show $d^{\mathcal{M}}(\mathcal{I}^*, \mathcal{S}_1) < \delta/2 + \epsilon$ to complete the proof. ☐

4.3 Case studies

We present two case studies to demonstrate the use of simulation distances for modeling conflicting requirements. These case studies do not consider large-scale examples, but rather serve to demonstrate that simulation distances and the synthesis from incompatible specifications framework are in principle suitable for specifying real-world problems.

4.3.1 Case study: Synthesis of Forward Error Correcting Codes

Consider the task of sending messages over an unreliable network. Forward Error Correcting codes (FECs) are encoding-decoding schemes that are tolerant to bit-flips during transmission, i.e., the decoded message is correct in-spite of errors during transmission. For example, the well-known Hamming (7,4) code can correct any one bit-flip that occurs during the transmission of a bit-block. The Hamming (7,4) code transmits 7 bits for every 4 data bits to be transferred, and the 3 additional bits are parity bits.

Suppose bit-blocks of length 3 are to be transferred over a network where at most 1 bit-flip can occur during transmission. We want to minimize the number of transmitted bits. Furthermore, we also allow some errors in the decoded block. Therefore, we have two incompatible specifications:

- *Efficiency.* To minimize the number of bits transmitted, we add a requirement that only 3 bits are transmitted and an error model that has a constant penalty of e for each additional bit transmitted.
- *Robustness.* We want the decoded block to be as correct as possible. In a standard FEC scheme, all bits are given equal importance. However, to demonstrate the flexibility of our techniques, we consider the first bit to be the most significant one, and the third to be the least significant one. We add a requirement that the decoded bit block is the same as the original, with the following error model: An error in the first, second, and third bit have a cost of $4d$, $2d$, and d, respectively.

Formal modeling. The output and input alphabets are $\{T_0, T_1, R_0, R_1, O_0, O_1, \perp\}$ and $\{I_0, I_1, F, \neg F, \perp\}$ where T_i, R_i, I_i and O_i stand for transmission, receiving, input and output of bit i respectively. Symbols F and $\neg F$ denote whether a bit-flip occurs or not during the current transmission. Symbol $\perp$ is used whenever the input/output does not matter.

Remark 4.4. *Here, we use the correctness distance to measure robustness instead of the robustness distance. This is done by modelling the system along with the uncontrollable errors.*

Example 4.5. *For example, the diagram below represents the transmission of bit-block* 010 *through a system without any error correction.*

$$\begin{array}{lcccccccccccc} In & I_0 & I_1 & I_0 & \perp & F & \perp & \neg F & \perp & \neg F & \perp & \perp & \perp \\ Out & \perp & \perp & \perp & T_0 & R_1 & T_1 & R_1 & T_0 & R_0 & O_1 & O_1 & O_0 \end{array}$$

First, three bits are input. Next, each of the three bits is transmitted and received. The environment decides that the first bit is flipped and the value received is 1 *even though* 0 *is transmitted. Finally, the bit block* 110 *is output.*

In addition to *Efficiency* and *Robustness* requirements above, we need the following. For these, we use the qualitative error model where even a single error is not allowed.

- *Encoding and Decoding.* For any input (resp., received) bit-block, the same sequence of bits should be transmitted (resp. output). The specification remembers the transmitted (resp., output) bits for each input (resp., transmitted) bit-block and ensures that the same bits are transmitted (resp., output) in the future.
- *Reception.* The received bit should be correctly flipped or not based on whether the input is F or $\neg F$.

Results. For different relative values of efficiency penalty e and robustness penalty d, different optimal FEC schemes are obtained. Suppose $b_1 b_2 b_3$ is the input bit-block.

- $e = 1 \wedge d = 100$. The implementation is fully robust, i.e., always outputs the right bit-block. For example, one of the optimal strategies transmits the 6 bits b_1, b_2, b_3, $b_2 \oplus b_3$, $b_1 \oplus b_3$ and $b_1 \oplus b_2$. The bit-block can always be recovered from the received bits. This has a total error of 3 for efficiency and 0 for robustness per round.
- $e = 100 \wedge d = 1$. The implementation transmits only the three input bits and in the worst case outputs the most significant bit wrong. The worst-case errors are 0 for efficiency and 4 for robustness per round.
- $e = 10 \wedge d = 10$. The implementation ensures the correctness of the most significant bit by transmitting it thrice (triple modular redundancy), i.e., transmits the 5 bits b_1, b_1, b_1, b_2 and b_3. In the worst case, the second bit is output wrong and the error for efficiency is 20 and for robustness is 20 per round.

These results show how we can obtain completely different FECs just by varying the costs in the error models.

4.3.2 Case study: Optimal Scheduling for Overloads

Consider the task of scheduling on multiple processors, where processes have definite execution times and deadlines. Deadlines are either “soft”, where a small delay is acceptable, but undesirable; or “hard”, where any delay is catastrophic. During overload, processes are either delayed or dropped completely; and usually these processes are chosen based on priorities. Our techniques can be used to schedule based on exact penalties for missing deadlines or dropping processes.

Each process repeatedly requests execution and scheduling is based on time-slices with each processor executing a single process in a time-slice. A process $\mathcal{P}(t, d, c)$ represents:

- the time-slices t needed for the computation;
- the deadline d from invocation time; and
- the minimum time c between the completion of one invocation and the next request.

We model a process as a reactive system with inputs $\{r, \tilde{r}\}$ and outputs $\{g, \tilde{g}, c\}$. The input r represents an invocation, the output g represents a single time-slice of execution, and the output c indicates completion of the invocation.

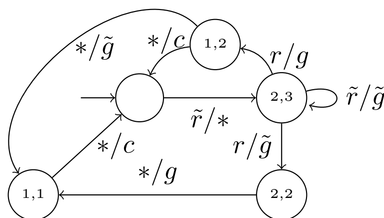


Figure 4.3: Modelling processes: $\mathcal{P}(2, 3, 1)$

In Figure 4.3[Pg. 77], all states (except the initial) are labeled by two numbers (t, d) representing, respectively, remaining execution steps, and time to deadline. Once request r is issued, execution starts at the state labeled $(2, 3)$ (input and output transitions are drawn together for readability). If the first time slice is granted, the execution goes to state $(2, 1)$ (i.e., deadline in two steps, and one step of work remaining). If the time slice is not granted, the execution transitions to a state labeled by $(2, 2)$. The model (specification) ensures that the task is completed before the deadline. After it is completed, the control is in the initial state, where a request cannot be issued for at least one time step.

We define both hard and soft deadline error models. In the hard deadline error model, a missed deadline leads to a one-time large penalty p_l, whereas in the soft deadline error model, a small recurring penalty p_s occurs every step until the process is finished. Furthermore, we have a specification that no more than n processes can be scheduled in each step, with the qualitative failure model (Figure 3.7c[Pg. 39]). We describe some optimal implementations obtained for various inputs.

- For two $\mathcal{P}(3, 6, 3)$ processes and one processor, we obtain a 0 cost schedule where each process is alternately scheduled till completion. This schedule is obtained independently of whether the deadlines are hard or soft.
- For $P_1 = \mathcal{P}(5, 5, 5)$, $P_2 = \mathcal{P}(3, 5, 5)$, and $P_3 = \mathcal{P}(2, 5, 5)$ with P_1 on a soft deadline (i.e. with the soft deadline error model described above), P_2 on a hard deadline, and P_3 on a hard deadline. With $p_s = 1$ and $p_l = 10$, we get a scheduler where P_2 and P_3 are treated as having a higher priority. Whenever P_2 or P_3 requests arrive, P_1 is preempted till P_2 and P_3 finish.
- For the same processes, but with $p_s = 5 \wedge p_l = 10$, we get a scheduler where P_1 is preferred over P_2 and P_3.

Chapter 5

Discussion

We close this part with discussion of possible extensions and related work.

5.1 Extensions and Future Work

In this section, we present a smorgasbord of possible extensions to the simulation distances framework. Each of these topics is a promising direction for future study.

Applications and Practicality. While we have covered several applications of the simulation distances framework in the case studies from the previous chapters. However, a large scale case study needs to be done to study the practicality of the framework. Towards this, we intend to rewrite a medium sized classical specification in the simulation distances framework to study the advantages and short-comings of the framework.

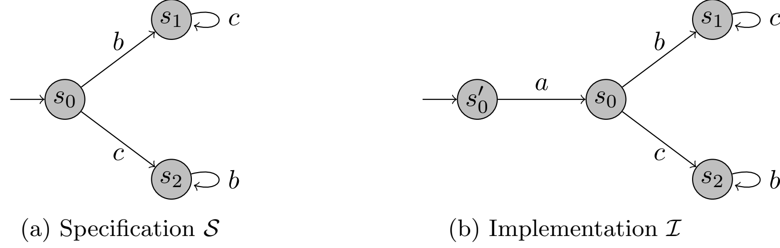
Another possible approach to make the simulation distances framework practical is using symbolic algorithms. In practice, most specifications and systems are expressed symbolically. However, it is not straightforward to apply symbolic techniques to simulation distances framework as there are no known symbolic algorithms for solving games with quantitative objectives.

Linear Distances versus Branching Distances. As with boolean specifications, a distinction can be made between linear trace containment (language inclusion) and branching trace containment (simulation relation) even in the quantitative setting. For example, the linear version of the correctness distance can be defined as follows: Given systems $\mathcal{I}$ and $\mathcal{S}$ and an error model $\mathcal{M}$, the *correctness inclusion distance* is

$$d_{\mathcal{M}}^{lin}(\mathcal{I}, \mathcal{S}) = \sup_{\pi_1 \in \mathit{Traces}(\mathcal{I})} \inf_{\pi_2 \in \mathit{Traces}(\mathcal{S})} \mathcal{M}(\pi_1/\pi_2)$$

where if $\pi_1 = s_0\sigma_0 s_1\sigma_1\ldots$, and $\pi_2 = s_0'\sigma_0' s_1'\sigma_1'\ldots$, we have $\pi_1/\pi_2 = (\sigma_0/\sigma_0')(\sigma_1/\sigma_1')\ldots$. The linear versions of the distances we have defined in the preceding chapters have many of the same properties as the branching versions. However, as in the classical boolean case, the linear versions of the distances are computationally more expensive to compute.

Further, in the classical case, we have that if $\mathcal{S}$ is deterministic, then $\mathcal{S}$ simulates $\mathcal{I}$ if and only if $Lang(\mathcal{I}) \subseteq Lang(\mathcal{S})$. However, in the quantitative case, there exists a deterministic specification $\mathcal{S}$ and an implementation $\mathcal{I}$ (see Figure 5.1[Pg. 79]) such that $d_{\text{cor}}(\mathcal{I}, \mathcal{S})$ is strictly more than $d^{lin}(\mathcal{I}, \mathcal{S})$ even for the standard error model. In Figure 5.1[Pg. 79], we have that $d_{\text{cor}}(\mathcal{I}, \mathcal{S}) = 1$ while $d^{lin}(\mathcal{I}, \mathcal{S}) = 0$ for the standard error model and limit-average objectives.



(a) Specification $\mathcal{S}$ (b) Implementation $\mathcal{I}$

Figure 5.1: Systems such that ${d_{\text{cor}}}^{lin}(\mathcal{I}, \mathcal{S}) < d_{\text{cor}}(\mathcal{I}, \mathcal{S})$ even when $\mathcal{S}$ is deterministic.

Continuous Look-ahead Simulation Distances. In the classical boolean case, continuous look-ahead simulations [125, 103] are used to bridge the gap between simulation and language inclusion. Continuous look-ahead simulations are parameterized by a parameter $k \in \mathbb{N}$. A specification $\mathcal{S}$ k-continuous look-ahead simulates an implementation $\mathcal{I}$ if Player 2 has a strategy to win in the following variation of the simulation game:

- First, Player 1 chooses a finite trace of k transitions from the implementation;
- In each round, Player 1 chooses a transition from the implementation and Player 2 chooses a transition from the specification.

As in the simulation game, Player 2 wins if the sequence of actions from the trace chosen from the specification match the sequence of actions from trace chosen from the implementation, and Player 1 wins otherwise. Intuitively, in a k-continuous look-ahead simulation, Player 2 is given information about the next k implementation transitions that will be chosen by Player 1 before she has to pick the corresponding specification transition. As Player 2 has more information, it can easily be seen that:

- if $\mathcal{S}$ simulates implementation $\mathcal{I}$, then $\mathcal{S}$ also k-continuous look-ahead simulates $\mathcal{I}$;
- if $\mathcal{S}$ $k-1$-continuous look-ahead simulates implementation $\mathcal{I}$, then $\mathcal{S}$ also k-continuous look-ahead simulates $\mathcal{I}$; and
- if $\mathcal{S}$ k-continuous look-ahead simulates implementation $\mathcal{I}$, then $Traces(\mathcal{I}) \subseteq Traces(\mathcal{S})$.

We can define the quantitative continuous look-ahead simulation distances by replacing the boolean acceptance condition in the continuous look-ahead simulation games with the corresponding error model as in standard quantitative simulation games. We write ${d_{\text{cor}}}^{k}_{\mathcal{M}}(\mathcal{I}, \mathcal{S})$ for the corresponding k-continuous look-ahead simulation distance from $\mathcal{S}$ to $\mathcal{I}$ with respect to error model $\mathcal{M}$. We have the following theorem.

Theorem 5.1. *For any error model $\mathcal{M}$, specification $\mathcal{S}$ and implementation $\mathcal{I}$, we have*

- $d_{cor}{}_{\mathcal{M}}^{k}(\mathcal{I},\mathcal{S}) \leq d_{cor}{}_{\mathcal{M}}(\mathcal{I},\mathcal{S})$;
- $d_{cor}{}_{\mathcal{M}}^{k}(\mathcal{I},\mathcal{S}) \leq d_{cor}{}_{\mathcal{M}}^{k-1}(\mathcal{I},\mathcal{S})$; *and*
- $d_{cor}{}_{\mathcal{M}}^{k}(\mathcal{I},\mathcal{S}) \leq d_{cor}{}_{\mathcal{M}}^{lin}(\mathcal{I},\mathcal{S})$.

It needs to examined if the properties proved for simulation distances such as abstraction, compositionality, and triangle inequality hold for continuous look-ahead simulation distances.

Multi-Pebble Simulation Distances. Like continuous look-ahead simulation relations, multi-pebble simulation relations are also coarser relations on the set of systems than simulation relations while still being finer than language inclusion. Intuitively, in a k-pebble simulation game, Player 2 is allowed to "hedge her bets" by picking k separate transitions instead of a single transition in a standard simulation game. For example, in the first step of the 2-pebble simulation game for specification $\mathcal{S}$ and implementation $\mathcal{I}$ from Figure 5.1[Pg. 79], Player 2 is not forced to choose to move to either s_1 or s_2 is the specification, but instead can post-pone the decision to later. A k-pebble simulation game for specification $\mathcal{S}$ and implementation $\mathcal{I}$ proceeds as follows:

- The current state of the game contains the current $\mathcal{I}$ state and k current $\mathcal{S}$ states.
- In each round, Player 1chooses an implementation transition from the current $\mathcal{I}$ state; and
- Player 2chooses k separate specification transitions from the current $\mathcal{S}$ states.

Player 2 wins the game if at least one of the $\mathcal{S}$ traces match the $\mathcal{I}$ trace chosen by Player 1. As with continuous look ahead simulations, we have that:

- if $\mathcal{S}$ simulates implementation $\mathcal{I}$, then $\mathcal{S}$ also k-pebble simulates $\mathcal{I}$;
- if $\mathcal{S}$ $k-1$-pebble simulates implementation $\mathcal{I}$, then $\mathcal{S}$ also k-pebble simulates $\mathcal{I}$; and
- if $\mathcal{S}$ k-pebble simulates implementation $\mathcal{I}$, then $\mathit{Traces}(\mathcal{I}) \subseteq \mathit{Traces}(\mathcal{S})$.

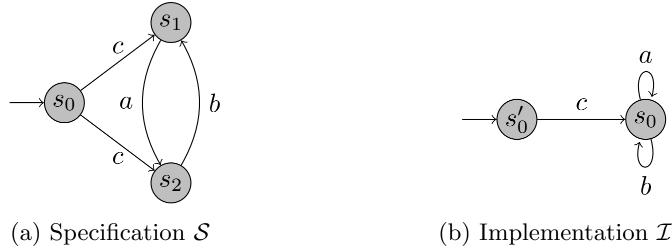
An additional property is that if $k \geq |\mathcal{S}|$ where $|\mathcal{S}|$ is the number of states in $\mathcal{S}$, then we have that $\mathcal{S}$ k-pebble simulates $\mathcal{I}$ if and only if $\mathit{Traces}(\mathcal{I}) \subseteq \mathit{Traces}(\mathcal{S})$.

Defining the quantitative k-pebble simulation game is slightly more involved due to the multiple transitions. However, intuitively, the value of a play is the minimum of the values assigned to the k words over $\Sigma \times \Sigma$ built considering the implementation trace in the play and the k separate traces are of the specification in the play.

Theorem 5.2. *For any error model $\mathcal{M}$, specification $\mathcal{S}$ and implementation $\mathcal{I}$, we have*

- $d_{cor}{}_{\mathcal{M}}^{k-peb}(\mathcal{I},\mathcal{S}) \leq d_{cor}{}_{\mathcal{M}}(\mathcal{I},\mathcal{S})$;
- $d_{cor}{}_{\mathcal{M}}^{k-peb}(\mathcal{I},\mathcal{S}) \leq d_{cor}{}_{\mathcal{M}}^{k-1-peb}(\mathcal{I},\mathcal{S})$; *and*
- $d_{cor}{}_{\mathcal{M}}^{k-peb}(\mathcal{I},\mathcal{S}) \leq d_{cor}{}_{\mathcal{M}}^{lin}(\mathcal{I},\mathcal{S})$.

However, we there exist specification $\mathcal{S}$ and implementation $\mathcal{I}$ (shown in Figure 5.2[Pg. 81]) such that even for $k = |\mathcal{S}|$, the k-pebble simulation distance is not equal to the linear distance.



(a) Specification $\mathcal{S}$ (b) Implementation $\mathcal{I}$

Figure 5.2: Systems such that ${d_{\text{cor}}}^{lin}(\mathcal{I}, \mathcal{S}) < d_{\text{cor}}(\mathcal{I}, \mathcal{S})$ even when $\mathcal{S}$ is deterministic.

Interface Simulation Distances. In [33], we extended simulation distances to a restricted class of interface automata. Interface automata are a formalism used to specify temporal aspects of system interfaces. Intuitively, interface automata are a transition system over inputs and outputs at a system interface. We prove various quantitative version of the classical properties of interface automata such as compositionality and provide an algorithm for finding the optimal quantitative shared refinement of two incompatible interface automata.

Additional Quantitative Objectives. In our work on simulation distances, we consider only two kinds of quantitative objectives for accumulating penalties—the limit-average objective and the discounted sum objective. However, a case can be made for the use of several other quantitative objectives. Foremost among these is the ratio limit-average objective where each individual penalty is specified by two numbers, a cost and a length. The value of a trace is then given by the average cost per length over the long run. For example, the length parameter could then be used to ensure that the penalties are averaged over only the transitions relevant to the parts of the specification and error model under consideration. Other quantitative objectives of interest in this setting include the maximum-lead distance [96] and the sum objective [77].

Richer Error Models. There are significant restrictions placed on the class of error models used in the previous chapters. The major one being that the automata representing error models need to be deterministic. However, some natural ways of penalizing errors in simulation cannot be captured using deterministic error models. For example, consider a request-grant system where each request is to be granted immediately in the same step. The error penalizing scheme penalizes a missed grant with a large one-time penalty, and a delayed grant with small penalties in each step till is grant is finally output. A deterministic error model cannot encode such a penalizing scheme as it will need to guess as soon as a request is not granted in the same step whether the grant will be missed or will just be delayed. Just letting the error models be nondeterministic in the definition of simulation distances will not solve the problem either, as the task of guessing whether the grant is delayed or missed is then shifted to Player 2. While the difficulty can be overcome in certain cases by using quantitative k-pebble simulation distances, or quantitative k-look-ahead simulation distances for a large enough k, the solution is not satisfactory as computing these distances is exponentially more expensive than standard simulation

distances.

One promising solution to this problem is to use error models given by a deterministic, but richer formalism such as cost-register automata [8] instead of standard weighted automata. However, first the problem of computing the value and finding optimal strategies in two player games where the objective is given using a cost-register automata needs to be solved.

Other Extensions. In the simulation distances framework for reactive systems, the simulation distances are defined over the worst-case inputs. However, the systems themselves might operate in an environment where some inputs are more likely than others. In this case, it would be beneficial to define a version of the simulation distances where the inputs are not chosen by Player 1 in the quantitative simulation game, but instead according to a Markov chain that models the environment.

Another possible extension is to define bisimulation distances based on the bisimulation relation instead of the simulation distances. This has already been explored in [81, 69]. However, these works explore mostly the theoretical aspects of the bisimulation distances and do not consider the possible applications to system development.

5.2 Related Work

Specification of Reactive Systems. The formal specification of reactive systems and the associated techniques have a long history which is too involved to recount completely here. Instead, we just recall the major milestones here.

The first work on reactive system synthesis was done in the late 1950s and 1960s, mainly by Büchi and Landweber where it was proved that synthesis of reactive systems from specifications in the S1S sequential calculus is decidable [116, 28]. Further improvements in the specification technology came from the introduction of various temporal logics in the 1970s and 1980s including linear temporal logic [132], computation tree logic [59], and quantified propositional temporal logic [152]. Along with the advent of temporal logics, the corresponding model checking algorithms were discovered [59, 140] and the first practical and industrial model checking tools were developed [101]. The synthesis algorithms for specifications given in various temporal logics were presented in [133], [59], and [134]. For linear temporal logic, the first synthesis algorithms were dependant on the tree automata and were ultimately based on the decidability results of Rabin [79].

On the notion of satisfaction (refinement) relation itself, the main works are those that introduced the simulation relation [127], alternating simulation [12, 11], refinement mappings [1], and other variations such as ready simulation [24] and fair simulation [95]. A hierarchy of various refinement relations was studied and surveyed in [165] and [166].

Other Quantitative Specification Frameworks. There have been several other quantitative specification frameworks in the literature [77, 46, 45, 4, 169, 13, 5]. We discuss some of them in detail here.

Weighted automata [22, 77] and quantitative languages [45, 46] provide a way to assign values to words, and to languages defined by finite-state sys-

tems. Intuitively, all the quantitative specification from the simulation distances framework can be expressed using weighted automata by taking the product of the specification and the error model. However, weighted automata themselves are hard to work with due to the undecidability of several basic questions [46]. Further, when combining the specification and the error model into one weighted automata, one loses the ability to express various important properties of the specification framework such as the triangle inequality. There are several other formalisms that add weights to various notions of automata and transition systems—for example, weighted modal systems [16] and priced timed automata [117].

Another important line of work for is quantitative logics (for example, see [4, 169]). These quantitative specification frameworks have the same advantages and disadvantages with respect to simulation distances framework as classical logic-based specifications have with respect to automata-based specifications, i.e., they are much easier handle and manipulate, but are harder to use with automated techniques.

An important point about most of the other quantitative specification frameworks is that they are value-based rather than distance-based, i.e., the fit of the system is expressed as a value of the system rather than as a distance to a specification. While each approach has its own advantages, we believe that the distance-based frameworks have several important properties that are absent in value-based frameworks. For example, the ability to do hierarchical design, i.e., by doing successive refinements of a design while reasoning separately separately about each refinement step is possible in distance-based frameworks only due to the triangle inequality properties.

Other work on Simulation Distances. There have been several other works on simulation distances and related concepts [81, 144, 145, 146]. The major work in this respect was by Fahrenberg, Legay and Thrane [81] where the authors introduce and study the whole linear time and branching time spectrum of simulation distances. Other works include [144, 145] and [146], where the authors provide an elegant algebraic and co-algebraic definition of simulation distances as for classical simulation relations. However, one major caveat of these definitions is that they apply mainly to the discounted objective and are hard to extend to other objectives. Further, these simulation distances defined by [146] more suitable for processes rather than reactive systems as process replication has a penalty in their definition—in the world of reactive systems, this is equivalent to having a penalty for benign non-determinism.

Quantitative Objectives in Synthesis. It was also observed that quantitative measures can lead to simpler specifications and the use of quantitative objectives to improve the quality of synthesis results was pioneered in [22]. In these works, the aim is to synthesize a system that is correct with respect to a boolean specification and has minimal cost with respect to a quantitative specification given as a weighted automata. In [49], this work was extended to probabilistic environments. One feature that is present in our work, but not in these works is the use of multiple quantitative objectives and the ability to handle incompatible specifications.

The Coverage and Robustness distances. There have been several metrics introduced specifically for for measuring coverage of a test-suite. Some of these are state coverage, transition coverage, and path coverage. However, many of these metrics are syntactic rather than semantic, i.e., they measure the coverage of the syntax of the implementation system rather than the semantics. Some of the exceptions we are aware of are the coverage metrics for verification of temporal logics introduced by Hana Chockler et al [56, 55].

Various boolean and quantitative notions of robustness have been studied in the literature. A full report of these is beyond what can be presented here. Instead, we list a couple of these works that are relevant to our presentation. For transducers, there have been several notions of robustness presented in [148] and [97]. Majumdar et al. introduced a notion of robustness for both discrete systems and controllers in [122] and [157] and have presented symbolic techniques for analysis of robustness of systems.

Incompatible Specifications. The fact that in practice requirements on systems might be inconsistent was recognized in the literature, and several approaches for requirement analysis [130, 21, 94] and requirement debugging [114] were proposed. The problem of an inconsistent specification was approached in [48] by synthesizing additional requirements on the environment so that unrealizability in the specification is avoided.

Synthesis from inconsistent specifications was considered in [78, 75]. Here the conflicts between various components of the specification are resolved by considering priorities for different components, in contrast to our approach of using quantitative measures of correctness. However, it is not possible to express several common resolutions of conflicts such as alternation using priorities.

System Metrics. It has been noted in the literature [71, 162] that boolean notions of correctness may not be suitable for systems that are inherently quantitative, such as real-time systems, or probabilistic systems. Metrics on such quantitative systems have been proposed [71, 162, 163, 43]. There have been several other attempts to give a mathematical semantics to reactive processes which is based on quantitative metrics rather than boolean preorders [161, 69]. In particular for probabilistic processes, it is natural to generalize bisimulation relations to bisimulation metrics [74, 164], and similar generalizations can be pursued if quantities enter not through probabilities but through discounting [70] or continuous variables [30] (this work uses the Skorohod metric on continuous behaviors to measure the distance between hybrid systems). In contrast, in our work, we consider distances between purely discrete finite-state (non-probabilistic, untimed) systems, and the quantitative aspect of the distance function arises only from the comparison of the behavior of the two systems between which we measure the distance.

Software metrics measure properties such as lines of code, depth of inheritance (in an object-oriented language), number of bugs in a module or the time it took to discover the bugs (see for example [83, 120]) have been used to measure system quality. These functions measure syntactic properties of the source code, and are fundamentally different from our distances that capture the difference in the behavior (semantics) of programs.

5.3 Conclusion

We have motivated the notion of distance between between systems, and introduced quantitative simulation games as a framework for measuring such distances. We presented three distances, namely, the correctness, coverage, and robustness distances and proved various properties of simulation distances that enable different system verification methodologies. We applied simulation distances to the problem of synthesizing from incompatible specifications and gave an algorithm to synthesize the implementation system that comes closest to satisfying multiple incompatible specifications. Further, we presented several case studies that illustrate the use of simulation distances in different settings in the process of system development.

Part II

Quantities as Measurement

In this part, we focus on extending some classical techniques used in verification and synthesis to quantitative properties. We mainly consider the quantitative properties of execution time and energy consumption. However, the techniques presented in Chapter 7[Pg. 110] apply to the much more general class of monotonic quantitative properties. Below, we describe the contributions of this part briefly.

Performance-aware Synthesis for Concurrency. In partial-program synthesis for concurrency, a programmer writes a the functional parts of the program, while leaving out the synchronization choices for the synthesizer to automatically fill in. There has been a large amount of work on this paradigm in recent years [153, 53, 168]. However, none of these works consider the performance of the synthesized solution program. This is especially important as in many cases, there is an obvious solution to the synthesis for concurrency problem which is to acquire a global lock before executing each thread.

Most synthesis for concurrency techniques in the literature use heuristics based on minimizing the size of the critical sections to choose between different solutions. However, in many common classes of programs smaller critical sections are not necessarily better performing. Further, in some cases such as optimistic concurrency, the size of the critical sections are fixed while the performance depends mainly on other parameters.

We present an algorithm for synthesis for concurrency which produces not just any correct solution, but the optimally performing one. The input to the algorithm consists of a performance model in addition to the partial program. The performance model is a weighted automata describing the cost of each operation that affects performance. We reduce the problem of finding the optimal correct program to the problem of finding the optimal memoryless strategy in a partial-information limit-average safety game. However, we also show that this problem is computationally hard, and instead, present several heuristics to speed up the search for the optimal strategy. The heuristics are based on abstraction, counter-example guided elimination of partial-strategies, and faster algorithms for solving Markov chains that exploit the structure of the programs.

Quantitative Abstraction Refinement. We present abstraction and abstraction refinement techniques for quantitative properties. The class of properties considered here are monotonic properties, i.e., the class of properties where increasing the weight of one transition in the system, and adding more behaviors to the system can only increase the value of the property. Most common quantitative properties such as worst-case execution time and energy consumption fall into this class.

We present two kinds of abstractions for such quantitative properties. The first one is the state-based *ExistMax* abstraction. Intuitively, the *ExistMax* abstraction is a straight-forward extension of the classical state-based existential abstraction. However, quantitative properties are usually path-based rather than state-based, i.e., the value of the property depends on the whole trace rather that states of the trace. Hence, a more suitable abstraction for quantitative properties is the segment-based *PathBound* abstraction. Here, the abstract object being reasoned about are not sets of states as in state-based abstractions, but parts of execution traces. We provide automatic counter-example

guided abstraction refinement algorithms for both the *ExistMax* abstraction and *PathBound* abstractions.

We apply the segment-based abstractions to the problem of computing the worst-case execution time of programs. We show that our techniques based on segment-based quantitative abstractions can give tighter bounds on the worst-case execution time than the standard tools. Further, we also show how many commonly used ad-hoc program transformations used as pre-processing steps in worst-case execution time analysis arise naturally as refinements of segment-based abstractions.

Model Checking of Battery Transition Systems. As described in Chapter 1[Pg. 6], most formalisms for modelling systems interacting with energy sources treat the state of the energy source as one number, i.e., the amount of energy remaining in the energy source. However, real batteries exhibit many behaviours that are cannot be exhibited by ideal sources that can be modelled using just the amount of energy remaining in the source.

We introduce a formal model of systems interacting with batteries, battery transition systems which is based on the physical battery model KiBaM [123]. Here, the energy in the battery is divided into two *tanks* — the available charge tank and the bound charge tank. The energy in the available charge tank is immediately available for use, while the energy in the bound charge tank is only available after diffusion into the available charge tank.

Battery transition systems do not fall into any previously known decidable class of infinite state transition systems. The closest related decidable class of systems are well-structured transition systems [86]. In fact, a partial order compatible with the transition relation can be defined for battery transition systems as in the case of well-structured transition systems. However, this partial order is not well-founded in general.

The model checking algorithms for battery transition systems are reminiscent of the forward exploration algorithms for well-structured transition systems in general, and specifically the Karp-Miller tree and Petri-nets. However, in the case of battery transition systems, unlike in the generic well-structured transition system case, we can accelerate cycles in the systems precisely and obtain decidability.

Chapter 6

Quantitative Synthesis for Concurrency

We present an algorithmic method for the quantitative, performance-aware synthesis of concurrent programs. The input consists of a nondeterministic *partial program* and of a *parametric performance model*. The nondeterminism allows the programmer to omit which (if any) synchronization construct is used at a particular program location. The performance model, specified as a weighted automaton, can capture system architectures by assigning different costs to actions such as locking, context switching, and memory and cache accesses. The quantitative synthesis problem is to automatically resolve the nondeterminism of the partial program so that both correctness is guaranteed and performance is optimal. As is standard for shared memory concurrency, correctness is formalized "specification free", in particular as race freedom or deadlock freedom. For worst-case (average-case) performance, we show that the problem can be reduced to 2-player graph games (with probabilistic transitions) with quantitative objectives. While we show, using game-theoretic methods, that the synthesis problem is computationally hard, we present an algorithmic method and an implementation that works efficiently for concurrent programs and performance models of practical interest. We have implemented a prototype tool and used it to synthesize finite-state concurrent programs that exhibit different programming patterns, for several performance models representing different architectures.

6.1 Motivation

A promising approach to the development of correct concurrent programs is *partial program synthesis*. The goal of the approach is to allow the programmer to specify a part of her intent declaratively, by specifying which conditions, such as linearizability or deadlock freedom, need to be maintained. The synthesizer then constructs a program that satisfies the specification (see, for example, [154, 153, 168]). However, quantitative considerations have been largely missing from previous frameworks for partial synthesis. In particular, there has been no possibility for a programmer to ask the synthesizer for a program that is not only correct, but also *efficient* with respect to a specific performance model.

```
 1: while(true) {
 2:     lver=gver; ldata=gdata;
 3:     n  = choice(1..10);
 4:     i = 0;
 5:     while (i < n) {
 6:         work(ldata); i++;
 7:     }
 8:     if (trylock(lock)) {
 9:         if (gver==lver) {
10:             gdata = ldata;
11:             gver = lver+1;
12:             unlock(lock);
13:         } else {
14:             unlock(lock)
15:         }
16:     }
17: }
```

Figure 6.1: Example 2

We show that providing a quantitative performance model that represents the architecture of the system on which the program is to be run can considerably improve the quality and, therefore, potential usability of synthesis.

Motivating examples. *Example 1.* Consider a *producer-consumer* program, where k producer and k consumer threads access a buffer of n cells. The programmer writes a partial program implementing the procedures that access the buffer as if writing the sequential version, and specifies that at each control location a global lock or a cell-local lock can be taken. It is easy to see that there are at least two different ways of implementing correct synchronization. The first is to use a global lock, which locks the whole buffer. The second is to use cell-local locks, with each thread locking only the cell it currently accesses. The second program allows more concurrent behavior and is better in many settings. However, if the cost of locks is high (relative to the other operations), the global-locking approach is more efficient. In our experiments on a desktop machine, the global-locking implementation out-performed the cell-locking implementation by a factor of 3 in certain settings.

Example 2. Consider the program in Figure 6.1[Pg. 91]. It uses classic conflict resolution mechanism used for optimistic concurrency. The shared variables are `gdata`, on which some operation (given by the function `work()`) is performed repeatedly, and `gver`, the version number. Each thread has local variables `ldata` and `lver` that store local copies of the shared variables. The data is read (line 2) and operated on (line 6) without acquiring any locks. When the data is written back, the shared data is locked (line 8), and it is checked (using the version number, line 9) that no other thread has changed the data since it has been read. If the global version number has not changed, the new value is written to the shared memory (line 10), and the global version number is increased (line 11). If the global version number has changed, the whole procedure

is retried. The number of operations (calls to `work`) performed optimistically without writing back to shared memory can influence the performance significantly. For approaches that perform many operations before writing back, there can be many retries and the performance can drop. On the other hand, if only a few operations are performed optimistically, the data has to be written back often, which also can lead to a performance drop. Thus, the programmer would like to leave the task of finding the optimal number of operations to be performed optimistically to the synthesizer. This is done via the choice statement (line 4).

The partial program resolution problem. Our aim is to synthesize concurrent programs that are both correct and optimal with respect to a performance model. The input for partial program synthesis consists of (1) a finite-state partial program, (2) a performance model, (3) a model of the scheduler, and (4) a correctness condition. A *partial program* is a finite-state concurrent program that includes nondeterministic choices which the synthesizer has to resolve. A program is *allowed* by a partial program if it can be obtained by resolving the nondeterministic choices. The second input is a *parametric performance model*, given by a weighted automaton. The automaton assigns different costs to actions such as locking, context switching, and memory and cache access. It is a flexible model that allows the assignment of costs based on past sequences of actions. For instance, if a context switch happens soon after the preceding one, then its cost might be lower due to cache effects. Similarly, we can use the automaton to specify complex cost models for memory and cache accesses. The performance model can be fixed for a particular architecture and, hence, need not be constructed separately for every partial program. The third input is the *scheduler*. Our schedulers are state-based, possibly probabilistic, models which support flexible scheduling schemes (e.g., a thread waiting for a long time may be scheduled with higher probability). In performance analysis, average-case analysis is as natural as worst-case analysis. For the average-case analysis, probabilistic schedulers are needed. The fourth input, the *correctness condition*, is a safety condition. We use "specification-free" conditions such as data-race freedom or deadlock-freedom. The output of synthesis is a program that is (a) allowed by the partial program, (b) correct with respect to the safety condition, and (c) has the best performance of all the programs satisfying (a) and (b) with respect to the performance and scheduling models.

Quantitative games. We show that the partial program resolution problem can be reduced to solving *imperfect information* (stochastic) graph games with quantitative (limit-average or mean-payoff) objectives. Traditionally, imperfect information graph games have been studied to answer the question of existence of general, *history-dependent* optimal strategies, in which case the problem is undecidable for quantitative objectives [73]. We show that the partial program resolution problem gives rise to the question (not studied before) whether there exist *memoryless* optimal strategies (i.e. strategies that are independent of the history) in imperfect information games. We establish that the memoryless problem for imperfect information games (as well as imperfect information stochastic games) is NP-complete, and prove that the partial program resolution problem is computationally hard for both average-case and worst-case

performance based synthesis. We present several techniques that overcome the theoretical difficulty of hardness in cases of programs of practical interest: (1) First, we use a lightweight static analysis technique for efficiently eliminating parts of the strategy tree. This reduces the number of strategies to be examined significantly. We then examine each strategy separately and, for each strategy, obtain a (perfect information) Markov decision process (MDP). For MDPs, efficient strategy improvement algorithms exist, and require solving Markov chains. (2) Second, Markov chains obtained from concurrent programs typically satisfy certain progress conditions, which we exploit using a forward propagation technique together with Gaussian elimination to solve Markov chains efficiently. (3) Our third technique is to use an abstraction that preserves the value of the quantitative (limit-average) objective. An example of such an abstraction is the classical data abstraction.

Experimental results. In order to evaluate our synthesis algorithm, we implemented a tool and applied it to four finite-state examples that illustrate basic patterns in concurrent programming. In each case, the tool automatically synthesized the optimal correct program for various performance models that represent different architectures. For the producer-consumer example, we synthesized a program where two producer and two consumer threads access a buffer with four cells. The most important parameters of the performance model are the cost l of locking/unlocking and the cost c of copying data from/to shared memory. If the cost c is higher than l (by a factor 100:1), then the fine-grained locking approach is better (by 19 percent). If the cost l is equal to c, then the coarse-grained locking is found to perform better (by 25 percent). Referring back to the code in Figure 6.1[Pg. 91], for the optimistic concurrency example and a particular performance model, the analysis found that increasing n improves the performance initially, but after a small number of increments the performance started to decrease. We measured the running time of the program on a desktop machine and observed the same phenomenon.

6.2 The Quantitative Synthesis Problem

6.2.1 Partial Programs

In this section we define threads, partial programs, programs and their semantics. We start with the definitions of guards and operations.

Guards and operations. Let L, G, and I be finite sets of variables (representing local, global (shared), and input variables, respectively) ranging over finite domains. A *term* t is either a variable in L, G, or I, or t_1 *op* t_2, where t_1 and t_2 are terms and *op* is a binary operator. Formulas are defined by the following grammar, where t_1 and t_2 are terms and *op* is a relational operator: $e := t_1 op\ t_2 | e \wedge e | \neg e$. *Guards* are boolean formulae over L, G, and I. *Operations* are simultaneous assignments to variables in $L \cup G$, where each variable is assigned a term over L, G, and I.

Threads. A *thread* is a tuple $\langle Q, L, G, I, \delta, \rho_0, q_0 \rangle$, with: (a) a finite set of control locations Q and an initial location q_0; (b) L, G and I are as before;

(c) an initial valuation of the variables ρ_0; and (d) a set δ of transition tuples of the form (q, g, a, q'), where q and q' are locations from Q, and g and a are *guards* and *operations* over variables in L, G and I.

The set of locations $Sk(c)$ of a thread $c = \langle Q, L, G, I, \delta, \rho_0, q_0 \rangle$ is the subset of Q containing exactly the locations where δ is non-deterministic, i.e., locations where there exists a valuation of variables in L, G and I, for which there are multiple transitions whose guards evaluate to true.

Partial programs and programs. A *partial program* M is a set of threads that have the same set of global variables G and whose initial valuation of variables in G is the same. Informally, the semantics of a partial program is a parallel composition of threads. The set G represents the shared memory. A *program* is a partial program, in which the set $Sk(c)$ of each thread c is empty. A program P is *allowed* by a partial program M if it can be obtained by removing the outgoing transitions from sketch locations of all the threads of M, so that the transition function of each thread becomes deterministic.

The guarded operations allow us to model basic concurrency constructs such as locks (for example, as variables in G and locking/unlocking is done using guarded operations) and compare-and-set. As partial program defined as a collection of fixed threads, thread creation is not supported.

Semantics. The semantics of a partial program M is given in terms of a transition system (denoted as $\mathsf{Tr}(M)$) which we describe informally below. Given a partial program M with n threads, let $\mathcal{C} = \{1, \ldots, n\}$ represent the set of threads of M.

- *State space.* Each state $s \in S$ of $\mathsf{Tr}(M)$ contains input and local variable valuations and locations for each thread in $\mathcal{C}$, and a valuation of the global variables. In addition, it contains a value $\sigma \in \mathcal{C} \cup \{*\}$, indicating which (if any) thread is currently scheduled. The initial state contains the initial locations of all threads and the initial valuations ρ_0, and the value $*$ indicating that no thread is currently scheduled.
- *Transitions.* The transition function Δ defines interleaving semantics for partial programs. There are two types of transitions: thread transitions, that model one step of a scheduled thread, and environment transitions, that model input from the environment and the scheduler. For every $c \in \mathcal{C}$, there exists a thread transition labeled c from a state s to a state s' if and only if there exists a transition (q, g, a, q') of c such that (i) $\sigma = c$ in s (indicating that c is scheduled) and $\sigma = *$ in s', (ii) the location of c is q in s and q' in s', (iii) the guard g evaluates to true in s, and (iv) the valuation of local variables of c and global variables in s is obtained from the valuation of variables in s' by performing the operation a. There is an environment transition labeled c from state s to state s' in $\mathsf{Tr}(M)$ if and only if (i) the value σ in s is $*$ and the value σ in s' is c and (ii) the valuations of variables in s and s' differ only in input variables of the thread c.

6.2.2 The performance model

We define a flexible and expressive performance model via a weighted automaton with limit-average objective that specifies costs of actions. A *performance*

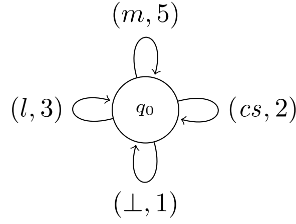


Figure 6.2: Example Performance Automaton

automaton W is a weighted automaton $W = (S_W, \Sigma_W, \Delta_W, s_\iota, v)$ where each component is as described in Chapter 2[Pg. 16]. The labels in Σ_W represent (concurrency related) actions that incur costs, while the values of the function v specify these costs. The actions in Σ_W are matched with the actions performed by the system to which the performance measures are applied. A special action $\bot \in \Sigma$ denotes that none of the tracked actions occurred. The costs that can be specified in this way include for example the cost of locking, the access to the (shared) main memory or the cost of context switches.

An example specification that uses the costs mentioned above is the automaton W in Figure 6.2[Pg. 95]. The automaton describes the costs for locking (l), context switching (cs), and main memory access (m). Specifying the costs via a weighted automaton is more general than only specifying a list of costs. For example, automaton based specification enables us to model a cache, and the cost of reading from a cache versus reading from the main memory, as shown in Figure 6.6[Pg. 107] in Section 6.5[Pg. 104]. Note that the performance model can be fixed for a particular architecture. This eliminates the need to construct a performance model for the synthesis of each partial program.

6.2.3 The partial program resolution problem

Schedulers. Informally, a *scheduler* has a finite set of internal memory states Q_{Sch}. At each step, it considers all the active threads and chooses one either (i) nondeterministically (nondeterministic schedulers) or (ii) according to a probability distribution (probabilistic schedulers), which depends on the current internal memory state. Formally, a scheduler can be modelled as a Markov chain where the transitions are labelled with threads of the program.

Composing a program with a scheduler and a performance model. In order to evaluate the performance of a program, we need to take into account the scheduler and the performance model. Given a program P, a scheduler Sch, and a performance model W, we construct a WPTS, denoted $\mathsf{Tr}(P, \text{Sch}, W)$, with a weight function v as follows. A state s of $\mathsf{Tr}(P, \text{Sch}, W)$ is composed of a state of the transition system of P ($\mathsf{Tr}(P)$), a state of the scheduler Sch and a state of the performance model W. The transition function matches environment transitions of $\mathsf{Tr}(P)$ with the scheduler transitions (which allows the scheduler to schedule threads) and it matches thread transitions with the performance model transitions. The weight function v assigns costs to edges as given by the weighted automaton W. Furthermore, as the limit average objective is defined only for infinite executions, for terminating safe executions of the program we

add an edge back to the initial state. The value of the limit average objective function of the infinite execution is the same as the average over the original finite execution. Note that the performance model can specify a locking cost, while the program model does not specifically mention locking. We thus need to specifically designate which shared memory variables are used for locking.

Correctness. We restrict our attention to safety conditions for correctness. We illustrate how various correctness conditions for concurrent programs can be modelled as Safety objectives: (a) *Data-race freedom.* Data-races occur when two or more threads access the same shared memory location and one of the accesses is a write access. We can check for absence of data-races by denoting as unsafe states those in which there exist two enabled transitions (with at least one being a write) accessing a particular shared variable, from different threads. (b) *Deadlock freedom.* One of the major problems of synchronizing programs using blocking primitives such as locks is that deadlocks may arise. A deadlock occurs when two (or more) threads are waiting for each other to finish an operation. Deadlock-freedom is a safety property. The unsafe states are those where there exists two or more threads with each one waiting for a resource held by the next one.

Value of a program and of a partial program. For P, Sch, W as before and Safety_B is a safety objective, we define the value of the program using the composition of P, Sch and W as: $\mathit{ValProg}(P, \mathrm{Sch}, W, \mathrm{Safety}_B) = \mathit{Val}(\mathsf{Tr}(P, \mathrm{Sch}, W), v, \mathrm{Safety}_B)$. For be a partial program M, let $\mathcal{P}$ be the set of all allowed programs. The value of M, $\mathit{ValParProg}(M, \mathrm{Sch}, W, \mathrm{Safety}_B) = \min_{P \in \mathcal{P}} \mathit{ValProg}(P, \mathrm{Sch}, W, \mathrm{Safety}_B)$.

Partial Program resolution problem. The central technical questions we address are as follows: (1) The *partial program resolution optimization problem* consists of a partial program M, a scheduler Sch, a performance model W and a safety condition Safety_B, and asks for a program P allowed by the partial program M such that the value $\mathit{ValProg}(P, \mathrm{Sch}, W, \mathrm{Safety}_B)$ is minimized. Informally, we have: (i) if the value $\mathit{ValParProg}(M, \mathrm{Sch}, W, \mathrm{Safety}_B)$ is ∞, then no safe program exists; (ii) if it is finite, then the answer is the optimal safe program, i.e., a correct program that is optimal with respect to the performance model. The *partial program resolution decision problem* consists of the above inputs and a rational threshold λ, and asks whether $\mathit{ValParProg}(M, \mathrm{Sch}, W, \mathrm{Safety}_B) \leq \lambda$.

6.3 Quantitative Games on Graphs

Games for synthesis of controllers and sequential systems from specifications have been well studied in literature. We show how the partial program resolution problems can be reduced to quantitative imperfect information games on graphs. We also show that the arising technical questions on game graphs is different from the classical problems on quantitative graph games.

Recall from Chapter 2[Pg. 16] the definitions related $2\frac{1}{2}$-player, observation based strategies, memoryless strategies, and values of strategies.

Given a game graph $\mathcal{G}$, an objective f, an observation mapping for Player 1 O and a rational threshold $q \in \mathbb{Q}$, the general decision problem (resp. memoryless decision problem) asks if there is a observation-based Player 1 strategy (resp. observation-based memoryless strategy) ϕ_1 with $\sup_{\phi_2 \in \Phi_2} \mathbb{E}(f(\mathit{Outcomes}(\phi_1, \phi_2))) \leq q$. Similarly, the value problem (memoryless value problem) is to compute $\inf_{\phi_1 \in \Phi_1} \mathit{ValGame}(f, \mathcal{G}, \phi_1)$ ($\min_{\phi_1 \in \Phi_1^M} \mathit{ValGame}(f, \mathcal{G}, \phi_1)$ resp.). Traditional game theory study always considers the general decision problem which is undecidable for limit-average objectives [73] in imperfect information games.

Theorem 6.1. *[73] The decision problems for* LimAvg *and* LimAvg-Safety *objectives are undecidable for imperfect information* $2\frac{1}{2}$*-player and imperfect information* 2*-player games.*

However, we show here that the partial program resolution problems reduce to the memoryless decision problem for imperfect information games.

Theorem 6.2. *Given a partial program* M*, a scheduler* Sch*, a performance model* W*, and a correctness condition* ϕ*, we construct an exponential-size* ImpIn $2\frac{1}{2}$*-player game graph* $\mathcal{G}_M^p$ *with a* LimAvg-Safety *objective such that the memoryless value of* $\mathcal{G}_M^p$ *is equal to* $\mathit{ValParProg}(M, \mathrm{Sch}, W, \mathrm{Safety})$*.*

The proof relies on a construction of a game graph similar to the product of a program, a scheduler and a performance model. Player 2 chooses the thread to be scheduled and Player 1 resolves the nondeterminism when the scheduled thread c is in a location in $\mathit{Sk}(c)$. The crucial detail is that Player 1 can observe only the location of the thread and not the valuations of the variables. This partial information gives us a one-one correspondence between the memoryless strategies of Player 1 and programs allowed by the partial program.

Proof. The proof relies on the construction of an imperfect information game graph, denoted $\mathcal{G}(M, \mathrm{Sch}, W)$, in which fixing a memoryless strategy ϕ_1 for Player 1 yields a WPTS $\mathsf{Tr}(P_{\phi_1}, \mathrm{Sch}, W)$ with weight function v that corresponds to the product of a program P_{ϕ_1} allowed by the partial program M, composed with the scheduler Sch and the performance model W. The construction of this game graph is similar to the construction of the product of a program, scheduler and performance model, but with a partial program replacing the program. Due to the nondeterministic transition function of the partial program, there will exist extra nondeterministic choices in the WPTS (in addition to the choice of inputs). This nondeterminism is resolved by Player 1 choices and the nondeterminism due to input (and possibly scheduling) is resolved by Player 2 choices. We refer to this game as the *program resolution game*.

The crucial point of the construction is the observations, i.e., the information about the state that is visible to Player 1. Since Player 1 is to resolve the nondeterminism from the partial program, he is allowed only to observe the scheduled thread and its current location. He may choose a set of transitions, from that location, such that only one of the set is enabled for any valuation of the variables. The formal description of the reduction of partial program resolution to imperfect information games is as follows.

- *State space.* Analogous to the construction of $\mathsf{Tr}(P, \mathrm{Sch}, W)$, a state in the state space of $\mathcal{G}(M, \mathrm{Sch}, W)$ is a tuple $(s, q_{\mathrm{Sch}}, q_W)$ where s, q_{Sch} and q_W are states of $\mathsf{Tr}(M)$, Sch and W, respectively.

- *Player-1 and Player-2 partition.* The state is a Player 1 state if s is labelled with a scheduled thread, and a Player 2 state if s has no thread scheduled and is labelled with a $*$.
- *Observation and observation mapping.* The set of observations O is the set of locations from all the threads of M along with a $\bot$ element, i.e., $O = \{\bot\} \cup \{(t,q) | t$ is a thread of M and q is a partial program location of $t\}$. All Player 2 states are mapped to $\bot$ by η. Player 1 states with thread t scheduled and thread t in location q are mapped to (t,q) by η.
- *Enabled actions and transitions.* Suppose $(s, q_{\mathrm{Sch}}, q_W)$ is a Player 1 state with $\eta((s, q_{\mathrm{Sch}}, q_W)) = (t,q)$. Any action a enabled in this state is a set of transitions of thread t from state q such that only one of them is enabled for any valuation of local, global and input variables. On choosing action a in $(s, q_{\mathrm{Sch}}, q_W)$, the control moves to the state $(s', q'_{\mathrm{Sch}}, q'_W)$ where s' is the state obtained by executing the unique enabled transition from a in s. The states q'_{Sch} and q'_W are as in $\mathsf{Tr}(P, \mathrm{Sch}, W)$. The set of Player 2 actions and transitions are as in $\mathsf{Tr}(P, \mathrm{Sch}, W)$.
- *Initial state.* The initial state of $\mathcal{G}(M, \mathrm{Sch}, W)$ is the tuple of initial states of M, Sch and W.

To complete the proof, we show that given a memoryless Player 1 strategy ϕ_1, there exists a program P_{ϕ_1} allowed by M such that $\mathsf{Tr}(P_{\phi_1}, \mathrm{Sch}, W)$ corresponds to the MDP obtained by fixing ϕ_1 in $\mathcal{G}(M, \mathrm{Sch}, W)$ and vice-versa.

Given a program P_{ϕ_1} allowed by the partial program, we construct a memoryless ϕ_1 as follows: $\phi_1((t,q))$ is the action consisting of the set of transitions from location q in thread t in P_{ϕ_1}. As P_{ϕ_1} is deterministic, only one of them will be enabled for a valuation of the variables. Similarly, given a memoryless Player 1 strategy, we construct P_{ϕ_1} by preserving only those transitions from location q of thread t which are present in $\phi_1((t,q))$. From the above construction we conclude the desired correspondence. □

6.3.1 Complexity of ImpIn Games and partial-program resolution

We establish complexity bounds for the relevant memoryless decision problems and use them to establish upper bounds for the partial program resolution problem. First, we state a theorem on complexity of MDPs.

Theorem 6.3. *[85] The memoryless decision problem for* LimAvg-Safety *objectives can be solved in polynomial time for MDPs.*

Theorem 6.4. *The memoryless decision problems for* Safety*,* LimAvg*, and* LimAvg-Safety *objectives are* NP*-complete for* ImpIn $2\frac{1}{2}$*- and* ImpIn 2*-player game graphs.*

For the lower bound we show a reduction from 3SAT problem and for the upper bound we use memoryless strategies as polynomial witness and Theorem 6.3[Pg. 98] for polynomial time verification procedure.

Lemma 6.5. *The memoryless decision problem for* ImpIn*-2-player game graphs with* Safety *and* LimAvg *objectives are* NP*-hard.*

Proof. We first show NP-hardness for safety objectives.

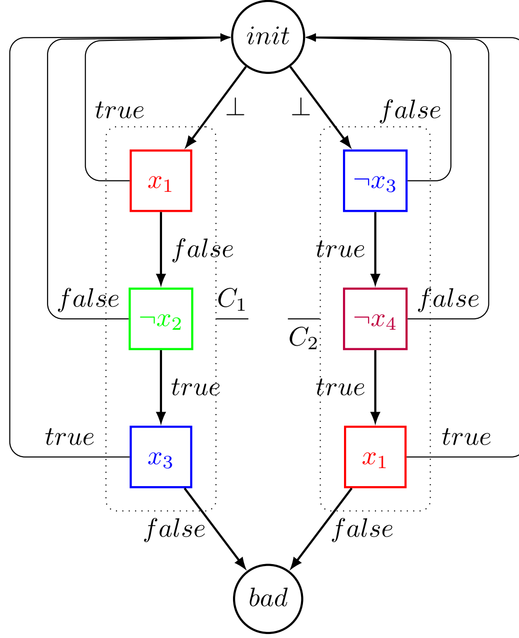


Figure 6.3: 3-SAT to memoryless imperfect information Safety games

(NP-hardness). We will show that the memoryless decision problem for ImpIn-2-player safety game is NP-hard by reducing the 3-SAT problem. Given a 3-SAT formula Φ over variables $x_1, x_2, \ldots x_N$, with clauses $C_1, C_2, \ldots C_K$, we construct an imperfect information game graph with $N+1$ observations and $3K+2$ states such that Player 1 has a memoryless winning strategy from the initial state if and only if Φ is satisfiable. The construction is described below:

- The states of the game graph are $\{init\} \cup \{s_{i,j} \mid i \in [1, K] \wedge j \in \{1, 2, 3\}\} \cup \{bad\}$.
- The observations and the observation mapping are as follows: *init* and *bad* are mapped with observation 0, and $s_{i,j}$ is mapped with observation k if the jth variable of the C_i clause is x_k or $\neg x_k$.
- *init* and *bad* are Player 2 states and all other states are Player 1 states.
- The actions and transition function of the game graph are as follows:
    1. For all $i \in [0, K]$, there is a transition from *init* to $s_{i,1}$ on the action $\perp$.
    2. If the jth literal of clause C_i is x_k, then there are two actions enabled ($true$ and $false$) and there is a transition from $s_{i,j}$ to *init* on $true$ and to $s_{i+1,j}$ on $false$ (for $j \in \{1, 2\}$). For $j = 3$, the transition on $true$ leads to *init* and the transition on $false$ leads to *bad*.
    3. If the jth literal of clause C_i is $\neg x_k$, there is a transition from $s_{i,j}$ to *init* on $false$ and to $s_{i+1,j}$ on $true$ (for $j \in \{1, 2\}$). For $j = 3$, the transition on $false$ leads to *init* and the edge on $true$ leads to *bad*.

- The objective for Player 1 is to avoid reaching *bad* and the objective for Player 2 is to reach *bad*.

Intuitively, Player 2 chooses a clause C_i in the initial state *init*. Player 1 then plays according to her memoryless strategy from each of the states $s_{i,j}$. If the action $a \in \{true, false\}$ chosen in $s_{i,j}$ makes the literal at position j in clause C_i true, control goes back to *init*. Otherwise, the control goes to the next $s_{i,j+1}$. If the choices at all three $s_{i,j}$'s make the corresponding literal false, the control goes to *bad*. The game graph structure is illustrated in Figure 6.3[Pg. 99].

Given a truth value assignment of x_i's such that Φ is satisfied, we can construct a memoryless strategy of Player 1 which chooses the action at observation i same as the valuation of x_i, and the memoryless strategy is winning for Player 1. In every triple of $s_{i,j}$'s at least one of the edges dictated by this strategy lead to *init*. If that were not the case, the corresponding clause would not have been satisfied. Given a winning memoryless strategy ϕ_1, the valuation of x_i's which assigns the $\phi_1(i)$ to x_i satisfies each clause C_k in Φ. This follows from a similar argument as above. Hence the hardness result follows.

The above reduction is slightly modified to show that the LimAvg memoryless decision problem is also NP-hard. This can be done by adding a self loop on state *bad* with weight 1 and attaching the weight 0 to all other edges. Now, Player 1 can obtain a value less than 1 if and only if she has a memoryless winning strategy in the Safety game. The desired result follows. □

Lemma 6.6. *The memoryless decision problem for* LimAvg-Safety *objectives for* ImpIn $2\frac{1}{2}$*-player game graphs is in* NP*.*

Proof. Given a memoryless winning strategy for a Player 1 in a ImpIn $2\frac{1}{2}$-player game graph, the verification problem is equivalent to solving for the same objective on the MDP obtained by fixing the strategy for Player 1. Hence the memoryless strategy is the polynomial witness, and Theorem 6.3[Pg. 98] provides the polynomial time verification procedure to prove the desired result. □

Lemma 6.5[Pg. 98] and Lemma 6.6[Pg. 100] gives us Theorem 6.4[Pg. 98].

Remark 6.7. *The* NP*-completeness of the memoryless decision problems rules out the existence of the classical strategy improvement algorithms as their existence implies existence of randomized sub-exponential time algorithms (using the techniques of [20]), and hence a strategy improvement algorithm would imply a randomized sub-exponential algorithm for an* NP*-complete problem.*

6.4 Practical Solutions for Partial-Program Resolution

Algorithm 1 Strategy Elimination

Input: M: partial program;
  W: performance model;
  Sch: scheduler;
  Safety: safety condition
Output: *Candidates*: Strategies
  StrategySet $\leftarrow$ `CompleteTree`(M)
  {A complete strategy tree}
  Candidates $\leftarrow \emptyset$
  while *StrategySet* $\neq \emptyset$ **do**
    Choose *Tree* from *StrategySet*
    $\phi_1 \leftarrow$`Root`(*Tree*)
    if `PartialCheck`(ϕ_1, Safety) **then**
      StrategySet =
        StrategySet $\cup$ `children`(*Tree*)
    if *Tree* is singleton **then**
      Candidates = *Candidates* $\cup \{\phi_1\}$
  return *Candidates*

We present practical solutions for the computationally hard (Pspace-complete) partial-program resolution problem.

Strategy elimination. We present the general strategy enumeration scheme for partial program resolution. We first introduce the notions of a partial strategy and strategy tree.
Partial strategy and strategy trees. A *partial memoryless strategy* for Player 1 is a partial function from observations to actions. A *strategy tree* is a finite branching tree labelled with partial memoryless strategies of Player 1 such that: (a) Every leaf node is labelled with a complete strategy; (b) Every node is labelled with a unique partial strategy; and (c) For any parent-child node pair, the label of the child (${\phi_1}^c$) is a proper extension of the label of parent (${\phi_1}^p$), i.e., ${\phi_1}^c(o) = {\phi_1}^p(o)$ when both are defined and the domain of ${\phi_1}^p$ a proper superset of ${\phi_1}^c$. A complete strategy tree is one where all Player 1 memoryless strategies are present as labels.

In the strategy enumeration scheme, we maintain a set of candidate strategy trees and check each one for partial correctness. If the root label of the tree fails the partial correctness check, then remove the whole tree from the set. Otherwise, we replace it with the children of the root node. The initial set is a single complete strategy tree. In practice, the choice of this tree can be instrumental in the efficiency of partial correctness checks. Trees which first fix the choices that help the partial correctness check to identify an incorrect partial strategy are more useful. The partial program resolution scheme is shown in Algorithm 1[Pg. 101].

The `PartialCheck` function checks for the partial correctness of partial strategies, and returns "Incorrect" if it is able to prove that all strategies com-

Algorithm 2 Synthesis Scheme

Input: M: partial program;
  W: performance model;
  Sch: a scheduler;
  Safety: a safety condition
Output: P: correct program or $\perp$
  $\mathit{Candidates} \leftarrow$ `StrategyElimination`$(M, \mathrm{Sch}, W, \mathrm{Safety})$
  $\mathit{StrategyValues} \leftarrow \emptyset$
  while $\mathit{Candidates} \neq \emptyset$ **do**
    Pick ϕ_1 from $\mathit{Candidates}$
    $\mathcal{G}_{\phi_1} \leftarrow \mathcal{G}(M, \mathrm{Sch}, W)$ with ϕ_1 fixed
    $\mathcal{G}^*_{\phi_1} \leftarrow$ `Abstract`$(\mathcal{G}_{\phi_1})$
    $\mathit{Valid} \leftarrow$ `SoftwareModelCheck`$(\mathcal{G}^*_{\phi_1}, \mathrm{Safety})$
    if Valid **then**
      $\mathit{Value} \leftarrow$ `SolveMDP`$(\mathcal{G}^*_{\phi_1})$
      $\mathit{StrategyValues} \leftarrow \mathit{StrategyValues} \cup \{\phi_1 \mapsto \mathit{Value}\}$
  if $\mathit{StrategyValues} = \emptyset$ **then**
    return $\perp$
  else
    $\mathit{OptimalStrategy}$ = `minimum`$(\mathit{StrategyValues})$
    return M with $\mathit{OptimalStrategy}$ strategy fixed

patible with the input are unsafe, or it returns "Don't know". In practice, for the partial correctness checks the following steps can be used: (a) checking of lock discipline to prevent deadlocks; and (b) simulation of the partial program on small inputs; The result of the scheme is a set of candidate strategies for which we evaluate full correctness and compute the value.

The result of the scheme is a set of candidate strategies for which we evaluate full correctness and compute the value. The algorithm is shown in Algorithm 2[Pg. 102]. In the algorithm, the procedures `SoftwareModelCheck`, `Abstract` and `SolveMDP` are of special interest. The procedure `Abstract` abstracts an MDP preserving the LimAvg-Safety properties as described in the following paragraphs. The `SolveMDP` procedure uses the optimizations described below to compute the LimAvg value of an MDP efficiently. The Safety conditions are checked by `SoftwareModelCheck` procedure. It might not explicitly construct the states of the MDP, but may use symbolic techniques to check the Safety property on the MDP. It is likely that further abstraction of the MDP may be possible during this procedure as we need abstractions which preserve Safety, and $\mathcal{G}^*_{\phi_1}$ is abstracted to preserve both Safety and LimAvg values.

Evaluation of a memoryless strategy. Fixing a memoryless Player 1 strategy in a ImpIn $2\frac{1}{2}$-player game for partial program resolution gives us (i) a non-deterministic transition system in the case of a non-deterministic scheduler, or (ii) an MDP in case of probabilistic schedulers. These are perfect-information games and hence, can be solved efficiently. In case (i), we use a standard min-mean cycle algorithm (for example, [109]) to find the value of the strategy . In case (ii), we focus on solving Markov chains with limit-average objectives efficiently. Markov chains arise from MDPs due to two reasons: (1) In many cases,

program input can be abstracted away using data abstraction and the problem is reduced to solving a LimAvg Markov Chain. (2) The most efficient algorithm for LimAvg MDPs is the strategy improvement algorithm [85], and each step of the algorithm involves solving a Markov chain (for standard techniques, see [85]).

In practice, a large fraction of concurrent programs are designed to ensure progress condition called *lock-freedom* [98]. Lock-freedom ensures that some thread always makes progress in a finite number of steps. This leads to Markov chains with a directed-acyclic tree like structure with only few cycles introduced to eliminate finite executions as mentioned in Section 6.2[Pg. 93]. We present a *forward propagation* technique to compute stationary probabilities for these Markov chains. Computing the stationary distribution for a Markov chain involves solving a set of linear equalities using Gaussian elimination. For Markov chains that satisfy the special structure, we speed up the process by eliminating variables in the tree by forward propagating, i.e., substituting the root variable. Using this technique, we were able to handle the Markov chains of up to 100,000 states in a few seconds in the experiments.

Quantitative probabilistic abstraction. To improve the performance of the synthesis, we use standard abstraction techniques. However, for the partial program resolution problem we require abstraction that also preserves quantitative objectives such as LimAvg and LimAvg-Safety. We show that an extension of probabilistic bisimilarity [119] with a condition for weight function preserves the quantitative objectives.

Quantitative probabilistic bisimilarity. A binary equivalence relation $\equiv$ on the states of a MDP is a *quantitative probabilistic bisimilarity* relation if (a) $s \equiv s'$ iff s and s' are both safe or both unsafe; (b) $\forall s \equiv s', a \in \Sigma$: $\sum_{t \in C} \Delta(s,a)(t) = \sum_{t \in C} \Delta(s',a)(t)$ where C is an equivalence class of $\equiv$; and (c) $s \equiv s' \wedge t \equiv t' \implies v(s,a,s') = v(t,a,t')$. The states s and s' are *quantitative probabilistic bisimilar* if $s \equiv s'$.

A *quotient* of an MDP $\mathcal{G}$ under quantitative probabilistic bisimilarity relation $\equiv$ is an MDP $({}^{\mathcal{G}}/_{\equiv})$ where the states are the equivalence classes of $\equiv$ and: (i) $v(C,a,C') = v(s,a,s')$ where $s \in C$ and $s' \in C'$, and (ii) $\Delta(C,a)(C') = \sum_{t' \in C'} \Delta'(s,a)(t)$ where $s \in C$.

Theorem 6.8. *Given an MDP $\mathcal{G}$, a quantitative probabilistic bisimilarity relation $\equiv$, and a limit-average safety objective f, the values in $\mathcal{G}$ and ${}^{\mathcal{G}}/_{\equiv}$ coincide.*

Proof. For every Player 2 strategy ϕ_2 in $\mathcal{G}$, we define a Player 2 strategy $({}^{\phi_2}/_{\equiv})$ in ${}^{\mathcal{G}}/_{\equiv}$ (or vice-versa) where: $\phi_2((s_1,a_1)(s_2,a_2)\dots(s_n,a_n)\cdot s_{n+1}) = ({}^{\phi_2}/_{\equiv})((C_1,a_1)(C_2,a_2)\dots(C_n,a_n)\cdot C_{n+1})$ where C_i is the equivalence class containing s_i. By the properties of $\equiv$, it is simple to check that both ϕ_2 and ${}^{\phi_2}/_{\equiv}$ have equal values. □ □

Consider a standard abstraction technique, *data abstraction*, which erases the value of given variables. We show that under certain syntactic restrictions (namely, that the abstracted variables do not appear in any guard statements), the equivalence relation given by the abstraction is a quantitative probabilistic bisimilarity relation and thus is a sound abstraction with respect to any limit-average safety objective. We also consider a less coarse abstraction, *equality and*

LC: CC	Granularity	Performance
1:100	Coarse	1
	Medium	1.15
	Fine	1.19
1:20	Coarse	1
	Medium	1.14
	Fine	1.15
1:10	Coarse	1
	Medium	1.12
	Fine	1.12
1:2	Coarse	1
	Medium	1.03
	Fine	0.92
1:1	Coarse	1
	Medium	0.96
	Fine	0.80

Table 6.1: Performance of shared buffers under various locking strategies: LC and CC are the locking cost and data copying cost

order abstraction, which preserves equality and order relations among given variables. This abstraction defines a quantitative probabilistic bisimilarity relation under the syntactic condition that the guards test only for these relations, and no arithmetic is used on the abstracted variables.

6.5 Experiments

We describe the results obtained by applying our implementation of techniques described above on four examples. In the examples, obtaining a correct program is not difficult and we focus on the synthesis of optimal programs.

The partial programs were manually abstracted (using the data and order abstractions) and translated into PROMELA, the input language of the SPIN model checker [101]. The abstraction step was straightforward and could be automated. The transition graphs were generated using SPIN. Then, our tool constructed the game graph by taking the product with the scheduler and performance model. The resulting game was solved for the LimAvg-Safety objectives using techniques from Section 6.4[Pg. 101]. The examples we considered were small (each thread running a procedure with 15 to 20 lines of code). The synthesis time was under a minute for all but one case (Example 2 with larger values of n), where it was under five minutes. The experiments were run on a dual-core 2.5Ghz machine with 2GB of RAM. For all examples, the tool reports normalized performance metrics where higher values indicate better performance.

Example 1. We consider the producer-consumer example described in Section 6.1[Pg. 90], with two consumer and two producer threads. The partial program models a four slot concurrent buffer which is operated on by producers and consumers. Here, we try to synthesize lock granularity. The synthesis results are presented in Table 6.1[Pg. 104]. The most important parameters in the

WC : LC	LWO	Performance for `n`				
		1	2	3	4	5
20:1	1	1.0	1.049	1.052	1.048	1.043
20:1	2	1.0	0.999	0.990	0.982	0.976
10:1	1	1.0	1.134	1.172	1.187	1.193
10:1	2	1.0	1.046	1.054	1.054	1.052

Table 6.2: Optimistic performance: WC, CC, and LWO are the work cost, lock cost, and the length of the work operation

performance model are the cost of locking/unlocking l and the cost c of copying data from/to shared memory. If c was higher than l (by 100:1), then the fine-grained locking approach is better (by 19 percent), and is the result of synthesis. If the cost l is equal to c, then the coarse-grained locking approach was found to perform better (by 25 percent), and thus the coarse-grained program is the result of the synthesis.

Example 2. We consider the optimistic concurrency example described in detail in Section 6.1[Pg. 90]. In the code (Figure 6.1[Pg. 91]), the number of operations performed optimistically is controlled by the variable `n`. We synthesized the optimal `n` for various performance models and the results are summarized in Table 6.2[Pg. 105]. We were able to find correspondence between our models and the program behavior on a desktop machine: (a) We observed that the graph of performance-vs-`n` has a local maximum when we tested the partial program on the desktop. In our experiments, we were able to find parameters for the performance model which have similar performance-vs-`n` curves. (b) Furthermore, by changing the cost of locking operations on a desktop, by introducing small delays during locks, we were able to observe performance results similar to those produced by other performance model parameters.

Example 3. We synthesize the optimal number of threads for work sharing (pseudocode in Figure 6.4[Pg. 106]). For independent operations, multiple threads utilize multiple processors more efficiently. However, for small number of operations, thread initialization cost will possibly overcome any performance gain.

The experimental results are summarized in Figure 6.5[Pg. 106]. The x- and y- axes measure the initialization cost and performance, respectively. Each plot in the graph is for a different number of threads. The two graphs (a) and (b) are for a different amounts of work to be shared (the length of the array to be operated was varied between 16, and 32). As it can be seen from the figure, for smaller amounts of work, spawning fewer threads is usually better. However, for larger amounts of work, greater number of threads outperforms smaller number of threads, even in the presence of higher initialization costs. The code was run on a desktop (with scaled parameters) and similar results were observed.

```
main:
   n = choice(1..10);
   i = 0;
   array[0..N];
   while (i < n) {
       spawn(worker, i * (N/n), (N/n));
       i++;
  }

worker(start, length):
   i = start;
   while(i < start + length) {
       work(array[i]);
  }
```

Figure 6.4: Pseudo-code for Example 3

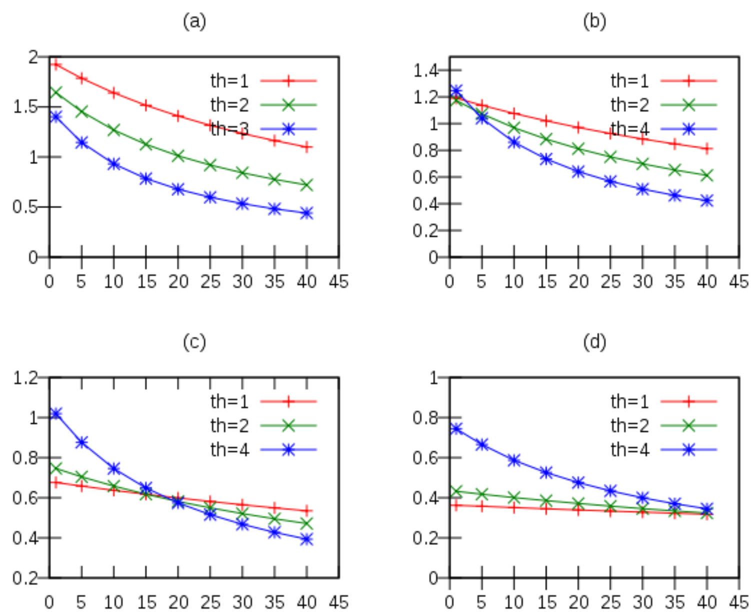


Figure 6.5: Work sharing for initialization costs and thread counts: More work is shared in case (b) than case (a)

```
1: while(true) {
2:     n  = choice(1..10);
3:     lock();
4:     while (i < n) {
5:         data = write(work(read(data)));
6:     }
7:     unlock(lock);
8:}
```

Figure 6.7: Pseudo-code for Example 4

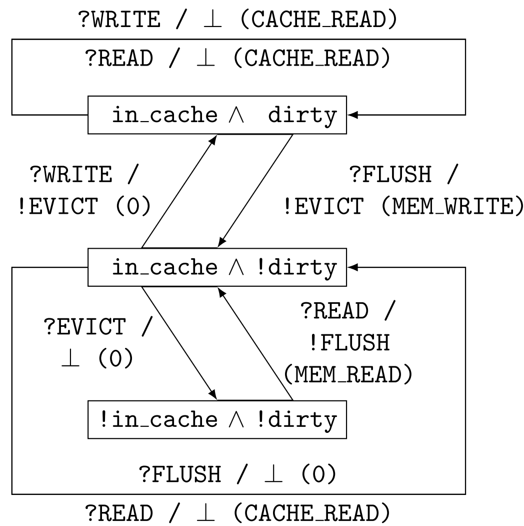


Figure 6.6: Performance automaton for Example 4

Example 4. We study the effects of processor caches on performance using a simple performance model for caches. A cache line is modeled as in Figure 6.6[Pg. 107]. It assigns differing costs to read and write actions if the line is cached or not. The performance model is the synchronous product of one such automata per memory line. The only actions in the performance model after the synchronous product (caches synchronize on `evict` and `flush`) are `READ` and `WRITE` actions. These actions are matched with the transitions of the partial program.

The partial program is a pessimistic variant of Figure 6.1[Pg. 91] (pseudocode shown in Figure 6.7[Pg. 107]). Increasing `n`, i.e., the number of operations performed under locks, increases the temporal locality of memory accesses and hence, increase in performance is expected. We observed the expected results in our experiments. For instance, increasing `n` from 1 to 5 increases the performance by a factor of 2.32 and increasing `n` from to 10 gives an additional boost of about 20%. The result of the synthesis is the program with $n = 10$.

6.6 Summary

Our main contributions are: (1) we developed a technique for synthesizing concurrent programs that are both correct and *optimal*; (2) we introduced a parametric performance model providing a flexible framework for specifying performance characteristics of architectures; (3) we showed how to apply imperfect-information games to the synthesis of concurrent programs and established the complexity for the game problems that arise in this context (4) we developed and implemented practical techniques to efficiently solve partial-program synthesis, and we applied the resulting prototype tool to several examples that illustrate common patterns in concurrent programming.

Chapter 7

Quantitative Abstraction Refinement

We propose a general framework for abstraction with respect to quantitative properties, such as worst-case execution time, or power consumption. Our framework provides a systematic way for counter-example guided abstraction refinement for quantitative properties. The salient aspect of the framework is that it allows anytime verification, that is, verification algorithms that can be stopped at any time (for example, due to exhaustion of memory), and report approximations that improve monotonically when the algorithms are given more time.

We instantiate the framework with a number of quantitative abstractions and refinement schemes, which differ in terms of how much quantitative information they keep from the original system. We introduce both state-based and trace-based quantitative abstractions, and we describe conditions that define classes of quantitative properties for which the abstractions provide over-approximations. We give algorithms for evaluating the quantitative properties on the abstract systems. We present algorithms for counter-example based refinements for quantitative properties for both state-based and segment-based abstractions. We perform a case study on worst-case execution time of executables to evaluate the anytime verification aspect and the quantitative abstractions we proposed.

7.1 Motivation

The quantitative analysis of systems is gaining importance due to the spread of embedded systems with requirements on resource consumption and timeliness of response. Quantitative analyses have been proposed for properties such as worst-case execution time (see [171] for a survey), power consumption (pioneered in [158]), and prediction of cache behavior for timing analysis (see, for example, [84]).

Anytime algorithms (see [25]) are algorithms that generate imprecise answers quickly and proceed to construct progressively better approximate solutions over time, eventually finding the correct solution. Anytime algorithms are useful in verification, as they offer a way to deal with the state-space explosion problem

```
int a, b, c, i;
volatile int v; // input
if (v == 1)
    for (i=0;i<16;i++)
        read(a);
else if(v == 2)
    for (i=0;i<16;i++)
        if (i mod 2 = 0)
            read(b);
else
    for (i=0;i<16;i++)
        if (i mod 4 = 0)
            read(c);
```

Figure 7.1: Example 1

— if the algorithm is terminated early, for example due to memory exhaustion, it is still able to report an approximation of the desired result. The term *anytime verification* has been proposed [151] recently in the context of verifying boolean properties, as a way to get (non-quantifiably) better estimates on whether a property holds for a system. The anytime concept, however, is particularly well-suited in the context of *quantitative* verification. In this context, abstraction gives a quantitative over-approximation of the quantitative answer to a verification question, and it is natural to require the anytime property: the more time the verification run is given, the (quantifiably) better the over-approximation of the correct answer should be. We implement this anytime property for quantitative verification through an abstraction refinement scheme that monotonically improves the answer. For instance, abstraction refinement may compute increasingly better approximations of power consumption of a system.

We propose a framework for abstraction and abstraction refinement for quantitative properties that is suitable for anytime verification. We explain the motivation and intuition behind the framework using the following example.

Motivating example. Consider the problem of estimating the worst-case execution time (WCET) of the program in Figure 7.1[Pg. 111]. We assume an idealized situation where the performance is affected mostly by the cache behavior. Let each program statement have a cost depending on whether it accesses only the cache (or no memory at all), for a cost of 1, or main memory for a cost of 25. we assume that the program variables `i,a,v` are mapped to different cache entries, while `b` and `c` are mapped to the same entry (different from the entries for the other variables). We consider abstractions that abstract only the cache (not the program). The cache is abstracted by an *abstract cache* with a smaller number of entries (accesses to the entries not tracked in the abstract cache are always considered to be a cache miss). Let us start the analysis with an abstract cache of size 2 which caches variables `i` and `v`. In the abstract system (i.e., the original program composed with the abstract cache), the worst-case trace has `v` equal to 1, and the program accesses the (uncached)

variable `a` 16 times. The analysis then uses this trace to refine the abstraction. The refinement extends the cache to include the cache entry for `a`. The worst-case execution then has v equal to 2, and it has 8 accesses to `b`. The analysis now refines the abstraction by extending the cache with an entry for `b`. The WCET estimate is thus tightened, until either the highest-cost trace corresponds to a real execution (and thus the WCET estimate is precise), or the analysis runs out of resources and reports the computed over-approximation.

Abstraction for quantitative properties. Our model of systems is weighted transition systems. We provide a way of formalizing quantitative properties of systems which capture important properties studied in literature, including *limit-average*, *discounted-sum*, and boolean properties such as safety and liveness. The framework makes it possible to investigate quantitative versions of the boolean properties: for instance, for safety one could ask not only if an error state is reached, but also how often it is reached. We focus on properties that admit a linear trace that maximizes (or minimizes) the value of the quantitative property. Such a trace is called the *extremal trace* (*ext*-trace for short).

We present two types of quantitative abstraction schemes. The first is *state-based*, that is, the elements of the abstract domain correspond to sets of states. The second is *segment-based*, that is, the elements of the abstract domain correspond to sets of trace segments.

State-based quantitative abstractions. The abstraction scheme *ExistMax* is state-based. It is a direct extension of predicate abstraction, where each abstract state corresponds to an equivalence class of concrete states. In addition, with each abstract state, *ExistMax* stores the maximum weight of the corresponding concrete states. We give conditions for the class of quantitative properties for which *ExistMax* is a *monotonic over-approximation* (that is, it provides better estimates as the underlying equivalence relation on states is refined). This class includes all the quantitative properties mentioned above. However, we show that there are naturally defined properties for which *ExistMax* is not a monotonic over-approximation. This is in contrast to the abstraction refinement of boolean properties (in the boolean case, we do not get less precise invariants if we add more predicates).

Segment-based quantitative abstractions. We introduce a number of segment-based abstraction schemes. A *segment* is a (finite or infinite) sequence of states that is a consecutive subsequence of an execution trace. As the quantitative properties we consider accumulate quantities along (infinite) traces, it is natural, and advantageous, to consider abstract domains whose elements correspond to sets of segments, not sets of states. This is similar to termination and liveness analysis using transition predicates [136], and their generalizations to segment covers [67]. We build upon these approaches to develop our quantitative abstractions. The *PathBound* abstraction scheme stores with each abstract state t (representing a set of segments) (i) $minp(t)$, the length of a shortest finite segment in t, (ii) $maxp(t)$, the length of the longest finite segment in t, (iii) $hasInfPath(t)$ a bit that is true if t contains a segment of infinite length, and (iv) $Val(t)$, a summary value of the weights of the states. Defining $Val(t)$ as the max-

imal weight of a state occurring in one of the segments in t makes *PathBound* a sound over-approximation for a general class of quantitative properties. To get better approximations for particular quantitative properties $Val(t)$ can be a different summary of the weights. For instance, we specialized *PathBound* to limit average by storing not the maximal occurring value, but the maximal average of values along a segment.

In order to compare state-based and segment-based abstractions, let us consider the limit-average property applied to a program with a simple `for` loop for which the loop bound is statically known to be 10. Let us assume that the cost of the operations inside the loop is much greater than the cost of the operations outside the loop. Now consider the state-based abstraction *ExistMax* with an abstract state t that groups together all states whose control location is in the `for` loop. In the *ExistMax* abstraction, this abstract state has a self-loop. Analyzing the abstract system would conclude that the highest-cost trace is the one which loops forever in t. This would be a very imprecise result, as the concrete traces all leave the loop after 10 iterations. To correct for this, the loop would have to be unrolled 10 times, using 10 counterexample-guided refinement steps. On the other hand, consider the *PathBound* abstraction, with an abstract state t representing all the segments in the loop. For t, $hasInfPath(t)$ is false and $maxp(t) = 10$, allowing immediately for more precise estimates.

The *PathBound* abstraction scheme is a sound abstraction in the sense that the quantitative value we obtain by analyzing the abstract system over-approximates the value for the concrete system. As in *ExistMax*, this holds for a large class of quantitative properties. However, we show that *PathBound* is not a monotonic over-approximation even for standard quantitative properties such as the limit-average property in the sense that after refinement of the abstract states, we may get worse estimates. We therefore present a hierarchical generalization of *PathBound* called *HPathBound*. In *PathBound*, each abstract state represents a set of segments, and stores some quantitative characteristics (such as *minp*, *maxp*) of that set. In order to compute better estimates of these quantitative characteristics, one can perform another level of refinement, within abstract states. This leads to the idea of a hierarchical abstraction. It is particularly useful for software, which already has a hierarchical structure in many cases (e.g., nested loops, function calls). This approach corresponds to the (multi-level) abstract inductive segment cover of introduced in [67, Section 16.1].

Refinement of state-based abstractions for quantitative properties. For the state-based abstraction scheme *ExistMax*, we give an algorithm for counterexample-guided abstraction refinement (CEGAR) for quantitative properties. The algorithm is based on the classical CEGAR algorithm [60], which we extend to the quantitative case. In the classical CEGAR loop, the counterexample to be examined is chosen using heuristics. For quantitative properties, it is clear that an extremal counterexample trace (the *ext*-trace) should be chosen for refinement. The reason is that if the *ext*-trace does not correspond to a real trace, then a refinement which does not eliminate this trace would have the same value as the previous abstract system.

Refinement of segment-based abstractions for quantitative properties. We propose a refinement algorithm for the segment-based abstraction *HPathBound*. We chose *HPathBound* because, as discussed above, it is particularly suitable for software, and it is a monotonic over-approximation for a large set of quantitative properties. An abstract counterexample for *HPathBound* is a hierarchical trace. Given an extremal abstract counterexample which does not correspond to a concrete counterexample, the abstract counterexample is traversed, similarly as in the classical CEGAR algorithm, until an abstract segment that provides values that are too "pessimistic" (e.g., *maxp* and *Val* values which cannot be achieved in the concrete system by a concretization of the abstract counterexample) is found. This abstract segment is then refined. Note that the fact that the counterexample is hierarchic gives freedom to the traversal algorithm — at each step, it can decide whether or not to descend one level lower and to find a mismatch between a concrete and abstract execution there.

Experimental results In order to evaluate the proposed abstraction schemes, both state-based and segment-based, we performed a case study on worst-case execution time analysis of x86 executables. We focused on one aspect, the cache behavior analysis, and in particular, on estimating the rate of cache misses over the course of the worst-case execution. In order to abstract the cache, we used the abstractions introduced in [84]. To the best of our knowledge, this is the first work on automated refinement for these abstractions.

We implemented two abstraction schemes: the state-based *ExistMax* and the segment-based *HPathBound*. We performed the case study on our own (small) examples, and on some of the benchmarks collected in [91]. The experiments show that we obtain more precise quantitative results as the abstraction is refined, for example, by having a larger abstract cache. Furthermore, we show that using the segment-based abstraction *HPathBound* enables scaling up. This is due to the fact that in the presence of loops, the *HPathBound* abstraction can quickly obtain good over-approximations if it can statically over-approximate loop bounds, whereas the *ExistMax* abstraction would have to unroll the loop many times to get comparable results. Similarly to the *ExistMax* case, the experiments show that we obtain more precise quantitative results as the *HPathBound* abstraction is (hierarchically) refined by computing better estimates for loop bounds. The running time of the analysis was under 35 seconds in all cases.

7.2 Quantitative properties

In this chapter, we use weighted transition systems where the weights are assigned to states rather than transitions. An *unlabelled weighted transition system* (UWTS) is given by $W = \langle S, \Delta, v \rangle$ where S are a finite set of states, $\Delta \subseteq S \times S$ are a set of unlabelled transitions, and $v : S \to \mathbb{Q}$ is a weight function mapping states to rational weights. We use $\mathcal{W}$ to denote the set of all UWTSs and $\mathcal{W}(\mathcal{S}) \subseteq \mathcal{W}$ to denote the set of all UWTSs with a set of states S such that $S \subseteq \mathcal{S}$.

A trace of a UWTS is given by a sequence $\pi = s_0 s_1 \ldots$ such that $(s_i, s_{i+1}) \in \Delta$ for each $i \geq 0$. A trace $\pi = s_0 s_1 \ldots$ is *memoryless* if for all $i, j \geq 0$, we have $s_i = s_j \to (s_{i+1} = s_{j+1})$. We extend the weight function v to traces by defining

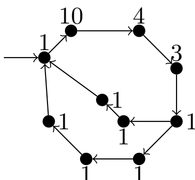


Figure 7.2: System W_1

$v(s_0 s_1 \ldots) = v(s_0)v(s_1)\ldots$. For instance, system W_1 in Figure 7.2[Pg. 115] has two memoryless traces, a trace π_1 with $v(\pi_1) = (1\ 10\ 4\ 3\ 1\ 1\ 1)^\omega$, and a trace π_2 with $v(\pi_2) = (1\ 10\ 4\ 3\ 1\ 1\ 1\ 1)^\omega$.

We define quantitative properties by using a *trace value function* and a *system value function*. A trace value function $f_t : \mathbb{R}^\omega \to \mathbb{R}$ maps a sequence of weights to a real number and a system value function $f_s : 2^{\mathbb{R}} \to \mathbb{R}$. For trace value functions, we use standard functions from Chapter 2[Pg. 16] such as limit-average, infimum, supremum, and discounted-sum. We also give some examples of standard system value functions.

- *Supremum and Infimum.* Intuitively, the supremum and infimum functions measure the worst-case and best-case traces in the system.
- *Threshold.* The threshold function checks if any of the values in the given set are above or below given thresholds. Formally, $threshold_u(R) = 1$ if $\exists r \in R : r \geq u$, and $threshold_u(R) = 0$ otherwise.

Any combination of a trace value function f_t and a system value function f_t defines a quantitative property of the system W as $f_s(\{f_t(\pi) \mid \pi \in W\})$.

In this chapter, we implicitly assume that the system value function for any quantitative property is sup unless otherwise mentioned. A trace π is an *extremal counterexample trace* (or *ext*-trace for short) if $f(W) = f_s(\{f_t(v(\pi))\})$. We restrict ourselves to a class of quantitative properties that admit *memoryless extremal counterexample traces*, i.e., every system W has a extremal counterexample trace that is memoryless. Formally, f is *memoryless* if and only if $\forall W \in \mathcal{W} : \exists \pi \in \Pi(W) : f(W) = f_s(\{f_t(v(\pi))\})$. Note that all properties mentioned above are memoryless.

7.3 State-based quantitative abstractions

A *quantitative abstraction* $C = (\mathcal{W}^C, f^C, \alpha^C)$ is triple consisting of a set of abstract systems $\mathcal{W}^C$, an abstract quantitative property $f^C : \mathcal{W}^C \to \mathbb{R}$ and an abstraction function $\alpha^C : \mathcal{W} \to \mathcal{W}^C$. A quantitative abstraction C is an *over-approximation* of f, if for all $W \in \mathcal{W}$, $f^C(\alpha^C(W)) \geq f(W)$.

7.3.1 *ExistMax* abstraction.

In this section, we present a quantitative abstraction technique based on state abstractions. In this case, the abstract system is a UWTS whose states are sets of states of the concrete systems. In particular, our abstraction scheme *ExistMax* is a direct extension of the classical predicate abstraction. Compared to predicate abstraction, *ExistMax* additionally stores, with each abstract state, the maximum weight occurring in the set of corresponding concrete states.

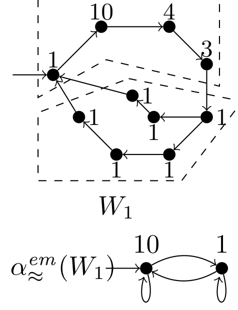


Fig. 7.3: *ExistMax* abstraction

The name *ExistMax* refers to transitions abstracted existentially, and that the weights abstracted by maxima.

ExistMax is a state based abstraction scheme: a family of quantitative abstractions parameterized by equivalence relations on $\mathcal{S}$. Given an equivalence relation $\approx \subseteq \mathcal{S} \times \mathcal{S}$, the quantitative abstraction $\mathit{ExistMax}_{\approx} = (\mathcal{W}^{em}, f^{em}, \alpha^{em}_{\approx})$ has (a) $\mathcal{W}^{em} = \mathcal{W}(2^{\mathcal{S}})$, (i.e., abstract states are set of concrete states), (b) $f^{em} = f$ as the abstract quantitative property, and (c) for a UWTS $W = (S, \Delta, v, s_\iota)$ we have that the abstract system $\alpha^{em}_{\approx}(W)$ is a UWTS $W^{em} = (S^{em}, \Delta^{em}, v^{em}, t^{em}_\iota)$ where:

- S^{em} are equivalence classes of $\approx$ that contain states from S;
- $(t_1, t_2) \in \Delta^{em} \Leftrightarrow \exists s_1 \in t_1, s_2 \in t_2 : (s_1, s_2) \in \Delta$;
- $v(t) = \sup\{d \mid \exists s \in t : v(s) = d\}$; and
- t^{em}_ι is the equivalence class in S^{em} that contains s_ι.

Intuitively, $\alpha^{em}(W)$ is an existential abstraction and v^{em} maps an abstract state to the maximum weight of the corresponding concrete state.

Example 7.1. *Consider again the system W_1 from Figure 7.2[Pg. 115], and the equivalence relation $\approx$, whose two equivalence classes (indicated by the dashed shapes) are shown in the upper part of Figure 7.3[Pg. 116]. The abstract system $W^{em} = \alpha^{em}_{\approx}(W_1)$ is in the lower part of Figure 7.3[Pg. 116]. We have* $\mathrm{LimAvg}^{em}(\alpha^{em}_{\approx}(W^{em})) = 10$*, due to a self-loop on the abstract node with weight* 10.

Over-approximation and monotonicity.

We characterize the quantitative properties for which *ExistMax* is an over-approximation, and for which monotonicity of refinements holds, i.e., where refinement of the abstraction leads to tighter approximations of the system value.

We borrow the classical notion of refinement for abstractions. An equivalence relation $\equiv$ is a *refinement* of an equivalence relation $\approx$ if and only if every equivalence class of $\equiv$ is a subset of an equivalence class of $\approx$.

We define the following quasi-orders:

- let $\leq_p \subseteq \mathbb{R}^\omega \times \mathbb{R}^\omega$ be defined by: $r^1_0 r^1_1 \ldots \leq_p r^2_0 r^2_1 \ldots$ if and only if $\forall i$: $r^1_i \leq r^2_i$, and
- let $\sqsubseteq \subseteq 2^{\mathbb{R}} \times 2^{\mathbb{R}}$ be defined by: for $U, U' \subseteq \mathbb{R}$, we have $U \sqsubseteq U'$ iff $\sup U \leq \sup U'$.

A quantitative property f is $(\leq_p, \sqsubseteq)$-*monotonic* if $\forall r, r' \in \mathbb{R}^\omega : r \leq_p r' \implies f_t(r) \leq f_t(r')$ and $\forall U, U' \in 2^{\mathbb{R}} : U \sqsubseteq U' \implies f_s(U) \leq f_s(U')$.

ExistMax is a *monotonic over-approximation* for a quantitative property f, if (a) *ExistMax* is an over-approximation for f, and (b) if for all $W \in \mathcal{W}(\mathcal{S})$, and for all equivalence relations $\equiv$ and $\approx$ on $\mathcal{S}$, such that $\equiv$ is a refinement of $\approx$, we have that $f^{em}(\alpha^{em}_{\equiv}(W)) \leq f^{em}(\alpha^{em}_{\approx}(W))$.

Theorem 7.2. *If f is $(\leq_p, \sqsubseteq)$-monotonic quantitative property, then ExistMax is a monotonic over-approximation of f.*

Proof. We first prove the monotonicity property of *ExistMax*. The over-approximation property follows naturally as follows. For any system W, we have $W = \alpha^{em}_{id}(W)$ where id is the identity relation. As id is a refinement of any equivalence relation $\approx$, by monotonicity it follows that $f(W) = f^{em}(\alpha^{em}_{id}(W)) \leq f^{em}(\alpha^{em}_{\approx}(W))$.

Let W be a system in $\mathcal{W}(\mathcal{S})$, and let $\equiv$ and $\approx$ be equivalence relations on $\mathcal{S}$. Let $\alpha_{\approx}(W) = (S^1, \Delta^1, v^1, s^1_\iota)$ and $\alpha_{\equiv}(W) = (S^2, \Delta^2, v^2, s^2_\iota)$ be *ExistMax* abstractions of W, where $\equiv$ is a refinement of $\approx$. Furthermore, let f defined by f_t and f_s be $(\leq_p, \sqsubseteq)$-monotonic. For each equivalence class t of $\equiv$, let $t^\sharp$ be the unique equivalence class of $\approx$ for which $t \subseteq t^\sharp$. The class $t^\sharp$ is guaranteed to exist as $\equiv$ is a refinement of $\approx$. Furthermore, by the definition of *ExistMax*, we have that: (a) $v^1(t) \leq v^2(t^\sharp)$, and (b) $(t_1, t_2) \in \Delta^1 \implies (t^\sharp_1, t^\sharp_2) \in \Delta^2$. Therefore, for any trace $\pi = t_0 t_1 \ldots$ of $\alpha_{\equiv}(W)$, there exists a trace $\pi^\sharp = t^\sharp_0 t^\sharp_1 \ldots$ of $\alpha_{\approx}(W)$ such that: $v^1(t_0)v^1(t_1)\ldots \leq_p v^2(t^\sharp_0)v^2(t^\sharp_1)\ldots$. By $\leq_p$-monotonicity of f_t, we get $f_t(v(\pi)) \leq f_t(v(\pi^\sharp))$. Hence, for each $w \in f_t(v(\Pi(\alpha_{\equiv}(W))))$, there exists $w^\sharp \in f_t(v(\Pi(\alpha_{\approx}(W))))$ with $w \leq w^\sharp$. This, in turn, gives us $f_t(v(\Pi(\alpha_{\equiv}(W)))) \sqsubseteq f_t(v(\Pi(\alpha_{\equiv}(W))))$. Hence, by $\sqsubseteq$-monotonicity of f_s, we get $f_s(f_t(v(\Pi(\alpha_{\equiv}(W))))) \leq f_s(f_t(v(\Pi(\alpha_{\approx}(W)))))$, or equivalently, $f(\alpha_{\equiv}(W)) \leq f(\alpha_{\approx}(W))$. This proves the required theorem. as $f = f^{em}$. ☐

It is easy to show that limit average, and discounted sum are $(\leq_p, \sqsubseteq)$-monotonic. The following proposition is a direct consequence.

Proposition 7.3. *ExistMax is a monotonic over-approximation for the limit-average, and discounted-sum.*

Example 7.4. *We describe a property for which ExistMax is not a monotonic over-approximation. Let f be defined by $f_t(r) = \sup_{i,j \geq 0}(r_i - r_j)$ and $f_s(U) = \sup U$. The property f can be used to measure the variance in resource usage (where the usage in each step is given by the weight) during the execution of a program. Consider the system in Figure 7.4[Pg. 118] and the ExistMax abstraction with abstract states given by the rectangles (the nodes outside the dotted boxes are each in a separate singleton equivalence classes) . Property f has value 2 on the abstract system due to the trace $A \to B \to C$ having maximal and minimal weights as 5 and 3 (under ExistMax abstract state A, B, and C have weights 5, 3, and 3 respectively). Refining the abstraction by completely splitting state B increases f to 4. Refining further by splitting both states A and C decreases f to 3 which is the true value of the concrete system. The sequence of refinements show that for property f, the ExistMax abstraction is neither an over-approximation (as the first abstract system has value 2 which is less than 3, the value of the concrete system), nor monotonic (as the sequence of values 2, 4 and 3 obtained through subsequent refinements first increase and then decrease).*

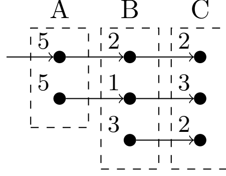


Fig. 7.4: Non-monotonic refinements

Evaluating quantitative properties on *ExistMax* abstractions.

Recall $f^{em} = f$ for *ExistMax* abstractions. To evaluate f^{em} (and obtain an *ext*-trace for refinement), any algorithm for finding *ext*-traces for the quantitative property f suffices. Standard algorithms exist when f is one of safety, liveness, discounted-sum, and limit-average properties. For limit-average, we use the classical Howard's policy iteration.

7.4 Segment-based quantitative abstractions

In this section, we present quantitative abstractions where elements of the abstract domain correspond to sets of trace segments.

Segments. A *segment* is a finite or infinite sequence of states in $\mathcal{S}$. Let $\mathcal{S}^*$ be the set of all finite segments, $\mathcal{S}^\infty$ the set of all infinite segments, and let $\mathcal{S}^{*\infty} = \mathcal{S}^* \cup \mathcal{S}^\infty$ be the set of all finite and infinite segments. Given a segment σ, let $\|\sigma\|$ denote the length of the segment (the range of $|.|$ is thus $\mathbb{N} \cup \{\infty\}$). Given two segments σ_1 and σ_2 in $\mathcal{S}^{*\infty}$ we write $\sigma_1\sigma_2$ for their concatenation, with $\sigma_1\sigma_2 = \sigma_1$, if σ_1 is in $\mathcal{S}^\infty$. Also, we the notation $last(\sigma_1)$ and $first(\sigma_2)$ to represent the last state of a finite segment σ_1 and the first state of a segment σ_2.

We dub a nonempty set of segments a *SegmentSet*. We define the following operations and relations on SegmentSets and sets of SegmentSets.

- For SegmentSets T_1 and T_2, we have $T_1 \Subset T_2$ if $T_1 \subseteq \{w \mid \exists x \in \mathcal{S}^*, \exists y \in \mathcal{S}^{*\infty} : xwy \in T\}$, that is, all segments from T_1 occur as sub-segments of segments in T_2.
- For a set of SegmentSets $\mathcal{T}$, we define $\biguplus \mathcal{T}$ to be set of segments which can be obtained by concatenation of segments contained in SegmentSets in $\mathcal{T}$. Formally, $\biguplus \mathcal{T} = \{\sigma_0\sigma_1 \ldots \sigma_n \mid \exists T_0, T_1 \ldots T_n : (\forall i : 0 \leq i \leq n \implies \sigma_i \in T_i)\} \cup \{\sigma_0\sigma_1 \ldots \mid \exists T_0, T_1 \ldots : (\forall i : 0 \leq i \to \sigma_i \in T_i)\}$.
- A set of SegmentSets $\mathcal{T}$ *covers* a SegmentSet T if and only if
  - for all $T_i \in \mathcal{T}$, we have $T_i \Subset T$, and
  - $T \subseteq \biguplus \mathcal{T}$.

Note that for a UWTS W, the set of all its traces $\Pi(W)$ is a SegmentSet. We call $\mathcal{T}$ a *segment cover* of a system W if and only if $\mathcal{T}$ covers $\Pi(W)$. For example, the two SegmentSets T_1 and T_2 in Figure 7.5[Pg. 119] form a segment cover of the system in Figure 7.2[Pg. 115]. It is easy to see that all traces of W are covered by segments in T_1 and T_2. Our notion of segment cover corresponds to the inductive trace segment cover from [67] with height 1. The notion of the segment cover plays the same role in segment-based abstractions as the equivalence relation on states plays in state-based abstractions.

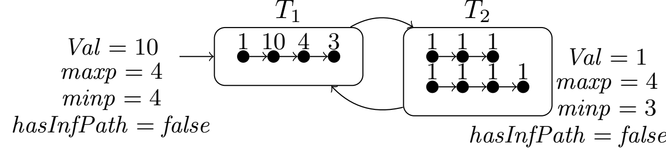


Figure 7.5: *PathBound* abstraction of W_1

7.4.1 *PathBound* abstraction.

PathBound is a segment-based abstraction scheme: a family of quantitative abstractions parameterized by sets of SegmentSets on $\mathcal{S}$. Given a set of SegmentSets $\mathcal{T}$, the quantitative abstraction $PathBound_{\mathcal{T}}$ is defined by $(\mathcal{W}^{pb}_{\mathcal{T}}, f^{pb}, \alpha^{pb}_{\mathcal{T}})$. We now define each element of the triple.

Abstract systems $\mathcal{W}^{pb}_{\mathcal{T}}$. An abstract system in $\mathcal{W}^{pb}_{\mathcal{T}}$ is a tuple $(R, \Delta_R, Val, minp, maxp, hasInfPath, R_0)$, where R is $\mathcal{T}$, Δ_R is a transition relation, and R_0 is the set of initial states. The type of Val, $minp$ and $maxp$ functions is $R \to \mathbb{R}$ and the type of $hasInfPath$ is $R \to \{true, false\}$. Their intuitive meaning is given below. The systems in $\mathcal{W}^{pb}_{\mathcal{T}}$ are called *pb-systems*.

Abstraction $\alpha^{pb}_{\mathcal{T}}$. The partial abstraction function $\alpha^{pb}_{\mathcal{T}}$ is defined as follows. For a UWTS W, if $\mathcal{T}$ is not a segment cover of $\Pi(W)$, then the value $\alpha^{pb}_{\mathcal{T}}(W)$ is undefined. Otherwise, given $W = (S, \Delta, v, s_\iota)$, let $\alpha^{pb}_{\mathcal{T}}(W) = W^{pb} \in \mathcal{W}^{pb}$ where $W^{pb} = (S^{pb}, \Delta^{pb}, Val, minp, maxp, hasInfPath, S^{pb}_0)$ and

- S^{pb} is $\mathcal{T}$
- $(T_1, T_2) \in \Delta^{pb}$ if $\exists \sigma_1 \in T_1, \sigma_2 \in T_2$, such that σ_1 is a finite segment in $\mathcal{S}^*$, $(last(\sigma_1), first(\sigma_2)) \in \Delta$.
- $Val(T) = \max\{v(s) \mid \exists \sigma \in T \text{ and } s \text{ occurs in } \sigma\}$, i.e., $Val(T)$ is the maximal weight of a state occurring in one of the segments in T
- $minp(T) = \min\{\|\sigma\| \mid \sigma \in T \text{ is a finite segment}\}$ if T contains a finite segment, and is ∞ otherwise.
- $maxp(T) = \max\{\|\sigma\| \mid \sigma \in T \text{ is a finite segment}\}$, if T contains a finite segment, and is ∞ otherwise.
- $hasInfPath(T) = true$ iff $T \cap \mathcal{S}^\infty \neq \emptyset$, i.e., $hasInfPath(T)$ is true if and only if T contains an infinite segment,
- S^{pb}_0 contains a set T in $\mathcal{T}$ iff T contains a segment whose first state is s_ι. As $\mathcal{T}$ is a segment cover of W, we have that S^{pb}_0 is non-empty.

A *pb-trace* ρ of a *pb*-system W^α is either (a) a finite sequence $T_0 T_1 \ldots T_n$ such that $T_0 \in S^{pb}_0$, $hasInfPath(T_n)$, and $\forall i : 0 \leq i < n : (T_i, T_{i+1}) \in \Delta^{pb}$, or (b) an infinite sequence $T_0 T_1 \ldots$ with $T_0 \in S^{pb}_0$, and $\forall i \geq 0 : (T_i, T_{i+1}) \in \Delta^{pb}$. The set of all *pb*-traces of W^α is denoted by $\Pi^{pb}(W^\alpha)$.

Example 7.5. *Recall the system W_1 from Figure 7.2[Pg. 115]. Consider a segment cover $\mathcal{T} = \{T_1, T_2\}$ of $\Pi(W)$ depicted in Figure 7.5[Pg. 119]. T_1 and T_2 can now act as abstract states, with the values Val,minp,maxp, and hasInfPath given in Figure 7.5[Pg. 119].*

Abstract quantitative property f^{pb}. In order to define the abstract quantitative property f^{pb}, we will need the following notions.

Let us fix a system $W = (S, \Delta, v, s_\iota)$. Let us also fix a set $\mathcal{T}$ of SegmentSets, such that $\mathcal{T}$ is a segment cover for W. Let $\alpha^{pb}_{\mathcal{T}}(W) = W^\alpha =$

$(S^{pb}, \Delta^{pb}, \mathit{Val}, \mathit{minp}, \mathit{maxp}, \mathit{hasInfPath}, S_0^{pb})$ be a *PathBound* abstraction of W for $\mathcal{T}$.

We now define a function B that for a given pb-trace ρ returns a set of possible sequences of weights that correspond to ρ. The function $B : \Pi(W^\alpha) \to 2^{\mathbb{R}^\omega}$ is defined as follows. The set $B(\rho)$ contains a sequence:

- $w_0^{n_0} w_1^{n_1} \dots w_n^{\infty}$ in $\mathbb{R}^\omega$ iff ρ is a finite pb-trace $T_0 T_1 T_2 \dots T_n$, such that (a) $\forall i$ such that $0 \leq i \leq n$, we have $w_i = \mathit{Val}(T_i)$, and (b) $\forall i$ such that $0 \leq i < n$, we have $\mathit{minp}(T_i) \leq n_i \leq \mathit{maxp}(T_i)$ and $0 < n_i \neq \infty$.
- $w_0^{n_0} w_1^{n_1} \dots$ in $\mathbb{R}^\omega$ iff ρ is an infinite pb-trace $T_0 T_1 \in \Pi^{pb}(W^\alpha)$ such that (a) $\forall i \geq 0 : w_i = \mathit{Val}(T_i)$, and (b) $\forall i \geq 0 : \mathit{minp}(T_i) \leq n_i \leq \mathit{maxp}(T_i) \wedge 0 < n_i \neq \infty$.

Let f be a quantitative property defined by a trace value function f_t and a system value function f_s. We are now able to define the abstract quantitative property f^{pb} by $f^{pb}(W^\alpha) = f_s(f_t((\bigcup_{\rho \in \Pi^{pb}} B(\rho))))$.

Example 7.6. *Recall again the system W_1 from Figure 7.2[Pg. 115] and the abstract cover described in Example 7.5[Pg. 119]. Consider the abstraction $\alpha_{\mathcal{T}}^{pb}(W_1)$ (the abstract system is depicted in Figure 7.5[Pg. 119]). There is only one pb-trace ρ of the abstract system, and we have $\rho = (T_1 T_2)^\omega$. Let us assume that the quantitative property we are interested in is the limit average quantitative property. We get that $B(\rho) = \{(10\ 10\ 10\ 10\ 1\ 1\ 1\ 1)^\omega, (10\ 10\ 10\ 10\ 1\ 1\ 1)^\omega\}$. We therefore obtain $f^{pb}(\alpha_{\mathcal{T}}^{pb}(W_1)) = (10 \cdot 4 + 1 \cdot 3)/(4+3) = \frac{43}{7}$, as the maximum value is achieved if the execution stays at the more costly abstract state T_1 as much as possible ($\mathit{maxp}(T_1)$ times), and at the less costly abstract state T_2 as little as possible ($\mathit{minp}(T_2)$ times).*

Over-approximation and monotonicity

The following theorem states that the abstraction scheme *PathBound* is an over-approximation for a large class of quantitative properties.

Theorem 7.7. *PathBound abstraction scheme is an over-approximation for a quantitative property f if f is $(\leq_p, \sqsubseteq)$-monotonic.*

Proof. Let W be a UWTS and let $\mathcal{T}$ be a trace cover of W. Furthermore, let $W^{pb} = \alpha_{\mathcal{T}}^{pb}(W) = (S^{pb}, \Delta^{pb}, \mathit{Val}, \mathit{minp}, \mathit{maxp}, \mathit{hasInfPath}, S_0^{pb})$. Let π be an extremal trace of W.

As $\mathcal{T}$ is a trace cover of W, we have either:

- *Infinite case.* There exist segments σ_0, σ_1, ... and SegmentSets T_0, T_1, ... with $\pi = \sigma_0 \sigma_1 \dots$ and $\forall i > 0 : |\sigma_i| < \infty \wedge \sigma_i \in T_i$.
- *Finite case.* There exist segments σ_0, σ_1, ..., σ_n and SegmentSets T_0, T_1, ..., T_n with $\pi = \sigma_0 \sigma_1 \dots \sigma_n$ with $\forall 0 \leq i \leq n : \sigma_i \in T_i \wedge (i \neq n) \implies |\sigma_i| < \infty$.

It is easy to show that $\pi^{pb} = T_0 T_1 \dots$ (resp. $\pi^{pb} = T_0 T_1 \dots T_n$) is pb-trace of W^{pb} in the infinite (resp. finite) case. Furthermore, it can be seen from the definitions of *minp*, *maxp* and *hasInfPath* that

- $\mathit{minp}(T_i) \leq |\sigma_i| \leq \mathit{maxp}(T_i)$ for $i \geq 0$ (resp. $0 \leq i < n$) for the infinite (resp. finite) case.
- $\mathit{hasInfPath}(T_n) = \texttt{true}$ in the finite case.
- $\mathit{Val}(T_i) \geq v(s)$ for all $s \in \sigma_i$.

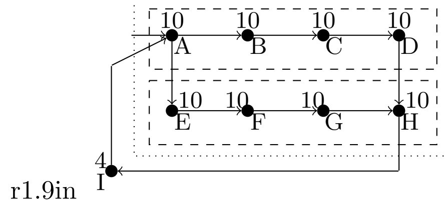


Figure 7.6: System W_2

Therefore, we have that $r = Val(T_0)^{|\sigma_0|}\,Val(T_1)^{|\sigma_1|}\ldots \in B(\pi^{pb})$ (resp. $r = Val(T_0)^{|\sigma_0|}\,Val(T_1)^{|\sigma_1|}\ldots Val(T_n)^{\infty} \in B(\pi^{pb})$) in the infinite (resp. finite) case.

Now, we have $v(\pi) \leq_p r$, and hence $f_t(v(\pi)) \leq f_t(r)$ from the fact that f_t is $\leq_p$-monotonic. From this, we get that $\{v(\pi)\} \sqsubseteq f_t(\bigcup_{\rho \in \pi^{pb}(W^{pb})} B(\rho))$. Hence, by $\sqsubseteq$-monotonicity of f_s, we have $f^{pb}(W^{pb}) \geq f_s(\{f_t(\pi)\}) = f(W)$. The last equality follows from the fact that π is an *ext*-trace of the system W. $\square$

A set of SegmentSets $\mathcal{T}_1$ *refines* a set of SegmentSets $\mathcal{T}_2$ iff for all $T \in \mathcal{T}_2$, there exists a set of SegmentSets $\mathcal{T}$, such that $\mathcal{T} \subseteq \mathcal{T}_1$, and $\mathcal{T}$ covers T.

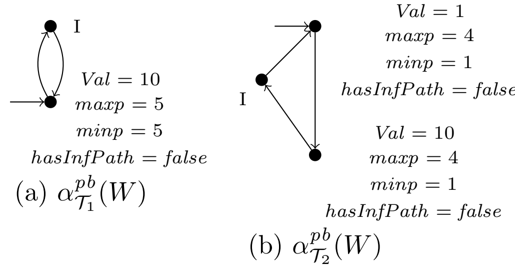
PathBound is a *monotonic over-approximation* for a quantitative property f, if (a) *PathBound* is an over-approximation for f, and (b) if for all $W \in \mathcal{W}(\mathcal{S})$, and for all sets $\mathcal{T}_1$ and $\mathcal{T}_2$ of SegmentSets such that (a) $\mathcal{T}_1$ covers $\Pi(W)$, (b) $\mathcal{T}_2$ covers $\Pi(W)$, and (c) $\mathcal{T}_2$ is a refinement of $\mathcal{T}_1$, we have that $f^{pb}(\alpha^{pb}_{\mathcal{T}_2}(W)) \leq f^{pb}(\alpha^{pb}_{\mathcal{T}_1}(W))$.

The abstraction *PathBound* is not a monotonic approximation in general even for quantitative properties that are $(\leq_p, \sqsubseteq)$-monotonic. We show this by constructing a counterexample (see Example 7.10[Pg. 122]) for which the abstraction is not a monotonic approximation for the limit-average property.

7.4.2 State-equivalence induced segment-based abstraction

Given an equivalence relation on states, we can define a set of SegmentSets. Let $W = (S, \Delta, v, s_\iota)$ be a UWTS, and let $\approx$ be an equivalence relation on states in $\mathcal{S}$. Given an equivalence class e of $\approx$, we can define a corresponding SegmentSet T_e as follows. First, let T'_e be the set of (finite or infinite) segments σ such that all states s that occur in σ are in e. Now we define T_e as the set of maximal segments in T'_e. A segment σ is *maximal* in T_e iff (a) σ is in T'_e; (b) there is a transition $(s_b, \mathit{first}(\sigma)) \in \Delta$ such that $s_b \notin e$; and (c) either $\sigma \in \mathcal{S}^\infty$ or there is a transition $(\mathit{last}(\sigma), s_f) \in \Delta$ such that $s_f \notin e$. Let $\mathcal{T}_\approx$ be a set of SegmentSets defined by $\{T_e \mid e \text{ is an equivalence class of } \approx\}$.

Example 7.8. *Consider again the system W_1 in Figure 7.2[Pg. 115], and the equivalence relation $\approx$ on its states given by the dashed shapes in Figure 7.3[Pg. 116]. The SegmentSets we get from the equivalence classes are given by the nodes T_1 and T_2 in Figure 7.5[Pg. 119]. The set of these SegmentSets is $\mathcal{T}_\approx$. As calculated in Example 7.6[Pg. 120], the value for the abstract system (for the limit-average objective) given by $PathBound_{\mathcal{T}_\approx}$ is $\frac{43}{7}$. Note that this is better (more precise) than the value given by the ExistMax abstraction defined by the*



(a) $\alpha^{pb}_{\mathcal{T}_1}(W)$

(b) $\alpha^{pb}_{\mathcal{T}_2}(W)$

Figure 7.7: Abstractions of system W_2

same equivalence relation $\approx$. As calculated in Example 7.1[Pg. 116], the value given by ExistMax is 10.

Given an equivalence relation $\approx$ on states, the abstraction $\mathit{PathBound}_{\mathcal{T}_\approx}$ gives a better over-approximation for limit-average objective than the $\mathit{ExistMax}_\approx$.

Proposition 7.9. *Let f be a $(\leq_p, \sqsubseteq)$-monotonic quantitative property. Let $\approx$ be an equivalence relation on $\mathcal{S}$ and consider the two abstraction schemes $\mathit{PathBound}_{\mathcal{T}_\approx} = (\mathcal{W}^{pb}, f^{pb}, \alpha^{pb}_{\mathcal{T}_\approx})$ and $\mathit{ExistMax}_\approx = (\mathcal{W}^{em}, f^{em}, \alpha^{em}_\approx)$ parameterized by $\approx$. Then, for all $W \in \mathcal{W}(\mathcal{S})$, $f^{pb}(\alpha^{pb}_{\mathcal{T}_\approx}(W)) \leq f^{em}(\alpha^{em}_\approx(W))$.*

We observe that if $\approx_2$ is a refinement $\approx_1$, then $\mathcal{T}_{\approx_2}$ is a refinement $\mathcal{T}_{\approx_1}$. However, we show an example system where the over-approximation computed using the $\mathcal{T}_{\approx_2}$ abstraction is worse (less precise) than the over-approximation computed using the $\mathcal{T}_{\approx_1}$. This means that, in general, *PathBound* is not a monotonic over-approximation for $(\leq_p, \sqsubseteq)$-monotonic quantitative properties.

Example 7.10. *Consider the system W_2 in Figure 7.6[Pg. 121]. Consider an equivalence relation $\approx$ on states given by the dotted rectangle in the figure (that is all states except the state I are equivalent to each other). This equivalence relation defines a set $\mathcal{T}_1$ of SegmentSets. The resulting abstract system $\alpha^{pb}_{\mathcal{T}_1}(W)$ is in Figure 7.7[Pg. 122] (a). Note that in Figure 7.7[Pg. 122] (a), the node for which the values of maxp, minp, etc. are given corresponds to the dotted rectangle in Figure 7.6[Pg. 121], the other node in the abstract system corresponds to the singleton segment of length one generated from the singleton equivalence class of the node I of system W_2. Consider now a refinement of $\approx$ where the equivalence class of the dotted rectangle is split into two equivalence classes given by the dashed rectangles. The new equivalence relation defines a set $\mathcal{T}_2$ of SegmentSets. The abstract system in Figure 7.7[Pg. 122] (b) results from a refinement where the equivalence class of the dotted rectangle is split into two equivalence classes given by the dashed rectangles. The resulting abstract system $\alpha^{pb}_{\mathcal{T}_2}(W)$ is in Figure 7.7[Pg. 122] (b).*

Let us now assume that the abstract quantitative objective is LimAvg. *The value we get for system $\alpha^{pb}_{\mathcal{T}_1}(W)$ is $((10 \cdot 5 + 4 \cdot 1)/6) = 9$. The value we get for its refinement $\alpha^{pb}_{\mathcal{T}_2}(W)$ is $((10 \cdot 4 + 10 \cdot 4 + 4 \cdot 1)/9) = \frac{84}{9} > 9$. This shows that the estimate is worse (less precise) for the refinement $\alpha^{pb}_{\mathcal{T}_2}(W)$ than for $\alpha^{pb}_{\mathcal{T}_1}(W)$.*

7.4.3 Specialization of *PathBound* abstraction for the limit-average quantitative property

We presented a general definition of the *PathBound* abstraction which is an over-approximation for a large class of quantitative properties. We now specialize the *PathBound* abstraction for the limit-average property by introducing sound optimizations.

We define the limit-average *PathBound*-abstraction (denoted by *PathBoundLA*) scheme as $(\mathcal{W}^{pba}, f^{pba}, \alpha_{\mathcal{T}}^{pba})$. Let W be a system and let $W^{pb} = \alpha_{\mathcal{T}}^{pb}(W) = (S^{pb}, \Delta^{pb}, \mathit{Val}, \mathit{minp}, \mathit{maxp}, \mathit{hasInfPath}, S_0^{pb})$ be a *PathBound* abstraction of W. Furthermore, we fix f to be the limit-average property, i.e., $f_t = \mathrm{LimAvg}$ and $f_s = \sup$ for the remainder of this subsection. We have that $W^{pba} = \alpha_{\mathcal{T}}^{pba}(W) = (S^{pb}, \Delta^{pb}, \mathit{Val}^{pba}, \mathit{minp}, \mathit{maxp}, \mathit{hasInfPath})$ is a *pb*-system similar to $\alpha_{\mathcal{T}}^{pb}(W)$. In the *PathBoundLA* abstraction scheme, we have the following differences:

- $\mathit{Val}^{pba}(T) = \max\{\sup_{\sigma \in (T \cap \mathcal{S}^*)} \frac{\sum_{s \in \sigma} v(s)}{|\sigma|}, \ \sup_{\sigma \in (T \cap \mathcal{S}^\infty)} \mathrm{LimAvg}(\sigma)\}$. Here, we let the value of an abstract SegmentSet T be the supremum of the average weight of the segments in T, rather than the maximum weight occurring in T. Note that if $T \subseteq \mathcal{S}^\infty$, we have $\mathit{Val}^{pba}(T) = \sup(\mathrm{LimAvg}(v(T)))$.
- The abstract quantitative property f^{pba} is defined in the same way as f^{pb} (at the beginning of Section 7.4[Pg. 118]), except that the definition of $B : \Pi(W^\alpha) \to 2^{\mathbb{R}^\omega}$ we have that:
  - $w_0^{n_0} w_1^{n_1} \ldots \in B(\rho)$ if and only if $\rho = T_0 T_1 \ldots$ with $w_i = \mathit{Val}^{pba}(T_i) \wedge 0 < n_i < \infty \wedge n_i \in \{\mathit{minp}(T_i), \mathit{maxp}(t_i)\}$.
  - $w_0^{n_0} w_1^{n_1} \ldots w_n^\infty \in B(\rho)$ if and only if $\forall 0 \leq i \leq n : w_i = \mathit{Val}^{pba}(T_i) \wedge (i < n \implies 0 < n_i < \infty \wedge n_i \in \{\mathit{minp}(T_i), \mathit{maxp}(t_i)\})$.

The above differences between the *PathBoundLA* and *PathBound* can be summarized as follows: (a) the value summarization function for each SegmentSet can be average instead of maximum, and (b) more crucially (from a practical point of view), the evaluation of the abstract property on a *pb*-system can be done by considering only the lengths of the longest and shortest finite paths of an SegmentSet, rather than considering all lengths between them. This is because limit-average is a memoryless property.

The following theorem states that *PathBoundLA* provides a better approximations of the limit-average property than *PathBound*.

Theorem 7.11. *Given a system W and an segment cover $\mathcal{T}$ of $\Pi(W)$ and f being the limit-average property, we have*

$$f(W) \leq f^{pba}(\alpha_{\mathcal{T}}^{pba}(W)) \leq f^{pb}(\alpha_{\mathcal{T}}^{pb}(W))$$

Proof. (a) The proof of the fact that $f(W) \leq f^{pba}(\alpha_{\mathcal{T}}^{pba}(W))$ is similar to the proof of Theorem 7.7[Pg. 120]. The key insight is as follows: let π be an extremal trace of W with $f_t(v(\pi)) = v^*$. Also, let π be composed of the infinite sequence of segments $\pi = \sigma_0 \sigma_1 \ldots$ with each $\sigma_i \in T_i \in \mathcal{T}$. The case where the sequence of segments is finite is simpler.

Consider the abstract trace $\pi^{pba} = T_0 T_1 \ldots$ and the sequence $(w_0, n_0)(w_1, n_1) \ldots \in B'(\pi^{pba})$ with $w_i = \mathit{Val}^{pba}(T_i)$, $n_i = \mathit{maxp}(T_i)$ if $w_i \geq v^*$ and $n_i = \mathit{minp}(T_i)$ if $w_i < v^*$. Intuitively, we take the value of the abstract

trace where we take the longest path in a SegmentSet if its value is high ($\geq v^*$) and shortest paths if its value is low ($< v^*$). It is now easy to show that $\text{LimAvg}(w_0^{n_0} w_1^{n_1} \ldots) > v^*$, we have that $f(W) \leq f_t^{pba}(\pi^{pba}) \leq f^{pba}(\alpha_{\mathcal{T}}^{pba}(W))$, which proves the required inequality.

(b) The inequality $f^{pb^*}(\alpha_{\mathcal{T}}^{pb^*}(W)) \leq f^{pb}(\alpha_{\mathcal{T}}^{pb}(W))$ easily follows from the fact that $Val^{pba}(T) \leq Val^{pb}(T)$ for all T. $\square$

Example 7.12. *Consider again the system W_1 from Figure 7.2[Pg. 115], and the SegmentSets T_1 and T_2 from Figure 7.5[Pg. 119]. In PathBoundLA abstraction, we have $Val(T_1) = (10+1+4+3)/4 = 18/4$ (while the other values for T_1 are as in Figure 7.5[Pg. 119]). For $Val(T_2)$, we have $Val(T_2) = (1+1+1)/3 = 1$. The value of the system is $f^{pba}(\alpha_{\mathcal{T}}(W_1)) = (18+3)/7 = 3$. Recall that for PathBound abstraction, the value f^{pb} for W_1 was calculated in Example 7.6[Pg. 120] to be $\frac{43}{7}$. For PathBoundLA abstraction, we thus get a better approximation than in the PathBound abstraction.*

Evaluating limit-averages on pb-abstract systems. Minimum-mean cycle algorithms compute the limit-average value for a graph with weights on edges. Therefore, from a pb-system, we construct a graph which has weights and lengths on edges, rather than nodes. Intuitively, we consider the graph with edges of the pb-system being the nodes. There are two edges between the node (T_1, T) and (T, T_2): one of weight $maxp(T) \times Val(T)$ and length $maxp(T)$, and another of weight $minp(T) \times Val(T)$ and length $minp(T)$. The node (T_1, T) has the self-loop of weight $Val(T)$ and length 1, if $hasInfPath(T)$ is true. We denote this graph by $W^\dagger$.

Howard's policy iteration was extended in [61] to compute the limit-average values in graphs where edges have both weight and length, as is the case of $W^\dagger$. Howard's policy iteration works by picking a policy (that maps states to successors) and improves the policy as long as possible. Each improvement takes linear time, but only an exponential upper-bound is known on the number of improvements required. However, a number of reports state that only linear number of improvements are required for most cases in practice [61].

7.5 Generalizations of *PathBound* abstractions

In this section, we present two generalizations of the *PathBound* abstraction scheme. The first one generalizes *PathBound* by considering different summaries of SegmentSets rather than set of properties $\{minp, maxp, Val, hasInfPath\}$. The second one generalizes *PathBound* by allowing inductive fixed-point style computations of properties.

7.5.1 Generalized segment-based abstraction

Let W be a UWTS, and let $\mathcal{T}$ be a segment over of $\Pi(W)$. In the *PathBoundLA* abstraction from Section 7.4.3[Pg. 123], we used the values of *maxp*, *minp*, *hasInfPath*, and Val^{pba} to abstract SegmentSets in $\mathcal{T}$. The abstract values are in turn used to compute an over-approximation of the limit-average property for W. In this subsection, we provide a generic segment-based abstraction scheme for any set of properties.

More specifically, let us assume that we have a set $\mathcal{P}$ of quantitative properties, a set $\mathcal{T}$ of SegmentSets that is a segment cover of a segment set T. We provide a generic technique to answer the following question: if we know the values of quantitative properties in $\mathcal{P}$ on all SegmentSets in $\mathcal{T}$, can we compute the values of quantitative properties in $\mathcal{P}$ on SegmentSet T?

We need to extend notation in two ways: (a) We will use quantitative properties f defined by f_t and f_s for both finite and infinite traces. The type of f_t will thus be $\mathbb{R}^\omega \cup \mathbb{R}^* \to \mathbb{R}$; and (b) We will evaluate quantitative properties on a SegmentSet T (instead of a UWTS) by letting $f(T) = f_s(\{f_t(v(\pi)) \mid \pi \in T\})$.

Fix an arbitrary UWTS $W = (S, \Delta, v, s_\iota)$. Consider an arbitrary set of SegmentSets $\mathcal{T} = \{T_0, T_1, \ldots, T_n\}$, where all segments are sequences of states from S. We define the set of *valid segments* generated by $\mathcal{T}$ as $\biguplus^\Delta \mathcal{T} = \{\sigma_0 \ldots \sigma_n \mid \forall j : \sigma_j \in \bigcup \mathcal{T} \wedge \forall j < n : (\mathit{last}(\sigma_j), \mathit{first}(\sigma_{j+1})) \in \Delta\} \cup \{\sigma_0 \ldots \mid \forall j : \sigma_j \in \bigcup \mathcal{T} \wedge \forall j \geq 0 : (\mathit{last}(\sigma_j), \mathit{first}(\sigma_{j+1})) \in \Delta\}$ where $\mathit{first}(\sigma)$ and $\mathit{last}(\sigma)$ denote the first and last states of a segment. Intuitively, the set $\biguplus^\Delta \mathcal{T}$ is the set of segments generated by $\mathcal{T}$ where the transition relation of the system W holds at the sub-segment boundaries.

Let $\mathcal{T} = \{T_1, \ldots, T_n\}$ be a cover of the SegmentSet T (not necessarily $\Pi(W)$). Note that $\biguplus^\Delta \mathcal{T}$ can be a proper subset of T, i.e., $T \subseteq \biguplus \mathcal{T}$, but $\biguplus^\Delta \mathcal{T} \subsetneq T$. We call the SegmentSet $T \cap \biguplus^\Delta \mathcal{T}$ the *strengthening* of the SegmentSet T by $\mathcal{T}$ and Δ. Our question thus becomes: provided that we know the values of quantitative properties in $\mathcal{P}$ on all SegmentSets in $\mathcal{T}$, can we compute the values of quantitative properties in $\mathcal{P}$ on the strengthening of the SegmentSet T by $\mathcal{T}$ and Δ, i.e. on $T \cap \biguplus^\Delta \mathcal{T}$?

Abstract SegmentSet and property domains. Let $\langle SEG, \subseteq \rangle$ be the set of all SegmentSets partially ordered by the subset relation, and let $\langle \mathcal{L}, \preceq \rangle$ be a lattice. The lattice serves as an abstract domain for describing SegmentSets. Elements of L can be for instance syntactical objects, such as formulas in a logic. Let $\langle SEG, \subseteq \rangle \underset{\alpha^{SEG}}{\overset{\gamma^{SEG}}{\leftrightarrows}} \langle \mathcal{L}, \preceq \rangle$ be a Galois connection (see [66]). We call the domain $\langle \mathcal{L}, \preceq \rangle$ the *abstract SegmentSet domain* and each element $\phi \in \mathcal{L}$ an *abstract SegmentSet*.

A *property set* $\mathcal{P}$ is a tuple $\langle \langle f_1^l, \ldots f_n^l \rangle, \langle f_1^u, \ldots f_m^u \rangle \rangle$ where (a) all f_i^l's are quantitative properties where $f_s = \inf$; and (b) all f_i^u's are quantitative properties where $f_s = \sup$. We define the corresponding property domain $\mathcal{D}_\mathcal{P} = \langle (\mathbb{R}^n \times \mathbb{R}^m), \sqsubseteq_\mathcal{P} \rangle$ to be the abstract domain where $((a_1^l, \ldots, a_n^l), (a_1^u, \ldots, a_n^u)) \sqsubseteq ((b_1^l, \ldots, b_n^l), (b_1^u, \ldots, b_n^u))$ if and only if all $a_i^l \geq b_i^l$ and $a_i^u \leq b_i^u$. We write $\mathcal{P}(T)$ for $((f_1^l(T), \ldots, f_n^l(T)), (f_1^u(T), \ldots, f_n^u(T)))$.

Example 7.13. *The set $\mathcal{P}^{LA} = \langle \langle \mathit{minp} \rangle, \langle \mathit{maxp}, \mathit{Val}^{pba}, \mathit{hasInfPath} \rangle$ is an property set. For example, $\mathit{minp}(T) = \inf(\{|\sigma| \mid \sigma \in T\})$ and $\mathit{hasInfPath}(T) = \sup(\{\mathit{hasInfPath}_t(\sigma) \mid \sigma \in T\})$ where $\mathit{hasInfPath}_t(\sigma)$ is 1 if $|\sigma| = \infty$ and 0 otherwise.*

It is easy to see that a Galois connection $\langle SEG, \subseteq \rangle \underset{\alpha^\mathcal{P}}{\overset{\gamma^\mathcal{P}}{\leftrightarrows}} \langle \mathcal{D}_\mathcal{P}, \sqsubseteq_\mathcal{P} \rangle$ can be defined by letting $\alpha^\mathcal{P}(T) = \mathcal{P}(T)$ and $\gamma^\mathcal{P}(P) = \bigcup \{T \mid \mathcal{P}(T) \sqsubseteq P\}$. Intuitively, $((l_1, \ldots, l_n), (u_1, \ldots, u_n)) \in \mathcal{P}(T)$ represents the largest SegmentSet T that respects the lower and upper bounds placed by P, i.e., $f_i^l(T) \geq l_i \wedge f_j^u(T) \leq u_j$.

We call $\mathcal{D}_\mathcal{P}$ the *property bound domain* and an individual $P \in \mathcal{D}_\mathcal{P}$ a *property bound*.

Let $\langle \mathcal{L} \times \mathcal{D}_\mathcal{P}, \leq \rangle$, where $\leq$ is $\preceq \times \sqsubseteq_\mathcal{P}$, be the product of $\langle \mathcal{L}, \preceq \rangle$ and $\langle \mathcal{D}_\mathcal{P}, \sqsubseteq_\mathcal{P} \rangle$. We call $\mathcal{L} \times \mathcal{D}_\mathcal{P}$ the domain of abstract bound pairs and each element (written as $\phi \wedge P$) an abstract bound pair. Let $\langle SEG, \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{L} \times \mathcal{D}_\mathcal{P}, \preceq \times \sqsubseteq_\mathcal{P} \rangle$ be a Galois connection naturally defined for the product of abstract domains, where $\alpha(T) = (\alpha^{SEG}(T), \alpha^\mathcal{P}(T))$, and $\gamma(\phi, P) = \gamma^{SEG}(\phi) \cap \gamma^\mathcal{P}(P)$. Intuitively, the element $\phi \wedge P$ represents a SegmentSet that is contained in ϕ and respects the property bounds P. We identity abstract segments $\phi \in \mathcal{L}$ with the abstract bound pair in $\phi \wedge \top$ where $\top = ((-\infty, \ldots, -\infty), (\infty, \ldots, \infty))$, i.e., there are no bounds on any of the properties in $\mathcal{P}$.

Example 7.14. *Let $\mathcal{L} = SEG$, i.e., the abstract SegmentSets are the same as concrete SegmentSets. Note that this assumption is to simplify the example. It would be more natural to use syntactical objects (e.g. formulas in some logic) for $\mathcal{L}$. Consider $\mathcal{D}_\mathcal{P}^{LA}$ for the property bound domain defined in Example 7.13[Pg. 125]. An example of an element in $\mathcal{L} \times \mathcal{D}_\mathcal{P}^{LA}$ is $T_1 \wedge ((4), (4, \frac{18}{4}, 0))$ where T_1 is from Example 7.12[Pg. 124]. Here, $((4), (4, \frac{18}{4}, 0))$ represent bounds on values of properties $((minp), (maxp, Val^{pba}, hasInfPath))$. Note that $T_1 \wedge ((4), (4, \frac{18}{4}, 0))$ contains exactly the same information that PathBoundLA stores about a SegmentSet.*

Evaluating quantitative properties on abstract SegmentSets. In what follows, we fix a UWTS W, and a segment cover $\mathcal{T} = \{T_1, T_2, \ldots, T_{\|\mathcal{T}\|}\}$ of $T = \Pi(W)$.

Let K be the interval $[1..\|\mathcal{T}\|]$. Given an abstract bound pair domain $\langle \mathcal{L} \times \mathcal{D}_\mathcal{P}, \preceq \times \sqsubseteq_\mathcal{P} \rangle$, where f is a property in $\mathcal{P}$, we can perform the computation of $f(T \cap \biguplus^\Delta \mathcal{T})$ in the abstract domain. For this computation, the abstract bound pair domain needs to support the following additional operations.

Let $\phi \wedge P$ be an abstract bound pair, and $\Phi = \{\phi_1 \wedge P_1, \ldots, \phi_{\|\mathcal{T}\|} \wedge P_{\|\mathcal{T}\|}\}$ be a set of abstract bound pairs. The abstract bound pairs domain $\langle \mathcal{L} \times \mathcal{D}_\mathcal{P}, \preceq \times \sqsubseteq_\mathcal{P} \rangle$ is an *inductive domain* if it supports the following operations in addition to the standard lattice operations.

- *Transition check.* This operation checks for two abstract SegmentSets, whether a segment from the first can be validly followed by a segment from the second. Formally, $TrCheck_\phi(\phi_i, \phi_j)$ is true if and only if there is validly generated segment σ, i.e., $\sigma \in \gamma(\phi) \cap \biguplus^\Delta \{\gamma(\phi_i) \mid i \in K\}$, where $\sigma = \sigma' \sigma_i \sigma_j \sigma''$ such that (a) $\sigma_i \in \gamma(\phi_i) \wedge \sigma_j \in \gamma(\phi_j)$, and (b) σ' and σ'' are also validly generated.
- *Reduce Property Bounds.* Given an abstract SegmentSet ϕ, compute (an over-approximation) of the property bounds on ϕ. Formally, $ReduceBound(\phi)$ returns $\phi \wedge P^*$ such that $\gamma(\phi \wedge P^*) \supseteq \gamma(\phi)$.
- *Property computation.* Given only the bounds P_i on abstract SegmentSets ϕ_i and the values $TrCheck_\phi(\phi_i, \phi_j)$, $PropComp$ computes (an over-approximation) of the property bounds on the abstract segment $\phi \wedge P$. Formally, given $\Delta^{gpb} = \{(i, j) \mid TrCheck_\phi(\phi_i, \phi_j) = true\}$, and the values P_i for all i, the function $PropComp(\phi, \Delta^{gpb}, \langle P_1 \ldots P_{\|\mathcal{T}\|}\rangle)$ outputs $\phi^* \wedge P^*$ such that $\phi^* \wedge P^* \geq (\phi \wedge \top) \wedge \alpha(\biguplus^\Delta \{\gamma(\phi_i \wedge P_i) \mid i \in K\})$. Note that we sometimes abuse the notation and write $PropComp(\phi, \{\phi_1 \wedge P_1, \ldots, \phi_{\|\mathcal{T}\|} \wedge$

$P_{\|\mathcal{T}\|}\}$) instead of $PropComp(\phi, \Delta^{gpb}, \langle P_1 \dots P_{\|\mathcal{T}\|}\rangle_i)$.

Example 7.15. *Continuing from Example 7.14[Pg. 126], i.e., $\mathcal{L} = SEG$ and the system under consideration is W_1 from Figure 7.5[Pg. 119].*

- *$TrCheck_{\Pi(W)}(T_1, T_2)$ can be as precise as the transition relation of PathBoundLA, i.e, $(T_1, T_2) \in \Delta^{pba} \Leftrightarrow TrCheck_{\Pi(W)}(T_1, T_2)$. Suppose $\sigma_1 \in T_1$ and $\sigma_2 \in T_2$. We have that σ_1 can be followed by σ_2 if and only if $(last(\sigma_1), first(\sigma_2)) \in \Delta$. This is the condition that defines $(T_1, T_2) \in \Delta^{pb}$.*
- *If we take $ReduceBound(T_2)$ to be precise (and computationally expensive) procedure that concretizes abstract states, it can return $T_2 \wedge ((3), (4, 1, 0))$, i.e., it computes (an over-approximation) of the information stored for the particular SegmentSetT_2.*
- *$PropComp(\Pi(W), \{T_1 \wedge P_1, T_2 \wedge P_2\})$ (where $P_1 = ((4), (4, \frac{18}{4}, 0))$ and $P_2 = ((3), (4, 1, 0))$) computes an over-approximation of $\mathcal{P}(\Pi(W))$ by computing on abstract states. If PropComp is defined using the same approach as the B and f^{pba} definitions from Section 7.4.3[Pg. 123], it returns $((4), (\infty, 3, 1))$. Here, $Val^{pba}(\Pi(W))$ is 3, $minp = 4$, and maxp and hasInfPath are over-approximated to the largest possible values.*

Remark 7.16. *Note that in the arguments of PropComp, we do not require P_i to be equal to $\mathcal{P}(\phi_i)$. In fact, even in the case where P_i is a $\sqsubseteq_{\mathcal{P}}$-over-approximation of $\mathcal{P}(\phi_i)$, the procedure PropComp is required to produce a valid $\sqsubseteq_{\mathcal{P}}$-over-approximation of the $\mathcal{P}$ value of $\gamma(\phi) \cap \biguplus^{\Delta}\{\gamma(\phi_i) \mid i \in K\}$.*

We call an abstract bound pair domain *effective* if: (a) each of the operations *ReduceBound*, *TrCheck*, and *PropComp* can be computed effectively, i.e., by a terminating procedure; and (b) *ReduceBound* and *PropComp* are monotonic, i.e., giving more precise inputs produces more precise outputs. Formally, (a) $\phi \preceq \phi' \implies ReduceBound(\phi) \leq ReduceBound(\phi')$; (b) $(\phi \preceq \phi' \wedge \forall i \in K : P_i \sqsubseteq P'_i \wedge \phi_i \preceq \phi'_i) \implies PropComp(\phi, \{\phi_1 \wedge P_1, \dots, \phi_{\|\mathcal{T}\|} \wedge P_{\|\mathcal{T}\|}\}) \leq PropComp(\phi, \{\phi_1 \wedge P'_1, \dots, \phi_{\|\mathcal{T}\|} \wedge P'_{\|\mathcal{T}\|}\})$; and (c) $(\phi^1_1 \wedge P^1_1 \leq \phi_1 \wedge P_1) \wedge (\phi^2_1 \wedge P^2_1 \leq \phi_1 \wedge P_1) \implies PropComp(\phi, \{\phi^1_1 \wedge P^1_1, \phi^2_1 \wedge P^2_1, \phi_2 \wedge P_2, \dots, \phi_{\|\mathcal{T}\|} \wedge P_{\|\mathcal{T}\|}\}) \leq PropComp(\phi, \{\phi_1 \wedge P_1, \phi_2 \wedge P_2, \dots, \phi_{\|\mathcal{T}\|} \wedge P_{\|\mathcal{T}\|}\})$.

We can now generalize *PathBound* abstraction scheme by letting the summaries of SegmentSets be the set of values of properties from any effectively inductive quantitative property set. Intuitively, given a UWTS W, and a property set $\mathcal{P}$, the $PathBound[\mathcal{L} \times \mathcal{D}_{\mathcal{P}}]$ abstraction stores with each SegmentSet T in the cover of $\Pi(W)$ the values $\alpha^{SEG}(T) \wedge \mathcal{P}(T)$. To compute the value of the abstract system, the procedures *ReduceBound*, *TrCheck*, and *PropComp* are used.

Formally, $PathBound[\mathcal{L} \times \mathcal{D}_{\mathcal{P}}]_{\mathcal{T}} = \langle \mathcal{W}[\mathcal{L} \times \mathcal{D}_{\mathcal{P}}], \alpha[\mathcal{L} \times \mathcal{D}_{\mathcal{P}}]_{\mathcal{T}}, f[\mathcal{L} \times \mathcal{D}_{\mathcal{P}}]\rangle$ is defined as:

- $\alpha[\mathcal{L} \times \mathcal{D}_{\mathcal{P}}]_{\mathcal{T}}(W) = \langle \phi, \langle \phi_1, \dots, \phi_{\|\mathcal{T}\|}\rangle, \Delta^{gpb}\rangle$ where (a) $\phi = \alpha^{SEG}(\Pi(W))$; (b) $\phi_i = \alpha^{SEG}(T_i)$; and (c) $(i, j) \in \Delta^{gpb}$ if and only if $TrCheck_{\phi}(\phi_i, \phi_j)$ returns true.
- $f[\mathcal{L} \times \mathcal{D}_{\mathcal{P}}]$ is computed as the value of f in P^* where $\phi^* \wedge P^* = PropComp(\phi, \Delta^{gpb}, \langle ReduceBound(\phi_1), \dots, ReduceBound(\phi_{\|\mathcal{T}\|})\rangle) \wedge ReduceBound(\phi)$.

Precise abstractions. In the *ExistMax* case, if the abstraction function does not lose any information, i.e., if the equivalence relation used is the identity

relation, the abstract system value is the same as the concrete system value. We give the conditions when an segment-based abstraction does not lose any information either.

Intuitively, we want the property computation for a SegmentSet T from a cover $\mathcal{T} = \{T_1, \dots T_n\}$ to be accurate when T is exactly covered by $\mathcal{T}$, i.e, $T = \biguplus^{\Delta} \mathcal{T}$.

An quantitative property set is *precisely inductive* if there exists a *PropComp* procedure which produces the output $(\phi, \mathcal{P}(\phi))$ when the following hold: (a) $P_i = \mathcal{P}(\gamma(\phi_i))$, and (b) $\gamma(\phi) = \biguplus^{\Delta} \{\gamma(\phi_i) \mid i \in K\}$.

Example 7.17. *The limit-average property is not precisely inductive, i.e., by knowing only the limit-average value of the subsegments of a segment, we cannot estimate the limit-average accurately without knowing the length of the subsegments. However, strengthening it to the property set* $\langle\langle minp\rangle, \langle maxp, \mathrm{LimAvg}, hasInfPath\rangle\rangle$ *makes it precisely inductive.*

7.5.2 Hierarchical segment-based abstraction

Effective abstract bound pair domains allow computation of property bounds for a whole set of properties on a SegmentSet from the property bounds on each element of the cover. For example, if $\mathcal{T}$ covers T, the four properties *minp*, *maxp*, *Val*, *hasInfPath* of SegmentSets in $\mathcal{T}$ are used to compute not only the LimAvg, but also the *maxp*, *minp*, and *hasInfPath* values of the SegmentSet T too. This leads to the possibility of computing these properties hierarchically for a multi-level trace segment cover.

An *inductive trace segment cover* [67] $\mathcal{C}$ is a finite rooted-tree where the nodes are labelled with abstract bound pairs such that for every non-leaf node labelled with SegmentSet $\phi \wedge P$, and $\phi_1 \wedge P_1, \dots, \phi_n \wedge P_n$ the set of labels of its children, $\{\gamma(\phi_1 \wedge P_1), \dots, \gamma(\phi_n \wedge P_n)\}$ is a segment cover of $\gamma(\phi \wedge P)$. An inductive trace segment cover $\mathcal{C}$ *inductively covers* a SegmentSet T if $T \subseteq \gamma(root(\mathcal{C}))$. Similarly, $\mathcal{C}$ inductively covers a system W if $\Pi(W) \subseteq \gamma(root(\mathcal{C}))$.

We can now introduce an abstraction scheme $HPathBound[\mathcal{L} \times \mathcal{D}_{\mathcal{P}}]_{\mathcal{C}} = \langle \mathcal{W}^{hpb}[\mathcal{L} \times \mathcal{D}_{\mathcal{P}}], \alpha^{hpb}[\mathcal{L} \times \mathcal{D}_{\mathcal{P}}]_{\mathcal{C}}, f^{hpb}[\mathcal{L} \times \mathcal{D}_{\mathcal{P}}]\rangle$ parameterized by an abstract bound pair domain $\mathcal{L} \times \mathcal{D}_{\mathcal{P}}$, and an abstract inductive trace segment cover $\mathcal{C}$. Intuitively, *HPathBound* stores for each internal node a of $\mathcal{C}$, a $PathBound[\mathcal{L} \times \mathcal{D}_{\mathcal{P}}]_{\mathcal{T}(a)}$ abstraction $\langle label(a), \mathcal{T}(a), \Delta^{gpb}\rangle$. The abstract trace segment cover $\mathcal{T}(a)$ for this abstraction is the labels of the set *children*(a).

Abstract hierarchical traces. We fix an abstract bound pair domain $\mathcal{L} \times \mathcal{P}_{\mathcal{D}}$. An *abstract hierarchical trace* $\psi = \langle(\phi_0 \wedge P_0)(\phi_1 \wedge P_1) \dots, sub_{\psi}\rangle$ consists of (a) a finite or infinite sequence of abstract bound pairs; and (b) a partial function sub_{π} from $\mathbb{N}$ to hierarchical traces.

The concrete traces $\gamma(\psi)$ corresponding to a given abstract trace $\psi = \langle(\phi_0 \wedge P_0)(\phi_1 \wedge P_1) \dots, sub_{\psi}\rangle$ are defined as $\{\sigma_0\sigma_1 \dots \| \forall i : \sigma_i \in \gamma(\phi_i \wedge P_i) \wedge sub_{\psi}(i) \neq \bot \implies \sigma_i \in \gamma(sub_{\psi}(i))\}$. Intuitively, a concrete trace of an abstract hierarchical trace ψ is made of segments σ_i from $\phi_i \wedge P_i$, with the additional condition that $\sigma_i \in sub_{\psi}(i)$ if $sub_{\psi}(i)$ is defined.

Given a property set $\mathcal{P}$, an effective abstract bound pair domain, and an inductive trace segment cover $\mathcal{C}$ of system W, we inductively compute property bounds (and hence, abstract system values) using Algorithm 3[Pg. 129]. The algorithm is based on the inductive proof method presented in [67]. It can be

Algorithm 3 Inductive Property Computation (InductiveCompute)

Input: UWTS W,
Effective abstract bound pair domain $\mathcal{L} \times \mathcal{D}_{\mathcal{P}}$,
Abstract Inductive Segment Cover $\mathcal{C}$ (labelled from $\mathcal{L} \times \mathcal{D}_{\mathcal{P}}$),
Output: abstract bound pair $\phi^* \wedge P^*$ such that $(\phi^* \wedge P^*) \geq \mathcal{P}(label(root(\mathcal{C})))$

1: $(\phi \wedge P) \leftarrow label(root(\mathcal{C}))$
2: **if** $root(\mathcal{C})$ has no children **then**
3: **return** $(\phi \wedge P) \wedge (ReduceBound(\phi \wedge P))$
4: **else**
5: $subTrees \leftarrow$ top level sub-trees of $\mathcal{C}$
6: **return** $(\phi \wedge P) \wedge PropComp(\phi \wedge P, \{InductiveCompute(\mathcal{C}') \mid \mathcal{C}' \in subTrees\})$

rewritten as a fixed-point computation in the lattice $\mathcal{C} \to \mathcal{L} \times \mathcal{D}_{\mathcal{P}}$ of maps from $nodes(\mathcal{C})$ to $\mathcal{L} \times \mathcal{D}_{\mathcal{P}}$ (the lattice ordered point-wise by $(\preceq, \sqsubseteq_{\mathcal{P}})$). At the final line in Algorithm 3[Pg. 129], the value of $\mathcal{C}$ will be the least fixed-point of the function which replaces each node in $\mathcal{C}$ with a best approximation obtained from among the current value of the node, and (a) *ReduceBound* applied on the node if it is a leaf node, or (b) *PropComp* applied on the node and its children if it is an internal node.

Monotonicity. We define refinements of *HPathBound* abstractions using refinement steps. Let $\mathcal{C}$ be an inductive trace segment cover and let a be a non-root node in $\mathcal{C}$, b be its parent, $\mathcal{C}(a)$ be the sub-tree of $\mathcal{C}$ rooted at b, and $\phi \wedge P$ and $\phi^b \wedge P^b$ the labels of a and b. A *one-step refinement* of $\mathcal{C}$ is one of the following:

- Horizontal refinement. Let ϕ_1 and ϕ_2 be such that $\gamma(\phi \wedge P) = \gamma(\phi_1 \wedge P_1) \cup \gamma(\phi_2 \wedge P_2)$. Let $\mathcal{C}(a)[label(a) \leftarrow \phi_1 \wedge P_1]$ and $\mathcal{C}(a)[label(a) \leftarrow \phi_2 \wedge P_2]$ be the tree $\mathcal{C}(a)$ with $\phi \wedge P$ replaced by $\phi_1 \wedge P_1$ and $\phi_2 \wedge P_2$, respectively. Let the tree $\mathcal{C}'$ be obtained by detaching $\mathcal{C}(a)$ from p and then attaching $\mathcal{C}(a)[label(a) \leftarrow \phi_1 \wedge P_1]$ and $\mathcal{C}(a)[label(a) \leftarrow \phi_2 \wedge P_2]$ to p. Then $\mathcal{C}'$ is a *one-step horizontal refinement* of $\mathcal{C}$.
- Vertical splitting refinement. Suppose $\{\gamma(\phi_1 \wedge P_1), \ldots \gamma(\phi_n \wedge P_n)\}$ cover $\phi \wedge P$ and that a does not have any children. Suppose $\mathcal{C}'$ is obtained from $\mathcal{C}$ by adding the children with labels $\phi_1 \wedge P_1, \ldots, \phi_n \wedge P_n$ to a. Then, $\mathcal{C}'$ is a *one-step vertical splitting refinement* of $\mathcal{C}$.
- Vertical joining refinement. Suppose that b has no grandchildren, and let $(\phi^b \wedge P^b) \leq PropComp(\phi \wedge P, children(b), \Delta)$ and $\forall c \in children(b) : label(c) \leq ReduceBound(label(c))$. If $\mathcal{C}'$ is the tree obtained by removing all the children of b in $\mathcal{C}$, then $\mathcal{C}'$ is a *one-step vertical joining refinement* of $\mathcal{C}$.
- Downward strengthening refinement. Suppose $\gamma(\phi') \subseteq \gamma(\phi)$ and $\phi^b \wedge P^b \subseteq \biguplus^{\Delta}\{\gamma(\psi) \mid \psi \in ((children(b) \setminus \{\phi \wedge P\}) \cup \{\phi' \wedge P'\})\}$. Let $\mathcal{C}'$ be the tree obtained by replacing $\mathcal{C}(a)$ by $\mathcal{C}(a)[label(a) \leftarrow \phi' \wedge P']$. Then, $\mathcal{C}'$ is a *one-step downward strengthening refinement* of $\mathcal{C}$.
- Upward strengthening refinement. Suppose that $\phi' \wedge P'$ is such that $\biguplus^{\Delta}\{\gamma(\psi) \mid \psi \in children(b)\} \subseteq \gamma(\phi' \wedge P') \subseteq \gamma(\phi^b \wedge P^b)$. If $\mathcal{C}'$ is obtained from $\mathcal{C}$ by replacing $\phi^b \wedge P^b$ with $\phi' \wedge P'$, then $\mathcal{C}'$ is a *one-step upward*

strengthening refinement of $\mathcal{C}$.

A $\mathcal{C}_n$ is a *refinement* of $\mathcal{C}_0$ if there exists a sequence $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_{n-1}$ such that $\mathcal{C}_i$ is a one-step refinement of $\mathcal{C}_{i-1}$ for all i such that $n \geq i > 0$.

HPathBound$[\mathcal{L} \times \mathcal{D}_\mathcal{P}]$ is *monotonic* if for all systems W and abstract inductive trace segment covers $\mathcal{C}$ and $\mathcal{C}'$ of W, if $\mathcal{C}'$ is a refinement of $\mathcal{C}$, then abstract value $f^{hpb}[\mathcal{L} \times \mathcal{D}_\mathcal{P}](\alpha^{hpb}[\mathcal{L} \times \mathcal{D}_\mathcal{P}]_{\mathcal{C}'}(W)) \leq f^{hpb}[\mathcal{L} \times \mathcal{D}_\mathcal{P}](\alpha^{hpb}[\mathcal{L} \times \mathcal{D}_\mathcal{P}]_{\mathcal{C}}(W))$.

Theorem 7.18. *If f is a property such that $f_s = \sup$, and the inductive property set $\mathcal{P}$ contains f, and $\mathcal{L} \times \mathcal{D}_\mathcal{P}$ is effectively inductive, HPathBound$[\mathcal{L} \times \mathcal{D}_\mathcal{P}]$ is monotonic for f.*

Intuitively, the theorem holds as every one-step refinement preserves the information about the property-bounds of nodes from the previous iteration. For all one-step refinements other than the vertical joining refinements, the monotonicity follows from the monotonicity of *PropComp* and *ReduceBound*. For vertical splitting one-step refinements, the monotonicity holds as all the information that can be generated from the deleted children is already present in the parent node.

7.5.3 *HPathBound* abstractions for control-flow graphs

A control-flow graph of a program written in a high-level language is contains many structures like loops and function calls. The traces of transition systems produced from such programs are structured. This opens the possibility of using hierarchical segment-based abstractions to analyze them.

Control-flow graphs and Programs. Following, e.g., [62], we abstract away from the syntax of a concrete programming language, and model programs as graphs whose nodes correspond to program locations. Here we assume simple programs with inlined function calls (this implies that there are no recursive functions).

Let V be a set of variables and let $D(V)$ be the combined range of variables in V. A *control-flow graph* (CFG) is a tuple of $\langle L, V, \Delta, \psi : \Delta \to 2^{D(V) \times D(V)} \rangle$ where L is a finite set of control locations, Δ is a transition relation, V is a set of variables, and ψ is a function from Δ to assertions over program variables V and their primed versions V'. The primed variables V' refer to the values of the variables after executing the transition. We assume that V contains a special variable wt ranging over $\mathbb{R}$ which denotes the weight of a particular state.

Given a CFG C, a corresponding transition system can be generated (with state-space $L \times D(V)$) in a standard way. We add the weight function $v(l, d) = w$, where w is the value of the special variable wt in d, to turn the transition system into a UWTS.

We assume the following about the CFG (these assumptions are valid for CFGs generated for programs in most programming languages): (a) *Reducibility.* In the graph (L, Δ), every maximal strongly connected component has a single entry and exit point, i.e., if $G \subseteq L$ is a maximal strongly connected component, there exists a node l_G such that $(l, l') \in \Delta \cap ((L \setminus G) \times G) \implies l' = l_G \wedge (l, l') \in \Delta \cap (G \times (L \setminus G)) \implies l = l_G$; (b) *Recursive reducibility.* Suppose $G \subseteq L$ is a maximal strongly connected component. The graph $(G, \Delta \cap (G \times (G \setminus \{l_G\}))$ is also reducible. Intuitively, the above conditions imply that loops in the CFG are single-entry and single-exit, and that they are nested.

```
while(true)
   b = false;
   j = 10;
   while j > 0
      b = not b;
      if b
         costlyOp;
      j--
   k = 10;
   while (k>0)
      k--
```

Figure 7.8: Sample Program

A *hierarchical control-flow graph* (HCFG) is a graph $\langle \mathcal{H}, \Delta \subseteq \mathcal{H} \times \mathcal{H}, h_i \rangle$ where each node $h \in \mathcal{H}$ is either a control-flow location, or a hierarchical control-flow graph. We call the first kind of nodes, *abstract state nodes* and the second kind of nodes *subgraph nodes*.

Any recursively reducible CFG C can be converted into a HCFG H of a special form using a standard algorithm on reducible graphs. In this special form, H and all sub-HCFGs occurring in H have the following property: either they are acyclic, or they have a single node with a self-loop on itself. Note that each loop of a program will correspond to such sub-HCFG. In what follows, we assume that all our HCFGs are of this special form.

Example 7.19. *Consider the example CFG shown in Figure 7.9[Pg. 133] (generated from program in Figure 7.8[Pg. 131]). The equivalent HCFG of the special form to this CFG is shown in Figure 7.10[Pg. 133]. The statements on the transitions in Figure 7.10[Pg. 133] are omitted for the sake of clarity. Each of the dotted boxes represent a HCFG. Intuitively, each loop has been separated out into an acyclic CFG containing the loop body and a single self-loop CFG.*

Inductive trace segment covers can be derived from HCFGs. In particular, every HCFG H represents a unique inductive trace segment cover $\mathcal{C}(H)$. Intuitively, the root of the inductive trace segment cover is the set of all traces of H, and the children of the root node are either (a) $\mathcal{C}(H')$ if H' is a subgraph node of H, and (b) $\{W\}$ if W is a abstract state node of H where $\{W\}$ is the SegmentSet containing all segments of length 1 with states in W. From now on, we use HCFGs and abstract inductive trace segment covers interchangeably.

Example 7.20. *Consider the program in Figure 7.8[Pg. 131]. Its CFG is in Figure 7.9[Pg. 133] and the corresponding HCFG is in Figure 7.10[Pg. 133]. We use regular expressions over control locations as our abstract SegmentSet domain $\mathcal{L}$. The regular expressions have their intuitive meaning. For example, (a) the expression $s_1 s_2$ represents all segments $s_1 s_2$ such that the control location of s_1 (resp. s_2) is s_1 (resp. s_2); and (b) the expression $(s_1 s_2)^*$ represents the set of segments obtained by concatenation of segments from $s_1 s_2$.*

The HCFG corresponds to the following inductive trace segment cover $\mathcal{C}$ of the traces generated by the CFG. The root of $\mathcal{C}$ is the expression $H = (s_1 s_2 s_3 (s_4 s_5 (s_6 + s_7) s_3)^* s_8 s_9 (s_{10} s_9)^*)^*$. Its only child is $F = s_1 s_2 s_3 (s_4 s_5 (s_6 +$

$s_7)s_3)^*s_8s_9(s_{10}s_9)^*$. *The children of F are $\{C_B, L_1, C_M, L_2\}$ where $C_B = s_1s_2s_3$, $L_1 = (s_4s_5(s_6 + s_7)s_3)^*$, $C_M = s_8s_9$, and $L_2 = (s_{10}s_9)^*$. Of these, only L_1 and L_2 are non-leaf nodes having one child each $C_{L_1} = s_4s_5(s_6 + s_7)s_3$ and $C_{L_2} = s_{10}s_9$, respectively.*

Evaluating LimAvg **for** ***HPathBound*** **abstractions induced by HCFGs.** Let H be a HCFG of a special form. We compute the values *maxp*, *minp*, Val^{pba}, and *hasInfPath* using the technique of computing loop bounds. We consider a the domain of abstract bound pairs with elements of the form $\phi \wedge P$ as before.

Suppose H is a HCFG with a single node and a self-loop. Furthermore, let $\phi \wedge P$ be the corresponding abstract SegmentSet property bound in $\mathcal{C}(H)$, and let $\{\phi_1 \wedge P_1, \ldots, \phi_n \wedge P_n\}$ be its children. Let segment σ be in $\gamma(\phi \wedge P) \cap \biguplus^{\Delta}\{\gamma(\phi_1 \wedge P_1), \ldots, \gamma(\phi_n \wedge P_n)\}$. Let $iters(\sigma) \subseteq \mathbb{N}$ where $n \in iters(\sigma)$ if and only if there exist $\sigma_0, \ldots, \sigma_{n-1}$ such that $\forall j < n. \exists i : \sigma_j \in \gamma(\phi_i \wedge P_i)$. Let $Iters(H) = \bigcup iters(\sigma)$ for all such σ. We define the *upper loop bound* (resp. *lower loop bound*), denoted by $ulb(H)$ (resp. $llb(H)$) as the value $\sup Iters(H)$ (resp. $\inf Iters(H)$). Note that there exists techniques to compute loop bounds using for example relational abstractions and ranking functions, see for example [90], and references therein.

Now, given the values of *maxp*, *minp*, Val^{pba}, and *hasInfPath* of subgraphs of an HCFG H, we inductively compute the value of the properties for H as follows:

- If H is a single node with no self-loop, i.e., an abstract state node, we have $maxp(H) = minp(H) = 1$, $Val^{pba}(H) = v(H)$, and $hasInfPath(H) = false$.
- If H is acyclic, we construct the following graph $H^\dagger$ as in Section 7.4.3[Pg. 123], i.e., edges of H correspond to nodes of $H^\dagger$, and every node of H corresponds to two edges of weights (resp. lengths) $maxp(H') \cdot Val^{pba}(H')$ and $maxp(H') \cdot Val^{pba}(H')$ (resp. $maxp(H')$ and $minp(H')$). Now, $maxp(H)$ and $minp(H)$ are equal to the length of the longest and shortest paths in $H^\dagger$. Also, $hasInfPath(H) = true$ if and only if there is a node H' in H with $hasInfPath(H') = 1$. The value $Val^{pba}(H)$ can be evaluated using Howard's policy iteration as in Section 7.4.3[Pg. 123].
- If H is a single node with a self-loop, there is only one subgraph of H (say H'). We have $maxp(H) = ulb(H) \cdot maxp(H')$, $llb(H) \cdot minp(H')$, $Val^{pba}(H) = Val^{pba}(H')$, and $hasInfPath(H) = hasInfPath(H') \vee ulb(H) = \infty$.

Thus for such an HCFG, we can evaluate the limit-average value by using the inductive evaluation scheme from Section 7.5.2[Pg. 128], i.e., using the values *minp*, *maxp*, Val^{pba} and *hasInfPath*, we can inductively compute the limit-average values for an HCFG.

Example 7.21. *Consider the HCFG in Figure 7.10[Pg. 133] and its inductive cover $\mathcal{C}$ from Example 7.20[Pg. 131]. We will illustrate a few steps of the inductive computation of the properties on $\mathcal{C}$. Fix $\mathcal{P} = \langle\langle minp\rangle, \langle maxp, Val^{pba}, hasInfPath\rangle\rangle$.*

Let us start for the leaves of $\mathcal{C}$, i.e., $C_B = s_1s_2s_3$, $C_M = s_8s_9$, $C_{L1} = s_4s_5(s_6 + s_7)s_3$, and $C_{L2} = s_{10}s_9$. Now, we can compute minp and maxp for these as the shortest and longest paths in the corresponding HCFGs. Furthermore, the hasInfPath values for each of these is 0 *as all of these CFGs*

Figure 7.9: Nested loop program

Figure 7.10: HCFG of a program

are acyclic. The Val^{pba} of all the leaves except C_{L1} is 0 as the weights of all nodes except s_7 are 0. For C_{L1}, the Val^{pba} can be computed to be $\frac{10}{4}$. Therefore, ReduceBound$(C_B) = ((3),(3,0,0))$, ReduceBound$(C_M) = ((2),(2,0,0))$, $P_1 =$ ReduceBound$(C_{L1}) = ((4),(4,\frac{10}{4},0))$, and $P_2 =$ ReduceBound$(C_{L2}) = ((2),(2,0,0))$.

Now, the nodes $L_1 = C^*_{L1}$ and $L_2 = C^*_{L2}$ have the children $\{C_{L1}\}$ and $\{C_{L2}\}$ respectively. The PropComp procedure has to now compute the properties $\mathcal{P}(L_1)$ using P_{L1}. However, note that the domain $\mathcal{L}$ does not give methods to compute loop-bounds (see Example 7.23[Pg. 138] for a refinement of the domain $\mathcal{L}$ which can compute loop-bounds). Therefore, assuming ulb and llb for the loops are 0 and ∞ respectively, we get the values $\mathcal{P}(L_1) \leq P_{L1} = ((0),(\infty,\frac{10}{4},1))$ and $\mathcal{P}(L_2) \leq P_{L2} = ((0),(\infty,0,1))$.

Proceeding similarly, we get the estimates for $\mathcal{P}(F) \leq P_F = ((5),(\infty,\frac{10}{4},1))$ and $\mathcal{P}(H) \leq P_H = ((0),(\infty,\frac{10}{4},1))$. Intuitively, this corresponds to a counterexample that ends with an infinite loop in L1 that contains the costly operation.

Example 7.22. *Consider the CFG in Figure 7.9[Pg. 133]. In the figure, the state labelled 10 (i.e., s_7) has weight 10 and all other states have weight 0. Note that the graph presented in Figure 7.9[Pg. 133] is not the actual UWTS corresponding to the program, but instead a control flow graph from which the UWTS can be generated. In the analysis that follows, the transition label of the control flow graph (which deal with variables i and j) are not used by the analysis, but are there to help understand the system W. The only information we need from them is the* loop bounds. *Intuitively, the loop bounds imply the the following fact about any trace of the system W. Suppose for 10 consecutive visits to state s_3 in the trace, the transition taken is $s_3 \to s_4$. Then, the next time state s_3 is visited, the transition taken is $s_3 \to s_1$. Similarly, for any 10 consecutive visits to state s_5 with the transition chosen being $s_5 \to s_6$, for the next visit to state s_5 the transition chosen will be $s_5 \to s_7$. This notion of loop bounds of control flow graphs will be formalized in Section 7.6.2[Pg. 136].*

Now, we consider an inductive trace segment cover of the traces of the program:

- $\Pi(W)$ *has the segment cover $\{\{\sigma_{12}\}, Loop_2\}$ where $\sigma_{12} = s_1s_2$, and $Loop_2$ is the SegmentSet containing all maximal segments starting and ending at state s_3 with only the states s_3, s_4, s_5, s_6 and s_7 occurring in it.*
- *$Loop_2$ has the segment cover $\{\{\sigma_{34}\}, \{s_7\}, Loop_1\}$ where $\sigma_{34} = s_3s_4$ and $Loop_1$ contains all maximal segments starting and ending at state s_5 with only the states s_4 and s_5 occurring in it.*
- *$Loop_1$ has the segment cover $\{\{\sigma_{56}\}\}$ where $\sigma_{56} = s_5s_6$.*

Now, to compute the property set $\mathcal{P} = \langle Val^{pba}, maxp, minp, hasInfPath\rangle$ inductively, we have the sequence of inductive steps shown in Figure 7.10[Pg. 133]. We explain the first few steps here:

- *Consider the first inductive step, i.e, inducting from the segment-cover $\{\{\sigma_{56}\}\}$ to the SegmentSet $Loop_1$. Given that $maxp(\{\sigma_{56}\}) = 2$ and $minp(\{\sigma_{56}\}) = 2$, to compute $maxp(Loop_1)$ and $minp(Loop_2)$, we use the loop bounds for the loop $s_5 \to s_6 \to s_5$. Knowing that the loop bound is 10, we get that $maxp(Loop_1) = 1 + maxp(\{\sigma_{56}\}) \cdot 10 = 21$ and $minp(Loop_1) = 1 + minp(\{\sigma_{56}\}) \cdot 10 = 21$. Similarly, by the loop bounds analysis, we get that there is no infinite path in $Loop_1$. For the value $Val^{pba}(Loop_1)$, we use the function f^{pb^*} defined in Section 7.4.3[Pg. 123],*

which gives us the value 5.

- *For the second inductive step, we have to follow the same procedure, but instead use the loop bounds for the loop* $\sigma_{34} \to Loop_1 \to \sigma_7 \to \sigma_{34}$. *The loop bounds analysis will give us that the maxp and minp values are* $1 + 10 \cdot (maxp(\{\sigma_{34}\}) + maxp(Loop_1) + maxp(\{\sigma_7\})) = 241$.

7.6 Quantitative refinements

In this section, we present quantitative refinement algorithms for the state-based and segment-based abstraction schemes.

7.6.1 Algorithmic Refinement of *ExistMax* abstractions

For the state-based abstraction scheme *ExistMax*, we give an algorithm for counterexample-guided abstraction refinement (CEGAR) for quantitative properties. The algorithm is based on the classical CEGAR algorithm [60], which we extend to the quantitative case. Here (as in [60]), we assume that the concrete system is finite-state, and obtain a sound and complete algorithm. In the infinite-state case, the algorithm is sound, but incomplete.

Let $W = (S, \Delta, \rho, s_\iota)$, let f be a $(\leq_p, \sqsubseteq)$-monotonic quantitative property that admits memoryless extremal counterexamples (as stated in Section 7.2[Pg. 114], we restrict ourselves to these properties), and let $\approx_1$ be an equivalence relation on S. Let us further assume that $W^\alpha = (S^\alpha, \Delta^\alpha, \rho^\alpha, s_\iota^\alpha)$ is the result of applying the *ExistMax* abstraction parameterized by $\approx_1$ on W. Let ρ_{ext} be the memoryless counterexample of W^α which realizes the value $f(W^\alpha)$.

Algorithm 4[Pg. 136] is a refinement procedure for *ExistMax* abstractions. Its input consists of the concrete and abstract systems, the equivalence relation that parameterizes the abstraction, the quantitative property, and the extremal trace ρ_{ext}. As the extremal counterexample is memoryless, it is of the shape $H_1 \dots H_k(H_{k+1} \dots H_n)^\omega$. The output of the algorithm is either a concrete counterexample (if one corresponding to ρ_{ext} exists), or a refined equivalence relation (which can be used to produce a new abstract system).

Let us consider a set of traces $\gamma(\rho_{ext})$ of the system W that correspond to an abstract memoryless trace $\rho_{ext} = H_1 \dots H_k(H_{k+1} \dots H_n)^\omega$. The first observation is that checking whether a concrete counterexample exists, i.e., whether $\gamma(\rho_{ext})$ is non-empty, can be done by checking whether a finite abstract trace ρ_u corresponds to a concrete trace. The finite abstract trace ρ_u can be obtained by unwinding the loop part of ρ_{ext} m number of times, where m is the size of the smallest abstract state in the loop part of ρ_{ext}, or formally, $m \leftarrow \min\{|H_i| \mid k+1 \leq i \leq n\}$ (line 1[Pg. 136]). This result can easily be adapted from [60] to the quantitative case.

Lines 3[Pg. 136] to 7[Pg. 136] traverse the finite abstract trace ρ_u, and at each step maintain the set of states that are reachable form the initial state along a path corresponding to ρ_u. (The *post* operator in line 6[Pg. 136] takes as input a set of concrete states L and a weight w, and returns all the successors of states in L that have weight w.)

If the traversal finishes because at i-th step the set R_i is empty, then the algorithm refines the equivalence relation $\approx_1$ by splitting the equivalence class given by G_{i-1} into U and $G_{i-1} \setminus U$ (line 11[Pg. 136]). The set U contains those

Algorithm 4 Refinement for *ExistMax*

Input: UWTS $W = (S, \Delta, \rho, s_\iota)$, quant. prop. f, eq. rel. $\approx_1$,
abstract UWTS $W^\alpha = (S^\alpha, \Delta^\alpha, \rho^\alpha, s_\iota^\alpha)$,
abstract counterexample $\rho_{ext} = H_1 \dots H_k(H_{k+1} \dots H_n)^\omega$
Output: refined eq. rel. $\approx_2$
or a concrete counterex. *tecx*

1: $m \leftarrow \min\{|\gamma(H_i)| \mid k+1 \leq i \leq n\}$
2: $\rho_u \leftarrow \text{unwind}(\rho_{ext}, m, k, n)$
   $\{$we have $\rho_u = G_1 \dots G_{k+(n-k)\cdot m}\}$
3: $R_1 \leftarrow \gamma(\sigma_1) \cap \{s_\iota\}$
4: $i \leftarrow 1$
5: **while** $R_i \neq \emptyset \wedge i < (k + (n-k) \cdot m)$ **do**
6: $\quad R_{i+1} \leftarrow post(R_i, \rho^\alpha(G_{i+1})) \cap \gamma(G_{i+1})$
7: $\quad i \leftarrow i + 1$
8: **if** $R_{i-1} \neq \emptyset$ **then**
9: $\quad$ **return** counterEx$(R_0, \dots, R_{i-1})$
10: **else**
11: $\quad U \leftarrow \{s \in \gamma(R_{i-1}) \mid \exists s' \in \gamma(G_i) : \Delta_W(s, s') \wedge \rho(s') = \rho^\alpha(G_i)\}$
12: $\quad$ **return** refine$(\approx_1, G_{i-1}, U)$

states that have a transition (corresponding to a transition in ρ_u) to a state with weight $\rho^\alpha(G_i)$. The intersection $U \cap R_{i-1}$ is empty, because R_i is empty. Thus separating U leads to eliminating the counterexample from the abstract system.

If the traversal finishes a pass over the whole trace ρ_u, it can construct a concrete counterexample using sets of states $R_0, \dots, R_{i-1}$ (line 9[Pg. 136]).

We have thus extended the classical CEGAR algorithm [60] to the quantitative case. The extension is simple, the main difference is in taking into account the weights in lines 6[Pg. 136] and 11[Pg. 136].

7.6.2 Algorithmic Refinement of *HPathBound* abstractions

In this subsection, we describe an algorithmic technique for refinement of segment-based abstractions. We assume that the abstract bound pair domain is precisely inductive.

We assume that the extremal counter-example from the evaluation of a *HPathBound* abstraction is returned as a abstract hierarchical trace $\psi = \langle(\phi_0 \wedge P_0)(\phi_1 \wedge P_1) \dots (\phi_k \wedge P_k)((\phi_{k+1} \wedge P_{k+1}) \dots (\phi_n \wedge P_n))^\omega, sub_\psi\rangle$. Note that we can assume a lasso-shaped counter-example due to the memoryless property of the quantitative properties we consider. Furthermore, without loss of generality we also assume that every leaf in the abstract trace segment cover is composed of segments of length 1.

The basic structure of the refinement algorithm is same as in *ExistMax*. However, the main difference is in the *post* operator. For hierarchical traces, we define a non-deterministic *post* operator in Algorithm 5[Pg. 137]. Intuitively, the algorithm non-deterministically chooses a level of the hierarchical trace to perform the analysis. First, given a hierarchical trace of length 1, *post* operator computes (Lines 3[Pg. 137] and 4[Pg. 137]) the set $\gamma(\phi_0 \wedge P_0) \circ \Delta = \{s\sigma \mid \sigma \in$

$\gamma(\phi_0 \wedge P_0) \wedge (s, \mathit{first}(\sigma)) \in \Delta\}$. Then, it computes the top-level post set R^* of states reachable from R using the segments from $\gamma(\phi_0 \wedge P_0) \circ \Delta$. Now, non-deterministically (Line 5[Pg. 137]) it chooses whether to descend into the next level of the hierarchy. If it decides to, the set of post states $R^\dagger$ is computed from the levels below, and then the strengthening of R^* by $R^\dagger$, i.e., $R^* \cap R^\dagger$ is returned.

Assume that the *post* computation is done at a particular level, i.e., the level below is not used. Intuitively, this means that all the segments in $\phi_0 \wedge P_0$ are assumed to be valid segments, and the property bounds are assumed to be tight, i.e., the part of the counter-example corresponding to $\phi_0 \wedge P_0$ is considered non-spurious. Note that in the case where the algorithm always descends to the lowest level, the set returned is exactly the set of states reachable using segments in $\gamma(\psi) \circ \Delta$. We also remark that nondeterminism in Algorithm 5[Pg. 137] can be instantiated in a manner suitable for a particular domain.

Algorithm 5 Counterexample analysis for *HPathBound*

Input: Hierarchical trace $\psi = \langle(\phi_0 \wedge P_0)\dots(\phi_k \wedge P_k), sub_\psi\rangle$,
Concrete set of states R,
Output: Over-approximation of states reachable through segments in $\gamma(\psi)$.

1: **if** $n > 1$ **then**
2:    **return** $post((\phi_1 \wedge P_1)\dots(\phi_n \wedge P_n), sub_\psi\rangle, post(\langle\phi_0 \wedge P_0, sub_\psi\rangle, R))$
3: $T \leftarrow \gamma(\phi_0 \wedge P_0) \circ \Delta$
4: $R^* \leftarrow \{s' \mid \exists s\sigma s' \in T : s \in R\}$
5: **if** $*$ **then**
6:    **if** $sub_\psi(0) \neq \bot$ **then**
7:       $R^\dagger \leftarrow post(sub_\psi(0), R)$
8: **else**
9:    $R^\dagger \leftarrow R^*$
10: **return** $(R^* \cap R^\dagger, R^*, R^\dagger)$

Let $\mathcal{C}$ be the inductive trace segment cover and let $\psi = \langle(\phi_0 \wedge P_0)(\phi_1 \wedge P_1)\dots(\phi_k \wedge P_k)((\phi_{k+1} \wedge P_{k+1})\dots(\phi_n \wedge P_n))^\omega, sub_\psi\rangle$ be the extremal abstract trace. The abstraction refinement procedure $hAbsRefine(\mathcal{C}, \psi, R_0)$ proceeds similarly to Algorithm 4[Pg. 136] as follows:

- The abstract hierarchical trace ψ is unwound m number of times, where $m = \min\{|\gamma(\phi_i \wedge P_i)| \mid i \in \{k+1, \dots, n\}\}$;
- Let R_0 be the set of concrete initial states. For each abstract SegmentSet property pair $\phi_i \wedge P_i$ in the unwound trace, we compute $(R_{i+1}, R^*, R^\dagger) = post(\langle\phi_i \wedge P_i, sub_\psi\rangle, R_i)$;
- If at any step $R_{i+1} = \emptyset$, we refine the inductive trace segment cover $\mathcal{C}$ using set R_i and $\langle\phi_i, sub_\rho\rangle$ as $hRefine(R_i, \langle\phi_i, sub_\rho\rangle)$ (explained below). Otherwise, return any concrete counter-example constructed from the set $\{R_0, R_1, \dots\}$.

We describe the computation of $hRefine(R_i, \langle\phi_i \wedge P_i, sub_\rho\rangle)$ when $post(\langle\phi_i, sub_\rho\rangle, R_i) = \emptyset$: during the computation $post(\langle\phi_i, sub_\rho\rangle, R_i)$ (execution of Algorithm 5[Pg. 137]) we have at least one of the following cases based on the values of R^* and $R^\dagger$. First, we define $T_R = \{\sigma \mid \exists s \in R(r, \mathit{first}(\sigma)) \in \Delta \wedge \sigma \in \gamma(\phi_i \wedge P_i)\}$ and $T_R^{below} = \{\sigma \mid \exists r \in R.(r, \mathit{first}(\sigma)) \in \Delta \wedge \sigma \in \gamma(sub_\psi(i))\}$. Intuitively, T_R is the set of concrete segments from R in $\phi_i \wedge P_i$, and T_R^{below} is

the set of concrete segments from R generated from the hierarchical levels under $\phi_i \wedge P_i$. We have one of the following options:

- $R^* = \emptyset \wedge R^\dagger = \emptyset$. In this case, the refinement returned is $hAbsRefine(\mathcal{C}, \langle sub_\rho(i), sub_\rho \rangle, R)$, i.e., we run the abstraction refinement procedure on the lower level, starting from the concrete set of states R.
- $R^* = \emptyset$. In this case, we perform a horizontal refinement to separate the sets T_R and $T \setminus T_R$, i.e., the node labelled $\phi_i \wedge P_i$ is split into $\phi^A \wedge P^A$ and $\phi^B \wedge P^B$ where $\gamma(\phi^A \wedge P^A) \cup \gamma(\phi^B \wedge P^B) = \gamma(\phi_i \wedge P_i)$ and $T_R \subseteq \gamma(\phi^A \wedge P^A) \wedge T_R \cap \gamma(\phi^B \wedge P_B) = \emptyset$. Intuitively, we are separating segments in $\phi_i \wedge P_i$ that are from R from those that are not from R.
- $R^* \neq \emptyset \wedge R^* \cap R^\dagger = \emptyset$. In this case, we perform multiple simultaneous refinements. The segment sets which need to be distinguished from each other are $T \setminus (T_R \cup T_R^{below})$, T_R and T_R^{below}. Intuitively, we are trying to separate the segments in $\gamma(\phi_i \wedge P_i)$ that (a) do not start from R (i.e., $T \setminus (T_R \cup T_R^{below})$), (b) those that start from R and are validly generated from the levels below (i.e., T_R^{below}); and (c) those that do start from R and are not validly generated from the levels below (i.e., T_R). Formally, let $\phi^A \wedge P^A$, $\phi^B \wedge P^B$, and $\phi^C \wedge P^C$ be such that:
  - $T \subseteq \gamma(\phi^A \wedge P^A) \cup \gamma(\phi^B \wedge P^B) \cup \gamma(\phi^C \wedge P^C)$;
  - $T_R \subseteq \gamma(\phi^A \wedge P^A) \wedge T_R^{below} \cap \gamma(\phi^A \wedge P^A) = \emptyset$;
  - $T_R^{below} \subseteq \gamma(\phi^B \wedge P^B) \wedge T_R \cap \gamma(\phi^B \wedge P^B) = \emptyset$; and
  - $(T_R \cup T_R^{below}) \cap \gamma(\phi^C \wedge P^C) = \emptyset$.

  First, we do a horizontal refinement splitting the node $\phi_i \wedge P_i$ into $\phi^A \wedge P^A$, $\phi^B \wedge P^B$, and $\phi^C \wedge P^C$. Second, in the subtree rooted at $\phi^A \wedge P^A$, the levels below contains the information that T_R is infeasible, but the root does not. So, we perform upward strengthening refinements till the root contains the same information. Third, in the subtree rooted at $\phi^B \wedge P^B$, the root contains the information that T_R^{below} is infeasible, but the levels below do not. So, we perform either (a) downward strengthening refinements till the levels below contain the same information; or (b) vertical joining refinements till there are no levels below. Note that if one of $\phi^A \wedge P^A$, $\phi^B \wedge P^B$, or $\phi^B \wedge P^B$ is empty, we omit it.

Example 7.23. *Consider the HCFG H and the corresponding abstract trace segment cover $\mathcal{C}$ from Example 7.21[Pg. 132]. We now show some examples of hierarchical counter-example guided refinements for computing the limit-average value.*

We work in a more powerful refined domain than in Example 7.21[Pg. 132], one that allows computation of loop bounds. Let $\mathcal{L}$ be the domain of regular expressions over HCFG's along with a relation between the values of the variables in the initial and final states. For example, the expression $((s_4 s_5 (s_6 + s_7) s_3, b' = \neg b)$ represents the set of segments which match $s_4 s_5 (s_6 + s_7) s_3$ and have the value of b is the last state is negation of the value of b in the first state.

Let us first start with abstract trace segment cover $\mathcal{C}$ from 7.21[Pg. 132]. A part of the abstract extremal trace generated from $\mathcal{C}$ will be $\langle L_1, sub \rangle$ where (a) $sub(1) = \langle (C_{L1})^\omega, sub' \rangle$, where (b) $sub'(i) = s_4 s_5 s_7 s_3$ for all $i \geq 1$. We will illustrate two refinement steps that might occur:

- *Suppose during the refinement process we are computing $post(\langle L_1, sub' \rangle, R)$ where R is the set of states at location s_3 with $j = 0$. Now, we can perform the analysis either at the top level or at the*

lower level:

  - *At the top level, we get the post states to be* R^* *where the control location of a state is in* s_3*.*
  - *At the lower level, we get the post states to be* $R^\dagger = \emptyset$ *as the transition from* s_3 *to* s_4 *is disabled due to* j *being* 0*.*

  Therefore, we need to refine the abstract SegmentSet $s_4 s_5 (s_6 + s_7) s_3$*. One possible valid refinement is to strengthen the set* $s_4 s_5 (s_6 + s_7) s_3$ *to* $s_4 s_5 (s_6 + s_7) s_3, j > 0 \wedge j' = j - 1$*. Using this strengthened set, we can compute that the upper and lower loop bounds for* L_1 *are* 10*. This leads to a improvement in the value of the system as now, there is no infinite path in the high value segment* L_1*. The new value of the system is* $\frac{20}{9}$*.*

- *Suppose during the refinement process we are computing* $post(\langle L_1 \wedge P, sub' \rangle, R)$ *where* P *bounds the limit-average of segments in* L_1 *to* $\frac{10}{4}$*,* R *is the set of states at location* s_3 *with* $b = true$*. Again, performing the analysis at top level produces* $R^* = \{s_7\}$*, but the lower level produces* $R^\dagger$ *where* $R^\dagger = \emptyset$*. Therefore, we can to refine the abstract SegmentSet* $s_4 s_5 (s_6 + s_7) s_3$ *and one possible refinement is a horizontal split into* $s_4 s_5 s_6 s_3, b = true \wedge b' = false$ *and* $s_4 s_5 s_7 s_3, b = false \wedge b' = true$*. Performing this refinement reduces the value of* L_1 *to* $\frac{20}{9}$ *and hence, by upward strengthening the value of the whole system to* $\frac{20}{9}$*.*

7.7 Case study: Cache hit-rate analysis

We present a case study to demonstrate anytime verification, and to evaluate *ExistMax* and hierarchical *PathBound* abstractions. Worst-case execution time (WCET) estimation is an important part of verifying real-time systems. We only study one aspect, i.e., cache behavior prediction. In a processor, clock cycles needed for an instruction can vary by factors of 10-100 based on cache hits vs misses. Assuming the worst-case for all instructions leads to bad WCET estimates. Abstract domains for cache behavior prediction are well studied (e.g., [84, 170]). However, we know of no work on automated refinement for these abstractions. Note that this case study and the accompanying implementation is not a complete WCET analysis tool, but a prototype used to illustrate the quantitative abstraction refinement approach. The prototype returns the average execution cost of a single instruction. Our intention is just to evaluate the anytime aspect of our approach.

We estimate the average instruction cost (and hence, the cache-hit rate) using the limit-average property. Intuitively, we put the whole program in a nonterminating loop. The limit-average then corresponds to the average cost of an instruction in the worst-case execution of a loop. (For a terminating program, it is the execution of the artificial outer loop.) We report limit-average values of the average instruction cost instead of the cache hit-rate.

Cache behavior and cache abstractions. We first present the structure of a simplified cache model we will be assuming. The cache model and the abstract domain used are from [84]. The cache consists of multiple *cache sets*, each with A *cache-lines* (usually, $A \leq 8$). The memory is partitioned into *blocks* with M being the set of blocks. Each block is cached in a unique cache set and occupies one cache-line. The replacement policy is LRU, i.e, least recently

Example	Step	Value	Time (ms)	Tracked
Basic Example	0	14.14	1240	
	1	6.50	2102	i
	2	4.87	2675	a
	3	4.75	3275	b
	4	1.27	3864	c
	5	1.03	4631	v
Binary search	0	15.77	908	
	1	11.15	1130	m
	2	8.23	1369	r
	3	5.0	1707	l
	4	3.76	1895	s
	5	3.0	2211	a[$\frac{(N-1)}{2}$]
	6	2.97	2527	a[$\frac{(N-3)}{4}$]
	7	2.85	3071	a[$\frac{(3N-1)}{4}$]
Polynomial Eval.	0	15.49	524	
	1	8.13	759	i
	2	4.45	1025	val
	3	2.95	1237	x
GCD	0	13.76	289	
	1	9.47	399	inp2
	2	6.65	472	inp1
	3	6.33	536	temp

Table 7.1: *ExistMax* abstraction results

used blocks from a cache-set are evicted first. We model a cache-line as a map $age^{-1} : \{1, \dots, A\} \rightarrow M$, with i mapping to i^{th} most recent block.

The *Must cache abstraction* from [84] can be used to compute invariants about cache contents. Any memory block present in a must-invariant at a program location must necessarily be present in the cache. The abstract must cache is a collection of the abstract must cache-sets. An abstract cache-set is a map $age_A^{-1} : \{1, \dots, A\} \rightarrow 2^M$; and it contains a concrete cache-set (age^{-1}) iff $\forall m : m \in age^{-1}(b) \implies \exists a \leq b : age^{-1}(a) = m$. Intuitively, $m \in age^{-1}(b)$ bounds m's age in the concrete cache-set. The join is computed by taking the maximum age for each block.

Multiple abstract domains are possible based on which cache sets are tracked. If no set is tracked, every memory access is deemed a cache-miss; whereas the invariant computation is very expensive in the domain which tracks all cache-sets. Here, we start from the empty cache and refine by tracking more cache-sets as necessary. For example, in the run of our tool on the binary search example from Figure 7.11, the initial abstract cache domain does not track any cache sets. The refined cache domain after one step tracks the cache set corresponding to the memory location of the variable m. Using this refined domain, we will find that most accesses to variable m are cache-hits.

7.7.1 Implementation details

We implemented a WCET analyzer based on the presented techniques in a tool QUART that analyzes x86 binaries.

Static analysis. We analyze the binary and produce the control flow graph. Instructions in the program may operate on non-constant addresses (for example, array indexing leads to variable offsets from a fixed address). However, if the exact addresses cannot be computed statically, we perform no further analysis and assume that the memory access is a cache miss. This restriction comes from the cache abstract domain we use [84].

Instruction cost computation. In the resulting graph, we annotate states with invariants from the current cache abstract domain. From the invariants, we compute costs of each transition (we use costs of 1 and 25 cycles for cache-hits and cache-misses, respectively). We then find the worst-case using techniques of Section 7.3[Pg. 115] and Section 7.4.3[Pg. 123] to find a the worst-case limit-average value. Furthermore, we implemented the extension of the algorithm to graphs with both edge weights and edge lengths [61].

Refinement. We analyze worst-case counter-example *ext*:
Feasibility analysis. We first check if *ext* is a valid program execution (ignoring the cache) using a custom symbolic execution computation. If *ext* is not a valid execution, we refine the abstract graph using standard CEGAR techniques.
Cache Analysis. If *ext* is valid, we compute the concrete cache states for it. If the concrete value obtained is the same as that of *ext*, we return the concrete trace.
Refinement heuristic. Otherwise, of all locations accessed in the loop of *ext*, we find the one with most abstract cache misses which are concrete cache hits. The current cache abstract domain is refined by additionally tracking this location.
Fall-back refinement. If all the locations accessed in *ext* are already being tracked, we use Algorithm 4[Pg. 136] and the algorithm given by *hAbsRefine* to do the refinement.

7.7.2 Evaluation of *ExistMax* abstraction

For evaluating the *ExistMax* abstraction and refinement methods, we consider binaries for five (small) C programs, including the example from the introduction (called Basic example in the table). The results are in Table 7.1[Pg. 140]. For each example program, the table contains lines, with each corresponding to a refinement step. For each refinement step we report the current estimate for the limit-average value, the running time for the analysis (cumulative; in milliseconds) and in case the refinement enlarged the abstract cache, we also show what new memory locations correspond to the entries in the abstract cache. In each case, the over-approximated limit-average value decreases monotonically as the tool is given more time.

Binary search. We analyze a procedure (Figure 7.11[Pg. 142]) that repeatedly performs binary search for different inputs on a given array. We start with the empty abstract cache domain and all behaviors have high values (with

```
while(true)
 input(s);
 l = 0; r = N - 1;
 do {
  m = l + r / 2;
  if(s > a[m])
   l = m + 1;
  else
   r = m - 1;
 } while(l <= r ∧ a[m] != s)
```

Figure 7.11: Binary Search

worst-case value 15.77). In the *ext*-trace, variable `m`, accessed 4 times every iteration of the inner loop, causes most spurious cache misses.

Using the *Refinement heuristic* we heuristically choose the location of `m` is additionally tracked in the cache abstract domain reducing the value to 11.15. Indices `l`, `r` and the input `s` are the next most frequently used, and are added subsequently to the cache abstract domain. More importantly, the most used array elements are added in order. During binary search, the element at position $N/2$ is accessed always, and the elements at $N/4$ and $3N/4$ half the times, and so on. The refinements obtained add these array elements in order. This illustrates the anytime aspect: refinement can be stopped at any point to obtain an over-approximation of the value.

7.7.3 Evaluation of the hierarchical *PathBound* abstraction

For evaluating the *PathBound* abstraction refinement procedure, we picked 4 benchmarks from the collection of WCET benchmarks in [91]. These benchmarks were larger than the ones for the *ExistMax* evaluation with around 150-400 lines of code each. The benchmarks we picked included a simple program which scanned a matrix and counted elements, matrix multiplication, and two versions of discrete-cosine transformations.

We used the hierarchical *PathBound* abstraction-refinement algorithm, i.e., the algorithm given by *hRefine*. We note that we do not perform any cache refinements. Nevertheless, the hierarchical aspect of hierarchical *PathBound* abstraction was evaluated, as three of the benchmarks contained a number of nested loops. The challenge addressed was to obtain good (and monotonically decreasing) estimates on WCET, as the abstraction is refined.

We summarize the results in Table 7.2[Pg. 143]. For each example program, the table contains a number of lines, with each line corresponding to a refinement step. For each refinement step we show the current estimate for the limit-average value, and the running time for the analysis (cumulative; in milliseconds). As it can be seen, the limit-average values monotonically decrease with longer execution time. It should be noted that for most of these programs, to obtain similar values with the *ExistMax* approach, one would need to perform a large number (in thousands) of counter-example guided refinements (as the nested loops would have to be unrolled).

Bench mark	Step	Value	Time (ms)
cnt	0	8.74	1810
	3	8.64	6349
	4	4.08	8298
matmult	0	8.73	4669
	2	8.71	15660
	5	8.71	30408
	6	4.17	35676
fdct	0	6.88	2142
	1	1.94	4274
	2	1.76	6685
jfdctint	0	6.95	3095
	1	3.35	5759
	2	1.89	8674
	3	1.57	11809

Table 7.2: *PathBound* abstraction results

7.8 Summary

This chapter makes four main contributions. First, we present a general framework for abstraction and refinement with respect to quantitative system properties. Refinements for quantitative abstractions have not been studied before.Second, we propose both state-based and segment-based quantitative abstraction schemes. Quantitative segment-based abstractions are entirely novel, to the best of our knowledge. Third, we present algorithms for the automated refinement of quantitative abstractions, achieving the monotonic over-approximation property that enables anytime verification. Fourth, we implement refinement algorithms for WCET analysis of executables, in order to demonstrate the anytime verification property of our analysis, and to investigate trade-offs between the proposed abstractions.

Chapter 8

Precision Refinement for Worst-Case Execution Time Analysis

The quantitative analysis of program performance has become an important goal for formal methods. The aim of this chapter is to obtain parametric estimates for the worst-case execution time (WCET) of programs. Our estimates are given as symbolic expressions defined in terms of program parameters such as loop bounds, array sizes, and configuration modes.

There are two novel technical contributions. First, for modeling and estimating the WCET of programs, we define hierarchical parametric graphs (HPGs) as a hierarchical form of weighted graphs whose weights are given by symbolic expressions. We then present an algorithm for computing the maximum-weight length-constrained path in an HPG. Second, we provide an automatic method for the construction and successive refinement of HPGs, using segment-based program abstraction and counterexample-guided predicate refinement introduced in the previous chapter (Chapter 7[Pg. 110]). This method can be used for obtaining increasingly tighter parametric WCET estimates.

We experimentally evaluate our approach on a set of examples from WCET benchmark suites and open-source linear-algebra packages. Most state-of-the-art WCET tools provide results for concrete numeric values of parameters. We show that our analysis, with comparable effort, provides parametric estimates, which, when instantiated to numeric values, in some cases significantly improve WCET estimates computed by existing tools.

8.1 Motivation

Worst-case execution time (WCET) analysis [171] is important in many classes of applications, ranging from the verification of safety-critical real-time embedded systems to the optimization of heavily-used methods in computer algebra packages. First, real-time embedded systems have to react within a fixed amount of time, so verifying that the response in the worst-case takes less time than the imposed limit is critical. Second, computer algebra libraries often provide dif-

ferent implementations for the most heavily-used methods. Users then have to choose the most suitable method for their particular system architecture. In both cases, a tool that soundly and tightly approximates the WCET of programs would thus be very helpful.

Most state-of-the-art WCET approaches derive a single number estimating the WCET of a program [171]. In many cases, this is however a pessimistic over-estimation. For instance, for a program that transposes a two-dimensional $n \times n$ matrix, estimating its WCET by a single numeric value requires giving the WCET for the largest possible value of n. On the other hand, a *parametric* WCET estimate, that is, a safe estimate given as a symbolic expression in terms of the program parameters (such as matrix dimensions), is more useful in many applications. For instance, a user of a computer algebra package might not know the matrix dimensions in advance, but still needs to choose an implementation suitable for her system. Note that what is required is not the asymptotic complexity, or simply the bounds on the number of loop iterations, but a parametric WCET estimate for a particular architecture.

Here, we address the challenge of *computing parametric WCET estimates.* To this end, we develop new program analysis techniques. First, we present an algorithm to maintain a representation of a set of paths deemed feasible as the abstraction of the program gets refined. The set of paths is represented by hierarchical parametric graphs (HPGs). The algorithm is based on segment-based abstract interpretation, and counterexample-guided refinement with interpolation. Second, we propose an algorithm for estimating the WCET of the set of paths represented by an HPG. This step uses a low-level analyzer to estimate the WCET of basic blocks of a program, and a novel algorithm to estimate the WCET of all paths represented by an HPG. The refinement then provides us with increasingly better estimates of WCET.

HPGs. *Hierarchical parametric graphs* (HPGs) are a hierarchical form of weighted graphs with parametric weights on nodes. HPGs are hierarchic as they are a result of an abstraction that preserves the structure of programs. HPGs are parametric as the weights of the nodes are symbolic expressions representing WCET estimates of sub-programs. We show that in order to find the WCET, we need to solve a maximum-weight length-constrained problem. This problem is a hierarchical and parametric variant of a previously-studied problem [156]. Solving the maximum-weight length-constrained problem gives us a parametric WCET for the current HPG.

Abstraction. The initial HPG is computed using the hierarchical segment-based abstraction of [67] for quantitative properties [38]. Worst-case execution time is not a property of a state, but rather a property of a *segment* (i.e. a sequence of instructions). Therefore, segment-based abstraction, where an abstract states corresponds to a set of segments, is more suitable for parametric WCET calculation than standard state-based abstractions, where an abstract state corresponds to a set of concrete states. The abstraction is hierarchical in order to capture the hierarchical nature of traces of structured programs (with nested loops and procedures). For example, we split the set of traces through a nested loop into repeated iterations of the outer loop, where each of these iterations is split into repeated iterations of the inner loop. The hierarchical abstraction is represented by a tree, called the *abstract segment tree.* Each node of the abstract segment tree is an abstract state representing a set of segments. The children of a node can be combined to produce every segment in the parent

node, as in the nested loop case. We show how to construct an HPG from an abstract segment tree. The weights of nodes of the HPG are obtained by over-approximating the maximal WCET for segments represented by each abstract state.

Refinement. After solving the maximum-weight length-constrained problem of the current HPG representation of feasible paths, we obtain a witness trace for the program. A *witness trace* is a path through the state space of the program that exhibits the current estimate of the WCET (for concrete values of parameters), and is feasible under the current abstraction. We next use the classical abstraction-refinement loop approach, adapted however to HPGs, as follows. We check whether the witness trace is feasible for the original program, i.e., check if the abstract trace also exists in the concrete program. If so, the current parametric WCET is also feasible, we report it and terminate the analysis. If the witness trace is not feasible in the concrete program, we refine the abstraction. For doing so, we develop a novel approach for refinement of hierarchical segment-based abstractions based on interpolation. However, our techniques differ significantly from interpolation for counter-example trace analysis for standard abstract trace analysis (see, for example, [105]) as our abstractions are segment-based rather than state-based. Intuitively, for state-based abstractions, interpolants are used to summarize information about the state obtained after executing a prefix of the trace, while in our case, interpolants are used to summarize information about the relation between the initial and final states of a segment of the trace.

We rely on the method of [100] for computing small interpolants and propose a new, heuristic-driven method to choose the best interpolant for our abstraction refinement. Intuitively, a good interpolant for our method is an interpolant which does not depend on concrete values of program parameters and is small in the number of its components. The abstraction refinements are monotonic with respect to WCET estimates: the estimates are monotonically decreasing. We are not aware of any other approach that combines segment-based abstraction with interpolation for quantitative program analysis, in particular for WCET computation.

Experimental Evaluation. We built a new software tool IBART, for parametric WCET estimations. We used the CalcWcet167 tool [112] to obtain the WCET estimates of basic blocks of a program on an Infineon C167 processor. Our implementation takes C programs as input, and provides parametric WCET estimates as output. We evaluated IBART on challenging examples from WCET benchmark suites and open-source linear algebra packages. All examples were solved under 60 seconds. Most state-of-the-art tools report a single numeric value as a WCET estimate; the one exception we know of is SWEET [29], which we describe in the related work section. Our tool provides a much more informative result, that is, a parametric WCET estimate. The quality of our results can be compared to other tools only when one chooses concrete numeric values of parameters for the other tools. We show that in this case, IBART provides significantly better WCET estimates.

Summary. Our two main technical contributions are: (a) the hierarchical parametric graphs (HPGs) and the reduction of parametric WCET computation to a maximum-weight length-constraint optimization problem over paths of HPGs; and (b) a novel interpolation-based approach for refining segment-based abstractions. This is used to refine the parametric WCET of the program.

8.2 Illustrative Examples

This section illustrates our approach to parametric WCET computation. We start with a simple example (Example 8.1[Pg. 147]) to present the main steps of our method: segment-based abstraction, estimation of WCET using HPGs, and counterexample-guided abstraction refinement with interpolation. We next show a more complicated example (Example 8.2[Pg. 150]) motivating the need of solving maximum-weight length-constraints in HPGs.

Example 8.1. *Consider the program from Figure 8.1[Pg. 148], which will be our running example through the chapter. Program blocks* `op1()`*,* `op2()`*, and* `op3()` *are operations whose executions take 10, 1, and 50 time units, respectively (these costs are derived from a low-level timing analysis tool). In this example, we assume that program conditionals and simple assignments take* 1 *time unit.*

It is not hard to see that for small values of the loop bound n*, the WCET path of this program visits the outermost* `else` *branch containing* `op3()` *– when* `n` *is small, the execution cost of* `op3()` *dominates the cost of the loop. However, for larger values of* `n`*, the WCET path visits the* `then` *branch of the outermost* `if` *and the* `for`*-loop. The WCET of this example thus depends on n. Our approach discovers this fact, and infers the WCET of the program as a function of n as follows: if $n \leq 5$ then* 51 *else $3+4n+9\lfloor n/2 \rfloor$. The WCET computed by our method is hence parametric in n. The computation proceeds as follows.*
Control-flow graph. *We construct the control-flow graph (CFG) of the program in Figure 8.1[Pg. 148]. First, we rewrite the program in a* `while`*-loop language with* `assume` *statements — see the right column of Figure 8.1[Pg. 148]. We have named the assumptions and the transition relations (i.e. transition predicates) of instructions. For example, (φ_1) denotes the assumption $a < b$; and (φ_8) represents the transition predicate $i' = i+1$ of the assignment $i := i+1$ (here, i' refers to the value of i after the assignment). The* `while`*-language representation provides a way to cleanly map statements from the program to its CFG given in Figure 8.2[Pg. 148]. Nodes in the CFG, written as C_k, denote locations in the program control flow and edges are annotated by formulae φ_k over program variables (and their primed versions), describing the control flow.*
Hierarchical segment-base abstraction. *We next apply segment-based abstraction on the CFG of Figure 8.2[Pg. 148]. The initial abstraction is given by the abstract segment tree (Figure 8.3[Pg. 149]). The tree structure comes from the hierarchical nature of the CFG. Nodes of the tree (denoted by A_k) represent a set of execution segments, i.e., a sequence of CFG nodes. Each node stores a* shape predicate *(denoted Shape) describing the paths of the segments through the CFG, and a* transition predicate *(denoted Trans) characterizing the transition relation of the set of segments. A shape predicate is an extended regular expression over either the children of the node, or over the CFG nodes. It is an extended regular expression, as it may contain symbolic exponents obtained, for example, from loop bounds. The transition predicate is a formula over the values of program variables at the beginning and at the end of segments. Note that in the abstractions defined in this chapter, the shape is stored as a transition system rather than a regular expression and the nodes store more detailed information. Here we opted for a regular expression for better readability.*

We describe node A_2 in more detail – the other nodes are constructed similarly. The construction of A_2 in the initial abstraction is done syntactically.

```
if (a<b)
   for (i:=0;i<n;i++)
      if (i<⌊n/2⌋)
         op1()   cost=10
      else
         op2()   cost=1
else
   op3()   cost=50
```

```
if (*)
    assume a<b;  (φ1)
    i:=0;  (φ2)
    while (*)
        assume (i<n);  (φ3)
        if (*)
            assume i<⌊n/2⌋;  (φ4)
            op1();  (φ5), cost=10
        else
            assume i≥⌊n/2⌋;  (φ6)
            op2();  (φ7), cost=1
        i:=i+1;  (φ8)
    assume (i≥n);  (φ9)
else
    assume a≥ b;  (φ10)
    op3();  (φ11), cost=50
```

Figure 8.1: Example 8.1[Pg. 147] (above); written in a while-language (right).

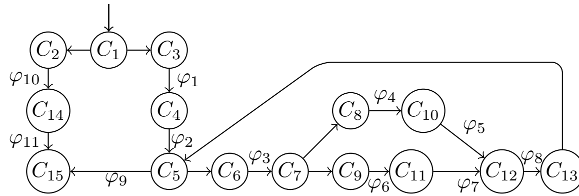


Figure 8.2: CFG of Example 8.1[Pg. 147].

Node A_2 represents all segments through the CFG nodes corresponding to the then-branch of the outermost `if`*. It is split into three sets of segments: (a) node A_3 denoting the set of segments before the loop, i.e., that is the set of segments through the sequence of nodes $C_1C_3C_4C_5$; (b) node A_4 denoting the set of segments given by the loop of the CFG; and (c) node A_5 representing the set of segments after the loop of the CFG. Take n as the bound on the number of loop iterations in the CFG. For building A_4 we use node A_6 describing all segments in one iteration of the loop in the CFG. The segments in A_6 can be concatenated to cover all segments in A_4. For computing loop bounds, we use [113]. The loop bound n is then noted in the shape predicate of A_4.*

WCET estimate using HPGs. *For each node in Figure 8.3[Pg. 149], we next calculate the cost (i.e., the weight) of the segments represented by this node. The weight of the abstract node represents its WCET. As each node is defined in terms of its children, we traverse the tree bottom-up. The root of the tree contains a WCET estimate of the complete set of segments, and hence of the program.*

To estimate the WCET of a node, we construct a hierarchical parametric graph (HPG) for each node of Figure 8.3[Pg. 149]. A node in an HPG can also be an HPG (however, a HPG cannot be recursive; for example, an HPG cannot be its own node). The construction of an HPG from an abstract segment tree node is straightforward. We use the shape predicate of the node to construct the

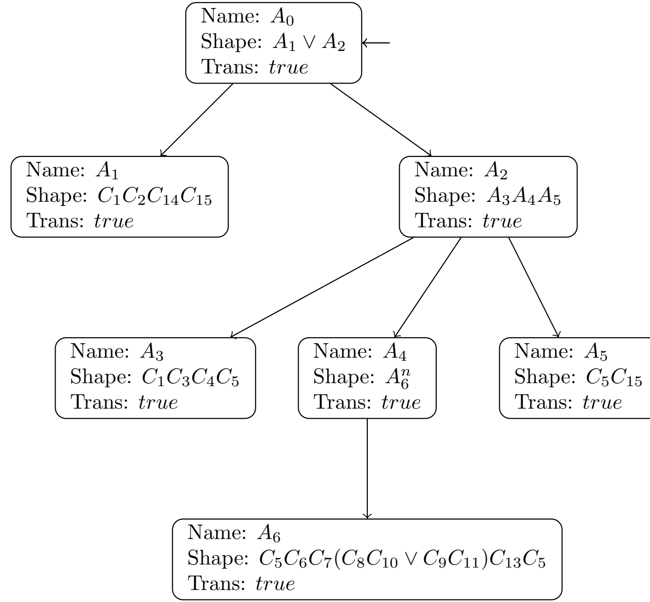


Figure 8.3: Initial abstraction for Example 8.1[Pg. 147].

HPG, and use the WCET of the children nodes to estimate the cost of the node. For example, for node A_2 we construct a graph with three nodes (corresponding to HPGs for A_3, A_4, A_5), with directed edges from A_3 to A_4 and from A_4 to A_5. For node A_4, we obtain a graph with one node (A_6) that can repeat at most n times. The costs of the abstract segment tree nodes are calculated as:

- $cost(A_6) = cost(\varphi_3) + \max(cost(\varphi_4) + cost(\varphi_5), cost(\varphi_6 + cost(\varphi_7)) + cost(\varphi_8) = 13$
- $cost(A_4) = n \cdot A_6 = 13n$
- $cost(A_3) = cost(\varphi_1) + cost(\varphi_2)$
- $cost(A_5) = cost(\varphi_9)$
- $cost(A_2) = cost(A_3) + cost(A_4) + cost(A_5) = 3 + 13n$
- $cost(A_1) = cost(\varphi_{10}) + cost(\varphi_{11}) = 51$
- $cost(A_0) = \max(cost(A_1), cost(A_2)) = \max(51, 3 + 13n)$
  $= \textit{if } n \leq 3 \textit{ then } 51 \textit{ else } 3 + 13n$

The WCET estimate of our running example is given by $cost(A_0)$, and depends on the value of n, i.e., when $0 \leq n \leq 3$ the WCET is different than in the case when $n > 3$. To ensure that the computed WCET estimate is precise, we need to ensure that our abstraction did not use an infeasible program path to derive the current WCET estimate. We therefore pick a concrete value of n for each part of the WCET estimate, and check whether the corresponding witness worst-case path is feasible. If it is, the derived WCET estimate is actually reached by the program and we are done. Otherwise, we need to refine our abstract segment tree. In our example, we thus have the following two cases:

- *Case 1: $n \leq 3$. We pick $n = 1$. The WCET estimate of A_0 is then 51. Here, the witness trace is $C_1C_2C_{14}C_{15}$. This trace is a feasible trace of Figure 8.1[Pg. 148].*
- *Case 2: $n > 3$ We pick $n = 4$ and the witness trace is $C_1C_3C_4$*

$(C_5C_6C_7C_8C_{10}C_{12}C_{13})^4C_5C_{15}$. This trace is infeasible in Figure 8.1[Pg. 148], and we proceed to the abstraction refinement step.

Counterexample-guided refinement using interpolation. *We refine the abstract segment tree of Figure 8.3[Pg. 149] using the infeasible trace. We traverse the tree top-down to refine each node of the counterexample. We refine the children of the node (through which counterexample passes), with new context information obtained from the counterexample, via interpolation. We detail our refinement approach only for A_4, the rest of the nodes are refined in a similar way. By analyzing the predecessor segments of A_4 in the counterexample, we derive $i = 0$ as a useful property for our refinement. This property is obtained using the same interpolation-based refinement process that we now describe for A_4.*

To refine A_4, we analyze its children, that is n repetitions (i.e., iterations) of A_6. In what follows, we denote by i_k the value of the variable i after the k-th iteration of A_6, for $0 \leq k \leq n$. Let i_0 denote the value of i before A_4. We compute the property $i_1 = i_0 + 1$ summarizing the first iteration of A_6, where the summarization process includes interpolation-based refinement. Similarly, from the second iteration of A_6 we compute $i_2 = i_1 + 1$. Hence, at the second iteration of A_6 the formula a $i_0 = 0 \wedge i_1 = i_0 + 1 \wedge i_2 = i_1 + 1 \wedge n = 4$ is a valid property of the witness trace; let us denote this formula by A. However, after the second iteration of A_6 we have $(i_2 < n) \wedge (i_2 < \lfloor n/2 \rfloor)$ as a valid property of the witness trace; we denote this formula by B. Observe that $A \wedge B$ is unsatisfiable, providing hence a counterexample to the feasibility of the current witness trace. From the proof of unsatisfiability of $A \wedge B$, we then compute an interpolant I such that $A \implies I$, $I \wedge B$ is unsatisfiable, and I uses only symbols common to both A and B. We derive $i_2 \geq \lfloor n/2 \rfloor$ as the interpolant of A and B.

We now use the interpolant $i_2 \geq \lfloor n/2 \rfloor$ to refine the segment abstraction of A_6, as follows. The interpolant $i_2 \geq \lfloor n/2 \rfloor$ is mapped to the transition predicate $i \geq \lfloor n/2 \rfloor$ over the program variables of Figure 8.1[Pg. 148]. We then split A_6 into two nodes: node A_6^f denoting the segment where $i \geq \lfloor n/2 \rfloor$ does not hold, and node A_6^t describing the segment on which $i \geq \lfloor n/2 \rfloor$ holds. The interpolants $i_1 = i_0 + 1$ and $i_2 = i_1 + 1$ computed from the first, respectively second iteration of A_6 yield the transition predicate $i' = i + 1$ over program variables and their primed version; this formula holds for every segment in A_6, and hence also in A_6^f and A_6^t. The transition predicates of A_6^t and A_6^f are then used to compute the new shape predicate $(A_6^f)^{\lfloor n/2 \rfloor}(A_6^t)^{n-\lfloor n/2 \rfloor}$ for A_4. The resulting (partial) refined abstract segment tree is given in Figure 8.4[Pg. 151].

After refining the abstract segment tree, we recalculate the WCET estimates using HPGs. As a result, we obtain: if $n \leq 5$ then 51 else $3 + 4n + 9\lfloor n/2 \rfloor$, a precise WCET estimate for the program in Figure 8.1[Pg. 148].

Example 8.2. *We now give a more complex example, explaining the need of length constraints on the maximum-weight HPG path.*

The program in Figure 8.5[Pg. 151] performs an operation `work()` *(of execution cost 3 time units) within a loop. In addition, every 20 loop iterations, it logs some program values into a file, by using the operation* `logValue` *whose execution takes 50 time units. Clearly, the* `logValue()` *operation is much more expensive than* `work()`*. However, in the initial abstract segment tree of Figure 8.5[Pg. 151], all loop paths are grouped in one segment. Therefore, the*

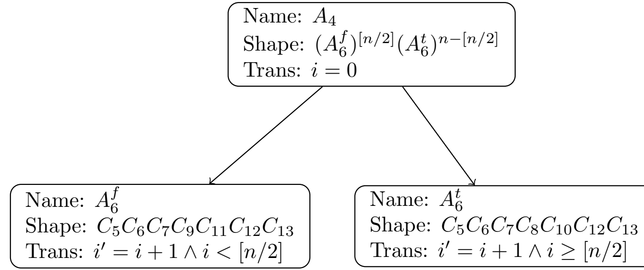


Figure 8.4: Partial structure of the refined tree of Figure 8.3[Pg. 149].

```
for (i=0;i<n;i++)
   if ((i mod 20) == 0)
      logValues()   cost=50
      work()   cost=3
   else
      work()   cost=3
```

Figure 8.5: Example 8.2[Pg. 150].

initial WCET estimate of the program considers each loop iteration to incur the cost of `logValue()`*. Let* B_1 *denote an abstract segment tree node describing all loop paths, and* B_2 *be an abstract segment tree node that represents all possible loop iterations. Similarly to Example 8.1[Pg. 147], the shape predicate of* B_1 *is computed to be* B_2^n*. The counterexample-guided refinement using interpolation will then refine the segment abstraction of* B_2*, by splitting* B_2 *into two nodes: node* (B_2^f) *where the formula (interpolant)* $((i \bmod 20) = 0)$ *holds, and* (B_2^t) *where the formula* $((i \bmod 20) = 0)$ *does not hold. The shape predicate of* B_1 *will therefore become* $((B_2^t)^1(B_2^f)^{19})^*$*, and we will keep the global bound* n *on the number of loop iterations. The segment* B_1 *is hence refined into* 1 *iteration of* B_2^t*, followed by* 19 *iterations of* B_2^f*. This information is enough to have a precise WCET estimate.*

8.3 Problem Statement

In this section, we state our counterexample-guided precision refinement problem for computing parametric WCET estimates.

Instruction and predicate language. We express program instructions, predicates, and assertions using standard first-order logic with equality. Let $\mathcal{F}(X)$ represent the set of linear integer arithmetic (Boolean-valued) *formulae* over integer variables X.

Following standard convention, we represent an instruction in a program over variables V as a formula from $\mathcal{F}(V \cup V')$. Intuitively, V' contains primed versions of all variables in V and represent the values of variables after execution of the instruction. For example, an instruction `i := i + j` in a C-like language would be represented as $i' = i + j$.

Program model. We model programs with assignments, sequencing, condi-

tionals and loops, over a finite set of scalar integer variables V. We do not handle procedure calls. However, all our techniques can be easily generalized to non-recursive procedure calls.

We represent programs by their control-flow graphs. A *control-flow graph* (CFG) is a graph $G = \langle \mathcal{C}, E, V, \Delta, \iota_0, init, F \rangle$, where (a) $\mathcal{C}$ is a set of nodes (representing control-flow locations); (b) $E \subseteq \mathcal{C} \times \mathcal{C}$ is a set of edges; (c) V is the set of program variables; (d) $\iota_0 \in \mathcal{C}$ is an initial control-flow location; (e) $init \in \mathcal{F}(V)$ is an initial condition on variables; (f) $F \subseteq \mathcal{C}$ is a set of final states; and (g) $\Delta : E \to \mathcal{F}(V \cup V')$ defines a transition relation on variables in V. Intuitively, the values of the program variables before and after executing the instruction from l_1 to l_2 satisfy $\Delta((l_1, l_2))$.

We refer to pairs of control flow locations and valuations $Val(V)$ of program variables V as *program states*. A state is denoted by a pair of the form (l, σ).

Semantics. The semantics of a CFG G, denoted by $[\![G]\!]$, is the set of *finite* sequence of program states (called *traces*) $(l_0, \sigma_0)(l_1, \sigma_1) \ldots (l_k, \sigma_k)$ such that: a $l_0 = \iota_0$ and $\sigma_0 \models init$, b $l_k \in F$, and c for all $0 \leq i < k$, we have $(l_i, l_{i+1}) \in E$ and $(\sigma_i, \sigma_{i+1}) \models \Delta((l_i, l_{i+1}))$.

Cost model. We assume a simple execution cost model for the program instructions (for extensions to more complicated cost models, see Section 8.6[Pg. 161]). We consider the function $cost : E \to \mathbb{N}$ and assume that an edge $(l_1, l_2) \in E$ *costs* $cost((l_1, l_2))$. Intuitively, $cost((l_1, l_2))$ represents the maximum execution time of the instruction from l_1 to l_2. We refer to edge costs also as *weights*.

A weight (or cost) $cost(\pi)$ of a trace in the CFG, where $\pi = (l_0, \sigma_0)(l_1, \sigma_1) \ldots (l_k, \sigma_k)$ is defined by $\Sigma_{i=0}^{k-1} cost((l_i, l_{i+1}))$.

The *worst-case execution time (WCET) of* G, denoted by $WCET(G)$, is defined as: $WCET(G) = \max_{\pi \in [\![G]\!]} cost(\pi)$.

Note that this simple cost model can already capture certain type of cache hit/miss information provided by the low-level WCET analysis [113]: for each instruction, if the low-level analysis determines that the it will always be a cache hit, it can provide the cost accordingly. An extension to a more complicated cost model is presented in Section 8.6.1[Pg. 162].

Solution language. We use a language of *disjunctive expressions* $\mathcal{E}(V)$ to represent valuation of expressions in various contexts. An element W of $\mathcal{E}(V)$ is a set of pairs $\{(D_i, N_i)\}_i$ with: (a) $D_i \in \mathcal{F}(V)$ where $\bigvee_i D_i$ holds and $D_i \wedge D_j \implies \bot$ for all $i \neq j$; and (b) N_i is either an arithmetic expression over V or ∞. Intuitively, the value of the disjunctive expression W is N_i when the condition D_i holds. If $Val(V)$ is a valuation of V, we represent by $W[Val(V)]$ the explicit integer value of evaluating W, i.e., $W[Val(V)] = N_i[V]$ if $D_i(Val(V))$ holds. For example, $\{(n < 5, n), (n \geq 5, 5)\}$ represents an expression whose value is n if $n < 5$ and 5 otherwise.

Note that it is easy to define standard arithmetic, comparison, and max operators over $\mathcal{E}(V)$. For example, if $W^1 = \{(D_i, N_i)\}_i$ and $W^2 = \{(D_j, N_j)\}_j$, then $W^1 + W^2 = \{(D_i \wedge D_j, N_i + N_j)\}$, and $\max(W^1, W^2) = \{(D_i \wedge D_j \wedge N_i > N_j)\} \cup \{(D_i \wedge D_j \wedge N_i \leq N_j)\}$.

Parameters. Let $P \subseteq V$ be a subset of program variables, called *program parameters*. Let us assume that formulas $\Delta(_)$ require that values of variables in P do not change along the transitions. Fixing a valuation $Val(P) : P \to \mathbb{N}$ of P, we obtain a new CFG $G_{Val(P)}$, by replacing in G the variables in P by their values given by $Val(P)$. A CFG is called *terminating* w.r.t. a set of parameters P, if for all valuations $Val(P)$, the supremum of the length of traces in $[\![G_{Val(P)}]\!]$

is finite.

Problem statement. We are now ready to define a parametric WCET estimate of a CFG G, and our problem statement. A *parametric WCET estimate* of a CFG G, denoted by $WCET_p(G, P)$, is an expression in $\mathcal{E}(P)$, such that for all valuations $Val(P)$ of variables in P we have $WCET_p(G, P)[Val(P)] \geq WCET(G_{Val(P)})$. Intuitively, the parametric WCET estimate, $WCET_p(G, P)$, is an over-approximation of the $WCET(G)$ for each valuation of the parameters.

The task of our *parametric WCET estimation problem* is: Given a CFG G with program variables V and a set of parameters $P \subseteq V$ such that G is terminating w.r.t. P, compute $WCET_p(G, P)$, the parametric WCET.

The rest of this chapter describes the main steps of our approach to solving this problem: segment-based abstraction, estimation of WCET using HPGs (Section 8.4[Pg. 153]), and counterexample-guided refinement with interpolation (Section 8.5[Pg. 157]). Our algorithm for parametric WCET computation is summarized in Section 8.6[Pg. 161].

8.4 Max-Weight Length-Constrained Paths

We start by fixing a set of *base nodes* BN using the CFG edges of a given program. We further consider $\mathcal{E}(P)$ as the set of all linear integer arithmetic expressions over program parameters P (see Section 8.3[Pg. 151]). The variables in P are called parameters of the HPG. Let $v : BN \to \mathcal{E}(P)$ be a function that assigns integer-valued expression (i.e. weight) to each base node.

A *hierarchic parametric graph* (HPG) is defined by the tuple $(Nodes, Trans, Init, Exit, slMin, slMax, gMax, gMin)$, where:

- $Nodes$ is a finite set such that each $h \in Nodes$ is either a base node in BN or is an HPG;
- $Trans \subseteq Nodes \times Nodes$ is a set of edges;
- $Init \subseteq Nodes$ is a set of initial nodes;
- $Exit \subseteq Nodes$ is a set of exit nodes;
- $slMin : Nodes \to \mathcal{E}(P)$ is a function that labels each $h \in Nodes$ by an integer expression defining a minimum number of consecutive repetitions of h in a trace through the HPG;
- $slMax : Nodes \to \mathcal{E}(P)$ is a function that labels each $h \in Nodes$ by an integer expression defining a maximum number of consecutive repetitions of h in an trace through the HPG;
- $gMax \in \mathcal{E}(P)$ is a bound on a maximal number of steps through the HPG.

Furthermore, we require that for a given HPG, $gMax$, $slMin$, and $slMax$ range over $\mathcal{E}(P)$ restricted to a single variable p, the same variable for all of $gMax$, $slMin$, and $slMax$. Note however, that if an HPG $\mathcal{H}$ contains a HPG $\mathcal{H}'$ as a node, then the constraints in $\mathcal{H}'$ might be over a different variable.

The HPGs are called hierarchic, as a node of an HPG can be an HPG itself. HPGs are defined to be non-recursive, i.e., there is no sequence of HPGs H_0, H_1, …, H_n where $H_0 = H_n$ and each H_i is a node in H_{i+1}. Note that the set BN of base nodes has been fixed, and hence all HPGs share the same set of base nodes. In particular, if an HPG $\mathcal{H}'$ is a descendant of an HPG $\mathcal{H}$, they share the same set of base nodes BN. The semantics of HPGs will further be defined in terms of sequences of base nodes.

The HPGs are parametric because they are a form of weighted graphs where

weights (and lengths) are expressions over a variable p. Using these weights, we define an optimization problem over HPGs, solving the task of parametric WCET computation. This optimization problem will be parametric over p. That is, we do not look for an optimal value of p, but for a solution in terms of p.

Example 8.3. *Consider our running example from Example 8.1[Pg. 147]. The HPGs we construct are built from abstract segment trees, as illustrated in Section 8.2[Pg. 147] (and formally defined in Section 8.5[Pg. 157]). For the abstract segment tree in Figure 8.3[Pg. 149], each node gives rise to an HPG. Consider the node A_2 in Figure 8.3[Pg. 149]. It yields an HPG with three nodes A_3, A_4, A_5, each of which is an HPG. There are just two transitions: from A_3 to A_4 and from A_4 to A_5. Consider the node A_4, which corresponds to an HPG with just one node, A_6. This node has a self-loop. For node A_6, both slMax and slMin are set to n, corresponding to the loop bound for the loop represented by A_4 (the node A_6 represents one iteration of the loop).*

Semantics of HPGs The semantics of HPGs is defined in terms of sequences of base nodes and parameters P. We define the semantics with respect to an arbitrary valuation $Val(P)$ of P. Given an HPG $\mathcal{H}$, we the HPG $\mathcal{H}_{Val(P)}$ by replacing expressions from $\mathcal{E}(P)$ with their valuations on $Val(P)$. The semantics $[\![\mathcal{H}_{Val(P)}]\!]$ is a subset of the set BN^* of finite sequences of base nodes, as follows.

Let us consider an HPG $\mathcal{H} = (Nodes, Trans, Init, Exit, slMin, slMax, gMax)$. A *one-level path* of $\mathcal{H}$ is a sequence $h_0 h_1 \dots h_n$ of nodes such that $h_0 \in Init$, $h_n \in Exit$, and for all $i < n$, we have $(h_i, h_{i+1}) \in Trans$. Further, for each maximal subsequence $t_0 t_1 \dots t_v$ such that $t_0 = t_1 = \dots = t_v$, we have $slMin(t_0) \leq v+1 \leq slMax(t_0)$.

Let $s = b_0 b_1 \dots b_k$ be a sequence of base nodes. Then, the sequence s is in $[\![\mathcal{H}_{Val(P)}]\!]$ if there is a one-level path $t = t_0 t_1 \dots t_r$ in $\mathcal{H}$ with $r+1 \leq gMax$, such that there exist sequences $s_0, s_1, \dots, s_r$ of base nodes satisfying the following conditions:

- $s = s_0 s_1 \dots s_k$;
- for each $i \leq r$, we have that: (a) if t_i is an HPG, then $s_i \in [\![t_i]\!]$, and (b) if t_i is a base node b, then $s_i = b$.

Let $s = b_0 b_1 \dots b_k$ be a sequence of base nodes in $[\![\mathcal{H}]\!]$. The weight of s, that is $v(s)$, is the sum of the weights of individual nodes. Namely, $v(s) = \sum_{i=0}^{k} v(b_i)$. Note that while the weight is computed for the entire path, the length constraints, given by $gMax$ for each $\mathcal{H}$, are enforced at each level separately. Based on the above notations, the problem statement from Section 8.3[Pg. 151] can now be reformulated in terms of HPGs as follows. To derive a parametric WCET, we solve the problem of computing the maximum-weight length-constrained path in a hierarchic parametric graphs. Given an HPG $\mathcal{H}$, we define the *maximum-weight length-constrained path in hierarchic parametric graphs*, denoted by $mwp(\mathcal{H})$, as an expression in $\mathcal{E}(P)$ such that for all valuations $Val(P)$ of the parameters P, we have that $mwp(\mathcal{H})[Val(P)] = \max\{v(s) \mid s \in [\![\mathcal{H}_{Val(P)}]\!]\}$.

8.4.1 Computation of Maximum-Weight Length-Constrained Paths

Given an HPG $\mathcal{H}$, we now describe our approach for computing the maximum-weight length-constrained path for each HPG node.

Let $\mathcal{H}$ be an HPG which contains $\mathcal{H}'$ as a node. If we find the maximum-weight length-constrained path for $\mathcal{H}'$, we can use it as a weight of the node $\mathcal{H}'$ in solving the problem for $\mathcal{H}$. Therefore, it is enough to give an algorithm for solving the problem for one node — we can assume that each node is a base node, with a given weight. For ease of presentation, we first present the algorithm for a special case when for each node k, we have $slMin(k) = slMax(k)$. Let us first give some definitions that enable us to explain our algorithm.

A set of base nodes L is a *loop set*, if there is a loop that contains each node in L exactly once. The weight of L ($v(L)$) is defined as the sum of the weights of the node in L, where each node n is counted *slMin* times.

Let $\mathcal{H}$ be an HPG with n base nodes (and no HPG nodes), and with *gMax* of $\mathcal{H}$ being denoted by N. Let $s = b_0 b_1 \dots b_k$ be a finite sequence of base nodes that starts at an initial node b_0 and ends with a final node b_k. A *loop set decomposition* of a path π is a tuple $(spath, (L_1, k_1), \dots, (L_m, k_m))$, where (a) the *spath* is a simple path starting in an initial node and ending in a final node (its length is at most n); (b) for each $1 \leq i \leq k$, L_i is a loop set, and k_i is a number of times a loop with this loop set was taken in the path; (c) all L_i are distinct. It is straightforward to show that every path can be decomposed to this form. For a path π, we write $LD(\pi)$ for its loop set decomposition.

We define an equivalence relation on paths as follows. Two paths π and π' are equivalent, denoted by $\pi \equiv \pi'$, if their projections involve the same loop sets, i.e. if $LD(\pi) = (spath, (L_1, k_1), \dots, (L_m, k_m))$ and $LD(\pi') = (spath', (L_1, k_1'), \dots, (L_m, k_m'))$.

A *one-heavy-loop (OHL) path* is a path where, intuitively, the path "spends as much time as possible" in the maximum-weight numeric loop it encountered. Let $|L|$ be the size of a loop set and $v(L)$ the weight of the loop set. A loop set L is *numeric* if the weights of the nodes are all numeric expressions, that is not expressions in $\mathcal{E}(P)$. Let π be a path, $LD(\pi) = (spath, (L_1, k_1), \dots, (L_m, k_m))$ be its decomposition, and let L_q be a loop with maximal weight-average in $\{L_1, L_2, \dots, L_m\}$. The weight average of a loop L is the ratio of $v(L)/|L|$. The path π is an OHL path if, for all $i \in [1, m]$, such that $i \neq q$, we have that for the $k_i \cdot |L_i| < LCM(|L_i|, |L_q|)$, where LCM denotes the least common multiple function.

The following key lemma states that the maximum-weight in an equivalence class is achieved by an OHL path.

Lemma 8.4. *Let R be an equivalence class of the equivalence relation $\equiv$. There exists an OHL path π in R such that for all path π' in R, we have that $v(\pi) \geq v(\pi')$.*

Proof. Let π be a path, $LD(\pi) = (spath, (L_1, k_1), \dots, (L_m, k_m))$ be its decomposition, and L_q a loop with maximal weight-average in $\{L_1, L_2, \dots, L_m\}$. The main idea of the proof is to show that if a path takes too many iterations of a numeric loop L_i other than L_q, we can obtain a path with greater or equal weight by "moving" some iterations to L_q. Formally, let us assume that there is a pair (L_i, k_i) such that the weight-average $v(L_i)/|L_i|$ of L_i is smaller than $v(L_q)/|L_q|$, and $k_i \cdot |L_i| \geq LCM(|L_i|, |L_q|)$. Then there exists k_i' such that $k_i' \leq k_i$, and $k_i' \cdot |L_i| = LCM(|L_i|, |L_q|)$. Let us consider these k_i' iterations of $|L_i|$. We obtain that for these $k_i' \cdot |L_i'|$ steps the trace gets $k_i' \cdot v(L_i)$ weight. On the other hand, if we iterate over $|L_q|$, for the same $k_i' \cdot |L_i'|$ steps, we get

$r = \frac{k'_i \cdot |L'_i|}{|L_q|}$ iterations, and a weight of $r \cdot v(L_q)$. Due to the assumption that $v(L_i)/|L_i| \leq v(L_q)/|L_q|$, we derive $r \cdot v(L_q) \geq k'_i \cdot v(L_i)$.We thus conclude that given a path in the equivalence path R, we can transform it into an OHL path in R with equal or greater weight. □

Lemma 8.4[Pg. 155] gives us a bound on a number of iterations of numeric loops that we need to consider. We can obtain also a constant bound on the number of iterations of symbolic loops. For a symbolic loop L, we can bound the number of iterations by over-approximating (with a constant) the ratio $gMax/v(L)$ for a symbolic loop L. This is possible using a (simple) algebraic solver, as in HPGs both *gMax* and all weights are linear expressions.
Maximum-weight length-constrained path computation in HPGs. Our algorithm for computing the maximum-weight length-constrained path through an HPG $\mathcal{H}$ is given in Algorithm 6[Pg. 156]. In line 1, the bounds on the number of iterations of numeric and symbolic loops are computed. The algorithm uses the result of Lemma 8.4[Pg. 155]: it first generates structures of the form $(spath, (L_1, k_1), \ldots, (L_m, k_m))$, that are decompositions of paths (line 2). Then, for each structure that is well-formed (i.e., connected) and bounded for numeric and symbolic bounds (line 3), we calculate the maximum weight represented by this decomposition (Line 6). The decomposition `dS` represents an equivalence class of paths. The function `calculateMaxWeight()` constructs an OHL path by adding as many iterations as possible of the loop with the highest weight-average such that the total number of steps is less then *gMax* of the input HPG $\mathcal{H}$. The function `calculateMaxWeight()` then returns the length of that path, and the path itself. The path returned by Algorithm 6[Pg. 156] will be used as the witness trace to check if we need to refine the segment-based abstraction.

Algorithm 6 MWPforHPG Algorithm.

Input: $\mathcal{H}$: HPG
```
1: (mxV,pth) ← 0; numB ← NumBnd(H); symB ← SymBnd(H)
2: for all dS ∈ Decomp(H) do
3:   if isCon(dS) ∧ isBound(dS,symB,numB) then
4:     (m,pth') ← calculateMaxWeight(dS)
5:     if mxV < m then (mxV,pth) ← (m,pth')
6:
7: return (maxV,path)
```

In Lemma 8.4[Pg. 155] and Algorithm 6[Pg. 156], we assumed $slMin(n) = slMax(n)$, for all nodes n. It can be shown that the general problem of finding the $mwp(\mathcal{H})$ in a HPG $\mathcal{H}$ can be reduced to this problem, where the above assumption holds by a polynomial-time algorithm which is based on a similar idea as the proof of Lemma 8.4[Pg. 155]. We thus have the following result.

Theorem 8.5. *For all HPGs $\mathcal{H}$,* MWPforHPG$(\mathcal{H}) = mwp(\mathcal{H})$.

This theorem states that Algorithm 6[Pg. 156] finds the maximum-weight length-constrained path. The key part of the correctness proof of Theorem 8.5[Pg. 156] is provided by Lemma 8.4[Pg. 155]. Note that Algorithm 6[Pg. 156] uses time doubly-exponential in the size of the input HPG. In practice, the high computational complexity is alleviated by the hierarchical approach of our

abstraction, where we can use the structure of the program to obtain graphs with a very simple shape. One can show that for loops arising from structured programs, the computational complexity of Algorithm 6[Pg. 156] is (singly) exponential. Furthermore, we can use the structure of the program to construct a hierarchical abstraction to obtain graphs that are small enough. Our experiments in Section 8.7[Pg. 163] show that, despite its theoretical high complexity, Algorithm 6[Pg. 156] scales very well for challenging examples.

8.5 Interpolation for Segment-Based Abstraction Refinement

We briefly recall the hierarchical segment-based abstraction of Chapter 7. Let us fix a CFG $G = \langle \mathcal{C}, E, V, \Delta, \iota_0, init, F \rangle$ with a set of parameters $P \subseteq V$ for the remainder of this section. A *segment* is a finite sequence of program states (recall that a program state is a pair (l, σ), where l is a control-flow location in $\mathcal{C}$, and σ is a valuation of variables V).

Abstract Segment Trees (ASTs). An *abstract segment tree* T is a rooted tree, where each node represents a set of segments. Each node is given by a tuple $(segPred, children, shape)$, where:

- the set *children* is such that for internal nodes, *children* is a subset of nodes in T, and for leaf nodes, *children* is a subset of $\mathcal{C}$, the set of control-flow locations of G;
- $shape \subseteq children \times children$ is a transition relation on *children*;
- the predicate *segPred* is given by a formula in $\mathcal{F}(V \cup V')$.

For an AST T, we define an AST $T_{val(P)}$ obtained by fixing the valuation $Val(P)$ of parameters.

Figure 8.3[Pg. 149] illustrates an abstract segment tree as defined above.

Semantics of ASTs. The semantics $[\![T]\!]$ of an AST T is a set of segments. We will first define $[\![A]\!]$ for a node A in terms of its children. The semantics of T is then the semantics of the root node.

Let $\pi = (l_0, \sigma_0) \ldots (l_k, \sigma_k)$ be a segment in the CFG G. We will need a function $form(\pi)$ that defines a formula representing the path by composition of formulas $\Delta((l_i, l_{i+1}))$ for $0 \leq i < k$.

We start by defining $[\![A]\!]$ for leaf nodes A. Let $A = (children, shape, segPred)$ be a leaf node. Recall that the set of children of a leaf node A is a subset of nodes in the CFG G. A segment $s = (l_0, \sigma_0) \ldots (l_k, \sigma_k)$ is in $[\![A]\!]$, iff: (a) for all i such that $0 \leq i \leq k$, we have that l_i is in *children*, and (b) for all i such that $0 \leq i < k$ we have that $shape(l_i, l_{i+1})$, and (c) $form(s)$ implies *segPred*.

Let $A = (segPred, children, shape)$ be an internal node in an AST T. A segment $s = (l_0, \sigma_0) \ldots (l_k, \sigma_k)$ is in $[\![A]\!]$, if there are segments $s_0, s_1, \ldots s_r$ such that: (a) $s = s_0 s_1 \ldots s_r$, (b) there exist $c_0, c_1, \ldots, c_r \in children$ such that for each i such that $0 \leq i \leq r$, we have that $s_i \in [\![c_i]\!]$, and for each i such that $0 \leq i < r$, we have that $shape(l_i, l_{i+1})$, and (c) $form(s)$ implies *segPred*.

Example 8.6. *Consider the program of Figure 8.1[Pg. 148]. Its CFG is Figure 8.2[Pg. 148], and its AST is Figure 8.3[Pg. 149]. Given a segment* $\pi = (l_0, \sigma_0)(l_1, \sigma_1) \ldots (l_k, \sigma_k)$*, we define its projection to CFG locations* $w(\pi) = l_0 l_1 \ldots l_k$*. A segment* π *such that* $w(\pi) = C_1C_3C_4C_5C_6C_7C_8C_{10}C_{13}C_5C_{15}$ *is in the semantics of the node* A_2 *of the AST, as it can be split into into*

three segments: (a) π_1, such that $w(\pi_1) = C_1C_3C_4C_5$; (b) π_2, such that $w(\pi_2) = C_5C_6C_7C_8C_{10}C_{13}C_5$; and (c) π_3, such that $w(\pi_3) = C_5C_{15}$. Thus, π_1 is in $[\![A_3]\!]$, π_2 is in $[\![A_4]\!]$, and π_3 is in $[\![A_5]\!]$.

Reducibility of CFGs. We assume that CFGs are reducible and recursively reducible. These assumptions hold for CFGs for programs in high-level programming languages. Reducibility means that every maximal strongly connected component has a single entry and exit point. Recursive reducibility means that if we remove a maximal strongly connected component form a CFG, the resulting graph is still reducible.

Initial abstraction. We describe a function $InitAbs(G)$ that takes a CFG as an input, and constructs an abstract segment tree T such that $[\![T]\!]$ is a superset of $[\![G]\!]$. The function *InitAbs* uses just the structure of reducible and recursively reducible CFGs. The idea of the construction is simple: each maximal strongly connected component (i.e., a loop) will correspond to a node with just one child. The child will represent the segment in the loop, that is, individual iterations. An example of this construction is the construction of nodes A_4 and A_6 in Figure 8.3[Pg. 149].

The *segPred* predicate for each node is initially set to true. The leaf nodes correspond to sequences of instructions in a straight-line code. As an example, consider again the program in Figure 8.1[Pg. 148], and its initial abstraction in Figure 8.3[Pg. 149]. Note however that the tree in Figure 8.3[Pg. 149] has information on loop bounds marked in the extended regular expressions, such as A_6^n. For ASTs, the loop bounds can be computed from the predicate *segPred*.

Proposition 8.7 (Soundness of *InitAbs*)**.** *Let G be a parametric CFG with parameters P. If $T = InitAbs(G)$, then for all valuations $Val(P)$ of parameters in P, we have that $[\![G_{Val(P)}]\!] \subseteq [\![T_{Val(P)}]\!]$.*

From ASTs to HPGs. We define a function, called $astToHPG(T)$, that given an AST T constructs the HPG of T. HPGs and ASTs have similar hierarchical structure. The purpose of having two definitions is to cleanly split the program analysis concerns about discovering predicates in the process of refinement (which is the goal of ASTs) from the optimization problem (which is the goal of HPGs).

The mapping from ASTs to HPGs is simple. For each internal AST node, we create a node of the HPG. We use the transition system defined by the *shape* transition relation of each AST node to construct the HPG. Leaf nodes of the AST will correspond to base nodes of the constructed HPG. We thus need to compute the weight of the leaf nodes. Let A be a leaf node of the AST. Recall that a leaf node produced by *InitAbs* represents segments corresponding to a straight-line code. We assume there exists a function $WCET(segPred_W, shape_W)$ that over-approximates (with a constant) the WCET of segments represented by leaf nodes. In practice, we use a low-level WCET analysis tool for this purpose.

We need to define how the bounds $slMax, slMin, gMax$ associated with a node of an HPG are computed. This process corresponds to computing loop bounds In principle, the bounds $slMax, slMin, gMax$ of an HPG node can be computed using the *segPred* predicate for the corresponding AST node, its parent, and its siblings. If *segPred* predicates are not precise enough, the bounds

slMax, *slMin*, *gMax* can be ∞. In practice, we use an external tool to infer loop bounds [113].

Here, we only illustrate how *slMax* can be computed using *segPred*; the other bounds are handled similarly. Recall that *slMax* is the bound on how many times a node can be repeated. If the node represents the loop body, *slMax* corresponds to a loop bound. Consider the following example. Let A be a node such that its *segPred* predicate implies $i = 0$ at the beginning of all segments in $[\![A]\!]$ and $i' = n$ at the end. Let A have an only child B, such that its *segPred* implies $i' = i + 1$. We can then infer a ranking function that shows that there can be at most n repetitions of B, which gives us *slMax* for B.

Summarizing, the function $astToHPG(T)$ constructs the HPG of a given AST T as detailed above. The corollary shows soundness of the inital abstraction. It is a consequence of Proposition 8.7[Pg. 158] and Theorem 8.5[Pg. 156].

Corollary 8.8. *Let G be a parametric CFG with parameters P, and T an AST such that $T = InitAbs(G)$. Let $\mathcal{H}$ be an HPG such that $\mathcal{H} = astToHPG(T)$. Then* $\mathrm{MWPforHPG}(\mathcal{H}) \geq WCET_p(G, P)$.

8.5.1 Interpolation

The choice of interpolants depends on the application in which they are used – applications may prefer logically strong interpolants [105], interpolants of a particular shape [3], or minimal with respect to various measures [100]. In our experiments, we decided to use the method of [100] and compute quantifier-free interpolants with a minimal number of components and symbols.

Example 8.9. *Let A be the formula $(i_0 = 0) \wedge (i_1 = i_0 + 1) \wedge (i_2 = i_1 + 1) \wedge (n = 4)$ and take B as $(i_2 < n) \wedge (2 * i_2 < n)$ (these formulae come from the running Example 8.1[Pg. 147]). Two possible interpolants I of A and B are: (a) $I_1 = (n = 4 \wedge i_2 = 2)$; (b) $I_2 = 2 * i_2 \geq n$. However, the interpolant I_1 is specific for the current choice of n and hence will not be useful in our application.*

8.5.2 Abstraction Refinement

We now describe our segment-based abstraction refinement algorithm using interpolation for an infeasible witness trace. A *witness trace wit* is a sequence of CFG nodes that witnesses the current WCET estimate. It is obtained by HPG analysis.

AST refinement algorithm. The main idea of the algorithm, given in Algorithm 7[Pg. 160] is to trace the *infeasible* witness trace (*wit*) through the abstract segment tree (AST), and refine the AST nodes touched by the *wit*. For each node N, we discover the segment predicates that are important at the interface of the subtree rooted at N and the rest of the AST. When processing an AST node, we split each child (visited by the *wit*) with some new "context" information, obtained via interpolation. Algorithm 7[Pg. 160] takes four inputs: (a) an AST T, (b) a node N in T, (c) an infeasible witness trace *wit* that is a segment in N, and (d) a formula *SumAbove* that summarizes the part of the original witness trace *wit* outside of the subtree rooted at N. Initially, the algorithm is called with N being the root of the AST, and the formula *SumAbove* is set to *true*.

Algorithm 7 Procedure Refine

Input: AST T, node N in T, witness trace wit and formula $SumAbove$
Output: Refined AST T

1: $s \leftarrow$ TraceWit(N,wit)
2: **for all** $i \in \{0, \ldots, |s|\}$ **do**
3: context $\leftarrow$ SumAbove $\wedge$ SumLR(s,wit,N) $\wedge$ $segPred(N)$
4: child $\leftarrow$ $form$(projection(wit,s_i))
5: I $\leftarrow$ Interpolate(child,context) {context $\wedge$ child unsat}
6: $r_t \leftarrow$ addToTree(s_i,I); $r_f \leftarrow$ addToTree(s_i,$\neg I$)
7: REFINE(T, r_t, $I\wedge$ $segPred(s_i)$, projection(wit,s_i))
{Recursively refine.}
8: StrengthenDown(r_f)
9: RemoveFromTree(T,s_i)
10: StrengthenUp(N)

Refinement procedure. We now detail the REFINE procedure of Algorithm 7[Pg. 160] and illustrate it on our running example. For a node N, the procedure REFINE obtains a sequence $s = s_0s_1 \ldots s_k$ of children of N that the witness trace wit passes through (line 1 of Algorithm 7[Pg. 160]). Note that a child can be repeated in s. The wit can be split into segments, where the i-th segment of wit, denoted as (wit_i), belongs to the i-th child s_i. Recall that the infeasible wit of Example 8.1[Pg. 147] was $wit = C_1C_3C_4\ (C_5C_6C_7C_8C_{10}C_{12}C_{13})^4C_5C_{15}$, obtained for $n = 4$. The CFG of Example 8.1[Pg. 147] is Figure 8.2[Pg. 148], and its AST is given in Figure 8.3[Pg. 149]. Consider the node A_4 in Figure 8.3[Pg. 149] as the node N. The node A_4 represents a loop and the node A_6 a single iteration. The sequence s is then $A_6A_6A_6A_6$ (recall $n = 4$).

Next, each child s_i is refined using wit_i (loop at line 2). The variable $context$ stores a formula that summarizes what we know about the trace wit outside of s_i (line 3). It is obtained as a conjunction of the formula $SumAbove$, the segment predicate of N, and the information computed by the function $SumLR()$. The function $SumLR()$ computes information about the trace wit as it passes through the children of N other than s_i. When refining s_i, $SumLR()$ returns a formula $\wedge_{k<|s|\wedge(k\neq i)}J_k$, where J_k is $form(wit_k)$ and wit_k is the part of the wit going through the node s_k. The variable $child$ stores a formula that summarizes what we know about the wit inside of s_i (line 4). It is computed as $form(wit_i)$, where wit_i was obtained by the projection of wit to the node s_i. For our running example, consider the third iteration of A_6. In this case, the value of `context` is $n = 4 \wedge i_0 = 0 \wedge i_1 = i_0 + 1 \wedge i_2 = i_1 + 1$ (we show only the relevant part of the formula) and the value of `child` is $i_2 < n \wedge i_2 < \lfloor n/2 \rfloor$.

Note that the formula $child \wedge context$ is unsatisfiable, as (a) the original wit is infeasible, and (b) $context$ and $child$ summarize the wit. We hence can use interpolation to infer a predicate that explains the infeasibility of the wit at the boundary of the subtree of child s_i and the rest of the AST. We compute an interpolant I from the proof of unsatisfiability $context \wedge child$ (line 5) such that $context \implies I$ and $child \wedge I \implies \bot$. In our running example, we obtain the interpolant $i_2 \geq \lfloor n/2 \rfloor$.

Using the computed interpolant I, we next replace the node s_i by two nodes r_t and r_f (line 6). The node r_t is like s_i (in terms of its children in the AST),

Algorithm 8 Counterexample-Guided Precision Refinement for Parametric WCET Estimates

1: **Input:** Program $\mathcal{P}$ with a set of parameters P
2: **Output:** Parametric WCET of $\mathcal{P}$
3: Build the CFG G of $\mathcal{P}$;
4: Construct the AST T corresponding to G; // ***Abstraction***
5: **for all** nodes A in T **do**
6: construct the HPG $\mathcal{G}_A$ of A;
7: compute $cost(A) \leftarrow WCET(A)$ of A; (Algorithm 6[Pg. 156]).
8: set $WCET_P(G, P) \leftarrow cost(A_0)$; // ***WCET estimates***
9: **for all** each $\theta \in \mathcal{F}(P)$ in the parametric $WCET_P(G, P)$ **do**
10: let w_i be the witness trace provided by Algorithm 6[Pg. 156]
11: instantiate parameters P by concrete values satisfying θ
12: check feasibility of w_i' for θ;
13: **if** w_i is infeasible **then**
14: Refine(T,w_i,root(T),true) (Alg. 7[Pg. 160]) // ***Refinement***
15: **go to** line 5[Pg. 161];
16:
17: **return** the parametric $WCET_P(G, P)$.

but has a transition predicate equal to $segPred(s_i) \wedge I$. Similarly, for r_f we take its transition predicate $segPred(r_f)$ as $segPred(s_i) \wedge \neg I$. In this way, each of these nodes has more information about its context than s_i had. We can further refine these two nodes and use them in the AST T instead of s_i.

Observe that for r_t we added the predicate I to its transition predicate. As $child \wedge I$ is unsatisfiable, the trace *wit* is not represented in r_t. The node r_t can thus be refined by a recursive call to the REFINE procedure (line 8). As $child \wedge \neg I$ is satisfiable, for r_f there is nothing more to learn from the *wit*. We simply need to strengthen the node, that is, propagate the new predicate, $\neg I$, to the children of r_f. This is done by calling the *StrengthenDown*() function (line 10), which propagates the new information $\neg I$ to the children t of the node r_f. To this end, it checks whether it finds a segment in the node t which is excluded from the node by $\neg I$, and then calls the REFINE procedure to perform refinement with the discovered segment used as a witness trace. In our running example r_f corresponds to A_6^f and r_t is A_6^t.

Finally, the function *StrengthenUp*() uses the information discovered during the refinement process for the children of a node N, and strengthens the *segPred* predicate of N (line 11).

8.6 Parametric WCET Computation

Algorithm 8[Pg. 161] describes our approach to computing precise parametric WCET estimates. Given a program $\mathcal{P}$, we first construct its CFG (line 3[Pg. 161]), and build the corresponding initial AST (line 4[Pg. 161]). From the AST, we construct an HPG and derive the WCET (lines 5[Pg. 161]-7[Pg. 161]), using Algorithm 6[Pg. 156]. The WCET is precise if a feasible program path exhibits the WCET. We therefore check if the witness trace exhibiting the WCET is indeed feasible (lines 9[Pg. 161]-15[Pg. 161]). If not, we refine our current AST

using Algorithm 7[Pg. 160]. Our method iteratively refines the AST and returns more precise parametric WCET estimates.

8.6.1 Improvements and Extensions

We now describe some improvements to Algorithm 8[Pg. 161].

Dependent loops. In the HPG evaluation algorithm (Algorithm 6[Pg. 156]), while computing the cost for a node, we always assume the worst-case cost of inner nodes. However, there is a common case where this gives bad estimates of the actual worst-case cost of the node, namely in the case of the dependent loops.

For example, consider the program from Figure 8.7[Pg. 163]. Now, the worst-case cost of an iteration of the outer loop is $n \cdot k$ (where k is the cost of an iteration of the inner loop). Using this worst-case cost, we get that the worst-case cost of the outer loop is $n^2 \cdot k$; note the non-linear expression! However, the cost of the inner-loop is only $k \cdot i$ in the ith iteration and the estimated cost is imprecise. In this case, we can incorporate the precise cost of the child node while computing the cost of the parent node, i.e., the cost of the outer loop is $\sum_{i=0}^{n-1} i \cdot k$, leading to a more precise estimate. Intuitively, using this method, when the cost of the child node in the ith repetition is a polynomial in i (say $p(i)$) we can compute the more precise estimates as $\sum_{i=1}^{n} p(i)$. We implemented this optimization, resulting in improvements of WCET estimates in the presence of nested loops. Note that this approach leads to non-linear expressions. The problem of checking whether an expression is always greater than another becomes undecidable for non-linear expressions. However, in all our experiments, the non-linear solver we use (Z3) handled these expressions.

Cost model. The simple cost model described Section 8.3[Pg. 151] is restrictive in regards to incorporating various timing features of processors, as it can incorporate only the *must-hit/may-miss* cache information. State-of-the-art WCET analysis techniques can however incorporate also other information like cache-persistence [171] (i.e., an instruction may cause a cache-miss the first time it is executed, but will successively be cache-hits). The common high-level technique to used to incorporate such information into WCET analysis is the implicit-path enumeration technique (IPET) [171]. In this technique, an integer linear program (ILP) is built containing the execution frequencies f_i and cache-hit f_i^h and cache-miss f_i^m frequencies of each instruction i. The ILP contains constraints $f_i = f_i^h + f_i^m$ and additional constraints $f_i^h = 0$, $f_i^m = 0$, or $f_i^m \leq$ based on whether the instruction is always-miss, always-hit or persistent.

Note that the structure of the HPG generated by our method incorporates all of the above constraints except for the persistence constraint. Persistence constraints can be handled by splitting the HPG node containing the corresponding instruction into two – where in one node, the instruction has cost of a cache-miss and in the other, it has cost of a cache-hit. The former node will be given *slMin* and *slMax* values 1. In this way, we can incorporate more complex cost-models into our algorithm. In fact, one can show that theoretically, the HPG can encode every cost model encodable through IPET.

Low-level timing refinement. Computing the instruction costs is usually done through abstract interpretation. For example, in [171], invariants from an abstract domain are used to compute cache-hit/miss information. A standard problem with such analyses is that they are, by default, context-insensitive.

```
n:=0;
while(n < iters)
  if(health==round0)
    HighVoltageCurrent(health)
    UpdatePeriod(temp, 5)
    if(hit_trigger_flag==0)
        ResetPeakDetector()
  if(health==round1)
      ...
  if(health==round4)
      LowVoltageCurrent()
  ...
  if(health!=0)
      health--
  else
      health=9
  n++
```

Figure 8.6: part of `ex2` from Debie suite.

```
for (i=0; i<n; i++){
 for (j=i+1; j<n; j++){
    B.re[j][i] = B.re[i][j];
    B.im[j][i] = -B.im[i][j];}
 B.im[i][i] = 0.0;}
```

Figure 8.7: part of `ex7` from Jampack.

Example 8.10. *Consider the program in Figure 8.5[Pg. 151]. Suppose that execution of the then- and else-branches move* `i` *out of and into the cache respectively. When performing flow-insensitive cache analysis of the loop as a whole, the variable* `i` *is not persistent in the cache and the analysis assumes cache-misses for each iteration. However, suppose we refine the set of iterations of the loop into sets containing iterations taking the then branch (A^t) and else branch (A^f). When computing costs of n continuous iterations of $(A^f)^*$, one can see that* `i` *is persistent in $(A^f)^*$. Incorporating this persistence information into our method, we can see that a cache-miss occurs for* `i` *only once every* 20 *iterations. This is a significant improvement over the initial estimate of a cache-miss every iteration.*

Example 8.10[Pg. 163] shows that an added advantage of segment-based abstraction is that one can refine low-level timing information for instructions in a segment using the context of the segment. This provides a systematic way of incorporating context information into low-level WCET analysis – avoiding common heuristics like loop-peeling, context-bounding, etc.

8.7 Experimental Evaluation

We implemented our approach in a tool called IBART. It takes C programs (with no procedure calls) as inputs, and returns a parametric WCET estimate for the program as output.

CFG construction. Our implementation analyzes the WCET of programs run on an Infineon C167 processor, and uses the CalcWcet167 low-level WCET analyzer [112] to compute instruction costs at the binary level. By exploiting the architecture-aware framework of r-TuBound, IBART is cache-aware.

Abstraction and Refinement. IBART implements new methods for constructing the ASTs from CFGs, building HPGs from ASTs, computing WCET estimates using HPGs, and refining the segment-based abstraction using interpolation, as detailed in Sections 8.4[Pg. 153]-8.5[Pg. 157].
Interpolation. For computing interpolants, IBART relies on Vampire [115]. IBART implements a heuristic on top of the interpolant minimization [100]. Our interpolants are minimal both in the number of their components and symbols, are quantifier-free, and contain a minimal number of disjunctions. We prefer conjunctive interpolants instead of disjunctive ones,
WCET estimates. For solving HPG constraints over WCET, we implemented a new algebraic solver in IBART, and derive the parametric WCET as a solution of max-expressions over linear integer arithmetic formulas. Our solver returns relational properties among program variables, and not a concrete assignment to program variables, for which the maximum-weight length-constrained problem of WCET computation is satisfied.
Benchmarks. We evaluated IBART on 10 examples (examples 2 to 11 in Table 8.1[Pg. 165]) taken from WCET benchmark suites and open-source linear algebra packages. We used small, but challenging examples. Of the 10 examples, 3 are small functions with less than 30 lines of code; the remaining 7 have between 34 and 109 lines of code. The examples were chosen to be challenging for WCET analysis, due to two features: (a) branching statements within loops, leading to branches with different execution times, and (b) nested loops, whose inner loops linearly depend on the outer loops.
WCET benchmark suites. We used the Debie and the Mälardalen benchmark suite from the WCET community [171], which are commonly used for evaluating WCET tools. We analyzed one larger example (109 lines) from the Debie examples (`ex2` in Table 8.1[Pg. 165]) and 4 programs from the Mälardalen suite. The parametric timing behavior of these examples comes from the presence of symbolic loop bounds. An excerpt from the Debie example is shown in Figure 8.6[Pg. 163].

Note that in Figure 8.6[Pg. 163], different paths in the loop body have different execution times. Moreover, every conditional branch is revisited at every tenth iteration of the loop. Computing the WCET of the program by taking the most expensive conditional branch at every loop iteration would thus yield a pessimistic overestimate of the actual WCET. Our approach derives a tight parametric WCET by identifying the set of feasible program paths at each loop iteration.
Linear algebra packages. We used 5 examples from the open-source linear algebra libraries JAMA and Jampack. These packages provide user-level classes for constructing and implement non-trivial mathematical operations, including inverse calculation (`ex7`), singular value decomposition (`ex8`), triangularization (`ex9`), and eigenvalue decomposition (`ex10`, `ex11`) of matrices in Java. We manually translated them to C. The control flow of these benchmarks contains nested loops, sometimes with conditionals, where inner loops linearly depend on outer loops. Figure 8.7[Pg. 163] gives a partial and simplified version of a Jampack class implementing matrix inverse operations. Observe that the inner loop of this example linearly depends on the outer loop, and the WCET of the program depends on the (matrix) dimension `n`. IBART computes the WCET of the program as a symbolic expression in this parameter (Table 8.1[Pg. 165]).
Results. We first evaluated IBART for parametric WCET computation. Next,

we compared IBART with a state-of-the-art WCET analyzer. All results were performed on a 64-bit 2.2 GHz Intel Core i7 CPU with 8 GB RAM and obtained in less than 60 seconds.

Ex	Source/File	Parametric WCET
ex1	Section 8.2[Pg. 147]	$\{(n \leq 5, 24940),$ $(n \geq 6, 5040 + 2800\lfloor n/2 \rfloor + 1900n)\}$
ex2	Debie/ health	$\{(n \leq 0, 2620),$ $(n > 0, 2620 + \lfloor n/10 \rfloor 59100+$ $(n - \lfloor n/10 \rfloor \times 10) * 6800)\}$
ex3	Mälardalen/ adpcm	$\{(dlt \neq 0, 4180 + 5060n),$ $(dlt = 0, 4260 + 2500n)\}$
ex4	Mälardalen/ crc	$\{(jrev > 0, 5560 + 3860len),$ $(jrev \leq 0, 4320 + 3380len)\}$
ex5	Mälardalen/ crc	$\{(len \geq -1 \wedge init = 0, 7800 + 3840len),$ $(init \neq 0, 3060)\}$
ex6	Mälardalen/ lcdnum	$\{(n \geq 0, 1740 + 2460n),$ $(n < 0, 1740)\}$
ex7	Jampack/ Inv	$\{(2 > n \wedge n \geq 0, 13540 + 6420n),$ $(0 > n, 13380),$ $(n > 2, 13380 - 3100n + 9480n^2)\}$
ex8	Jampack/ Zsvd	$\{(nc \leq nr \wedge r \geq c, 3840),$ $(nc > nr \wedge c > r > b, 18260 + 18820(r - b)),$ $(nc \leq nr \wedge c < r, 3920)\}$ ...
ex9	JAMA/ Cholesky-Decomposition	$\{(1 = n, 14260 + 27420n + 3200n^2),$ $(1 > n, 14260),$ $(n > 1, 14260 + 15447n + 13419n^2 + 1754n^3)\}$
ex10	JAMA/ Eigenvalue-	$\{(1 > n, 11780),$ $(n \geq 1, -11784 + 17602n - 5146n^2 + 11108n^3)\}$
ex11	Jampack / Eigenvalue Decomposition	$\{(n < 0, 25460),$ $(n \geq 0, 25460 + 28400n + 9500n^2+$ $+11220n^3)\}$

Table 8.1: Parametric WCET computation using IBART.

IBART results. Our results are summarized in Table 8.1[Pg. 165]. Column 2 lists the source of the example. We denote by `ex1` our running example from Figure 8.1[Pg. 148]. Column 3 shows the parametric WCET calculated by IBART, using the solution language of Section 8.3[Pg. 151]. In all cases, the number of refinements needed to obtain the WCET result was between 2 and 6.

Comparison with WCET tools. We compared the precision of IBART to r-TuBound [113] supporting the Infineon C167 processor. This was the only possible direct comparison, as the other tools do not support that processor. We chose C167 as the target platform, due to its comparably simple architecture, while it is still deployed in real-world applications. Note that r-TuBound can only report a single numeric value as a WCET estimate. Therefore, to allow a fair comparison of the WCET results, symbolic parameters in the flow facts need to be instantiated with concrete values when analyzing the WCET with r-TuBound. When comparing IBART with r-TuBound, we hence instantiated our parametric WCET (from Table 8.1[Pg. 165]) with the values used in r-TuBound.

Our results, summarized in Table 8.2[Pg. 166], show that IBART provides significantly better WCET estimates. For larger values of parameters, these differences increase rapidly, as shown by `ex2`. This is because r-Tubound over-approximates each iteration much more than IBART, so if the number of iterations increases, the difference grows. The first two columns of Table 8.2[Pg. 166] describe the source of the examples, similarly to Table 8.1[Pg. 165]. Column 3

Ex	Source/File	Parameter assignments	IBART	r-TuBound
`ex1`	Section 2	$n = 5$	**22300**	26060
		$n = 100$	**388040**	480160
`ex2`	Debie/	$n = 10$	**62020**	124920
	health	$n = 50$	**298420**	612920
		$n = 200$	**1184920**	2442920
`ex3`	Mälardalen/	$n = 6, dlt = 0$	**19260**	345040
	adpcm	$n = 0, dlt \neq 0$	**4180**	4260
		$n = 0, dlt = 0$	4260	4260
`ex4`	Mälardalen/	$len = 5, jrev < 0$	24860	24860
	crc	$len = 5, jrev \geq 0$	**21220**	24860
		$len = 0, jrev \geq 0$	**4320**	5560
`ex5`	Mälardalen/	$init = 0, len = 255$	987000	987000
	crc	$init = 1, len = 255$	**2920**	987000
`ex6`	Mälardalen/	$n = 10$	**21660**	26340
	lcdnum	$n = 5$	13960	13960
`ex7`	Jampack/	$n = 5$	**243880**	519280
	Inv	$n = 1$	**19760**	19960
`ex8`	Jampack/	$r = 4, c = 5,$	**93540**	100480
	Zsvd	$nr < nc, b = 0$		
`ex9`	Jampack/	$n = 5$	**646220**	1545760
	Cholesky-	$n = 1$	44880	44880
`ex10`	JAMA/	$n = 5$	**1335920**	2606180
	Eigenvalue	$n = 0$	11620	11620
`ex11`	Jampack/	$n = 5$	1799620	1799620
	Eigen-Decomp.	$n = 1$	74580	74580

Table 8.2: WCET comparisons.

lists the value assignments of parameters. Columns 4 and 5 show respectively the WCET computed by IBART and r-TuBound.

Summary. We addressed the problem of computing precise parametric WCET estimates of programs. The two main technical contributions are: (a) the hierarchical parametric graphs (HPGs) and and the reduction of the problem of parametric WCET computation to a maximum-weight length-constraint optimization problem over paths of HPGs; and (b) a novel interpolation-based approach for refining of segment-based abstractions. This is used to refine the parametric WCET of the program. We evaluated our method on On the practical side, the contribution of the chapter comes with the implementation and evaluation of our method on examples taken from WCET benchmark suites and open-source linear algebra packages. When compared to existing WCET tools, our experiments show that our method improves the state-of-the-art in WCET computation.

Chapter 9

Battery Transition Systems

The analysis of the energy consumption of software is an important goal for quantitative formal methods. Current methods, using weighted transition systems or energy games, model the energy source as an ideal resource whose status is characterized by one number, namely the amount of remaining energy. Real batteries, however, exhibit behaviors that can deviate substantially from an ideal energy resource. Based on a discretization of a standard continuous battery model, we introduce *battery transition systems*. In this model, a battery is viewed as consisting of two parts – the available-charge tank and the bound-charge tank. Any charge or discharge is applied to the available-charge tank. Over time, the energy from each tank diffuses to the other tank.

Battery transition systems are infinite state systems that, being not well-structured, fall into no decidable class that is known to us. Nonetheless, we are able to prove that the ω-regular model-checking problem is decidable for battery transition systems. We also present a case study on the verification of control programs for energy-constrained semi-autonomous robots.

9.1 Motivation

Systems with limited energy resources, such as mobile devices or electric cars, have become ubiquitous in everyday life. In accordance, there is a growing attention to the formal modeling of such systems and the analysis of their behavior. These systems are commonly modeled as weighted transition systems, where the states of the transition system represent the system configurations, the transitions represent the possible operations, and the weights on the transitions correspond to the energy consumed (negative value) or added (positive value) during the operation. In recent literature (for example, [44, 47, 118]), weighted transition systems have been analyzed with respect to various problems, such as finite-automaton emptiness problem (starting from a given initial energy, can a specific configuration be reached while keeping the energy positive in all intermediate steps?), and Büchi emptiness problem (can a specific configuration be visited repeatedly, while keeping the energy positive?).

In all these works, the energy-resource is idealized. In particular, it is assumed that its status can be completely characterized by one number, namely the amount of remaining energy. However, physical systems with energy re-

strictions often use batteries, which are far from an ideal-energy source. One such non-ideal behavior of a battery behavior is the “recovery effect”, where the available energy at certain times is smaller than the sum of energies consumed and charged. Intuitively, the recovery effect is a result of the fact that energy is consumed from the edge of the battery, while the total charge is spread across the entire battery. When the consumption is high, additional time may be required until the charge diffuses from the inside of the battery to its edge, during which period there is no available energy, possibly failing the required operation.

The recovery effect is often noticed in our daily usage of battery-powered systems, for example mobile phones – a phone might shutdown due to an “out of power” condition, but then become live again after an idle period.

We aim to formally model such energy systems with non-ideal resources. We define a “battery transition system” (BTS), where the system is viewed as a weighted transition system, as is standard. However, the semantics of its possible traces is specified differently, to capture non-ideal behaviors. The semantics we specify for BTSs correspond to a discretization of a well-known battery model – the kinetic battery model (KiBaM) [123]. There are various battery models in the literature, admitting various accuracies and complexities, among which the KiBaM model is a good choice for the purpose of properly analyzing systems with the recovery effect [107]. We elaborate, in Section 9.2[Pg. 171], on various battery models, and explain the derivation of BTS semantics from the KiBaM model.

Semantics. The status of the battery in a BTS is a pair (x, y), where x represents the *available charge* (available for immediate usage) and y the *bound charge* (internal charge in a battery that is not immediately available). During each transition, some amount of charge diffuses between x and y. The weight of the transition (say w) affects only x in the current step. The diffusion rate depends on the difference between x and y, and on two constants of the battery: a *width constant* $c \in \mathbb{R}$ with $0 < c < 1$, and a *diffusion constant* $k \in \mathbb{R}$ with $0 < k < c(1-c)$. Formally, taking a transition of weight w from a battery status (x, y) results in the battery status (x', y'), where $x' = x - k \cdot (\frac{x}{c} - \frac{y}{1-c}) + w$ and $y' = y + k \cdot (\frac{x}{c} - \frac{y}{1-c})$. The value $k \cdot (\frac{x}{c} - \frac{y}{1-c})$ represents the amount of charge diffused between x and y. The above transition is legal if the available charge x remains positive after the transition.

The mathematical properties of a BTS are shown to be inherently different from those of a simple-energy transition system (where only the value of $x + y$ is considered), as illustrated in Fig. 9.1[Pg. 170]: Considering the system $\mathcal{B}_1$ as a simple-energy system, where only the total energy should remain positive, s_1 is directly reachable from s_0, and the cycle $s_0 \to s_2 \to s_3 \to s_0$ cannot aid in any way as it has a negative total weight. On the other hand, viewing $\mathcal{B}_1$ as a BTS, in order to go from state s_0 to s_1, counter-intuitively, a legal trace must first take the cycle through s_2 and s_3. Though decreasing the total energy, the cycle temporarily increases the available energy, allowing the transition to s_1. Furthermore, it is known that a simple-energy system (even with multi-dimension energies) admits an illegal trace if and only if it admits a memoryless illegal trace (always making the same choice at each state) [47]. However, an illegal trace in the system $\mathcal{B}_2$ of Fig. 9.1[Pg. 170] must make different choices at

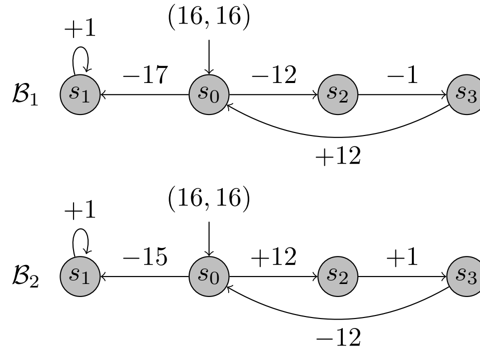


Figure 9.1: In the battery system $\mathcal{B}_1$, a trace reaching the state s_1 must make a cycle with total negative energy. In $\mathcal{B}_2$, an illegal trace must make different choices at different visits in s_0. Here, we have the diffusion constant $k = \frac{1}{8}$ and the width constant $c = \frac{1}{2}$.

different visits in state s_0 (Theorem 9.3[Pg. 177]).

Model checking. We consider the finite-automaton, Büchi, and Streett emptiness problems for a BTS; these problems are central to the model checking of systems with no fairness constraints, weak fairness constraints, and strong fairness constraints, respectively.

As BTSs are infinite-state systems, it is natural to ask if they fall into a known, tractable, class of infinite state systems. For example, standard model-checking algorithms exist for well-structured transition systems (e.g., lossy channel systems, timed automata [6], etc), where a well-quasi ordering can be defined on the states of the infinite systems, and this ordering is compatible with the transition relation of the system. However, for BTSs, we can show that the infinite sequence $(1, 1), (\frac{3}{4}, 1\frac{1}{4}), (\frac{11}{16}, 1\frac{5}{16}), (\frac{43}{64}, 1\frac{21}{64}), \ldots$ of battery statuses is monotonically decreasing with respect to any ordering that is compatible with transitions. Intuitively, this sequence contains battery statuses that have equal total charge, but strictly decreasing available charge. This implies that model-checking algorithms from the domain of well-structured transitions systems do not apply directly to BTS.

We solve the finite-automaton emptiness problem by building a forward reachability tree, along the lines of the Karp-Miller tree for Petri nets [110]. There, using the well-structured properties of Petri-nets, the Karp-Miller tree is shown to be a finite summarization of all reachable states, despite there being infinitely-many reachable states. A BTS is not well-structured, yet we are able to generate a finite "summary tree", having all the reachability data, by proving the following key observations: 1. Once the total energy in a battery status is high-enough, the problem can be reduced to simple-energy reachability; 2. Considering some characteristic properties of battery statuses allows us to define a simulation-compatible total ordering between statuses having the same total energy; and 3. Repeating a cycle whose total energy sums to 0 makes the battery status converge monotonically to a limit value independent of the initial status. Despite the fact that the above ordering is not well-founded, i.e., there may be infinite chains, the last observation lets us take limits of infinite chains

while constructing the reachability tree.

In simple-energy systems, extending the finite automaton emptiness algorithm to a Büchi emptiness algorithm is straightforward – checking whether there is a reachable Büchi state that has a cycle back to itself, such that the sum of weights on the cycle is non-negative. In a BTS, such a simple solution does not work – a cycle that does not decrease the total energy might still fail the process after finitely many iterations, as the available charge can slightly decreases on every iteration. A simple modification, seeking cycles that do not decrease both the total and the available energies is too restrictive, as the available charge may still converge to a positive value. We solve the Büchi emptiness problem by showing that if there exists an accepted trace, there also exists a lasso-shaped accepted trace having one of two special forms. These forms concern the way that the available charge changes along the cycle. By a delicate analysis of the reachability tree, we then solve the question of whether the transition system allows for a trace in one of these special forms. The Streett emptiness problem is solved similarly, by using a small extension of the Büchi emptiness algorithm.

We show that our algorithms for the finite-automaton, Büchi, and Streett emptiness problems are in PSPACE with respect to the number of states in the transition system and a unary representation of the weights. If weights are represented in binary, or if the battery constants are arbitrarily small and represented in binary, the space complexity grows exponentially.

Case study: Robot control. We examine a semi-autonomous robot control in an energy-constrained environment. We present a small programming language for robot-controllers and define quantitative battery-based semantics for controllers written in that language. We solve the ω-regular model-checking problem for programs written in this language, using our results on battery transition systems. We demonstrate the inadequacy of standard quantitative verification techniques, where the battery is viewed as an ideal energy resource – they might affirm, for example, that the robot can reach some target locations, while taking into account the non-ideal behavior of its battery, it cannot.

9.2 Battery Models

We provide a short description of how batteries are modeled in the literature, and explain how we derive our formal model of a battery.

A battery consists of one or more electro-chemical cells, each of which contain a negative electrode (anode), a positive electrode (cathode), and a separator between them. During discharge and recharge, electrons move through the external circuit, while chemical reaction produces or consumes chemical energy inside the battery. For example, during discharge in lithium-ion (Li-ion) batteries, positive lithium ions move from the anode to the cathode, while the reverse occurs during recharge (see Fig. 9.2[Pg. 172]).

Batteries of all types have a "recovery effect", meaning that the chemical reaction inside the battery does not keep up with the rate of the external activity. Internally, in Li-ion batteries, the concentration of the electro-active species near the electrodes becomes smaller than their concentration in the interior of the battery. When the battery has low load for some time, the ions have enough

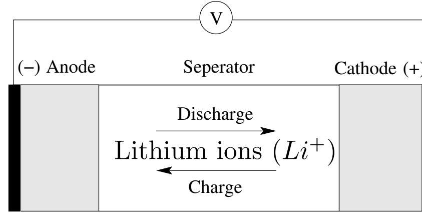


Figure 9.2: Schematic of a lithium-ion (Li-ion) battery.

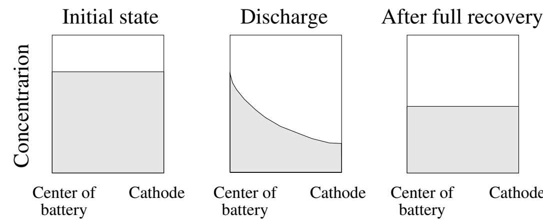


Figure 9.3: Concentration of electro-active species along the battery, following a discharge and a recovery phase.

time to diffuse to the electrodes, and charge recovery takes place (see Fig. 9.3[Pg. 172]). The well-known symptom of this is that a battery might be "empty" after some usage, but then becomes "charged" after an idle period.

There are many battery models modeling various aspects of a real battery. The most accurate ones model the electro-chemical reactions in detail [27, 76, 87, 137]. Though highly accurate, they require configuration of many (usually around 50) parameters, making them difficult to analyze. Another approach taken is to model the electrical properties of the battery using voltage sources, resistors, and other elements [89, 92]. These approximate battery voltage behavior well, but their modeling of the available battery capacity is inaccurate. A third class consists of the analytical models that describe the battery at a high abstraction level, modeling only its major properties by means of a few equations. The dominant models of this class are the kinetic battery model [123], and the diffusion model [141]. A detailed description of the various models can be found in [106, 107].

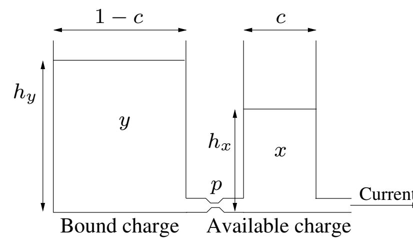


Figure 9.4: Kinetic battery model (KiBaM).

The possibly simplest, yet useful, model that handles the recovery effect is

the kinetic battery model (KiBaM) [123]. While being originally developed for Lead-Acid batteries to model both battery capacity and battery voltage, its capacity modeling was found to be a good approximation even for more modern batteries such as the Li-ion battery. In [106, 107], it was theoretically shown that KiBaM is a first order approximation of the diffusion model, which was designed for Li-ion batteries. In addition, their experimental results show that it has up to 7 percent deviation from the accurate electro-chemical models.

We concentrate on a battery's available capacity, and hence, adopt the KiBaM model. In this model, the battery charge is distributed over two tanks: the available-charge tank, denoted x, of width $c \in (0,1)$, and the bound-charge tank, denoted y, of width $1-c$ (see Fig. 9.4[Pg. 172]). The external current gets electrons only from the available-charge tank, whereas electrons from the bound-charge tank flow to the available-charge tank. When recharging, the reverse process occurs, electrons are added directly to the available-charge tank, from there they flows to the bound-charge tank. The charge flows between the tanks through a "valve" with a fixed conductance p. The parameter p has the dimension 1/time and influences the rate at which the charge can flow between the two tanks. This rate is also proportional to the height difference between the two tanks. If the heights are given by $h_x = x/c$ and $h_y = y/(1-c)$, and the current load by $w(t)$, the charge in the tank over time behaves according to the following system of differential equations [123]:

$$\frac{dx}{dt} = -w(t) - p(h_x - h_y); \qquad \frac{dy}{dt} = p(h_x - h_y) \tag{9.1}$$

with initial conditions $x(0) = c \cdot C$ and $y(0) = (1-c) \cdot C$, where C is the total battery capacity. The battery cannot supply charge when there is no charge left in the available-charge tank.

We are interested in calculating the battery status along a discrete-time transition system, thus consider the equations (9.1[Pg. 173]) for fixed time steps. We get the following equations:

$$x_{i+1} = x_i - w_i - k(h_{x_i} - h_{y_i}); \qquad y_{i+1} = y_i + k(h_{x_i} - h_{y_i}) \tag{9.2}$$

where x_i and y_i are the values of x and y before the time step i, respectively, w_i is the total load on the battery at time step i, and $k = p \times$ (length of a time step). The smaller the time steps are, the smaller k is, and the more accurate the discretization is.

We further need to ensure that the discretization does not introduce undesirable behaviours. In Eq. 9.1[Pg. 173] if $h_x > h_y$ and $w(t)$ is 0, the relation $h_x > h_y$ keeps holding. We should ensure this in the discrete model, i.e., if $h_{x_i} > h_{y_i}$ and $w_i = 0$, then it cannot be that $h_{x_{i+1}} \leq h_{y_{i+1}}$.

Formalizing the above requirement, we have

$$h_{x_{i+1}} = \frac{x_{i+1}}{c} = \frac{x_i - 0 - k(h_{x_i} - h_{y_i})}{c} = h_{x_i} - \frac{k(h_{x_i} - h_{y_i})}{c};$$
$$h_{y_{i+1}} = \frac{y_{i+1}}{1-c} = \frac{y_i + k(h_{x_i} - h_{y_i})}{1-c} = h_{y_i} + \frac{k(h_{x_i} - h_{y_i})}{1-c}.$$

$$\text{Hence,} \quad h_{x_{i+1}} - h_{y_{i+1}} = (h_{x_i} - h_{y_i})(1 - k(\frac{1}{c(1-c)})).$$

Therefore, the parameter k is acceptable if $k(\frac{1}{c(1-c)}) < 1$, leading to the conclusion that

$$k < c(1-c) \tag{9.3}$$

9.3 Battery Transition Systems

We incorporate the discrete battery model from Equation 9.2[Pg. 173] into a weighted transition system. The system consists of finitely many control-states and weighted transitions between them, where the weights denote the amount of energy recharged/consumed at each operation.

9.3.1 Weighted Transition Systems and Battery Semantics

A *unlabelled transition system* is a tuple $\langle S, \Delta, s_\iota \rangle$ where S and s_ι are a set of states and an initial state as in a labelled transitions system. The set $\Delta \subseteq S \times S$ is a set of transitions with no labels. Similarly, a *unlabelled weighted transition system* (UWTS) is a tuple $\mathcal{S} = \langle S, \Delta, s_\iota, v \rangle$ where $\langle S, \Delta, s_\iota \rangle$ is an unlabelled transition system, and $v : \Delta \to \mathbb{Z}$ is a weight function labeling transitions with integer weights[1].

A *battery transition system* (BTS or *battery system*, for short) is a tuple $\mathcal{B} = \langle \langle S, \Delta, s_\iota, v \rangle, c, k \rangle$ where $\langle S, \Delta, s_\iota, v \rangle$ is a UWTS with a finite number of control states (i.e., $|S| < \infty$), $c \in \mathbb{R}$ is a *width constant* with $0 < c < 1$, and $k \in \mathbb{R}$ is a *diffusion constant* with $0 < k < c(1-c)$.

Semantics. Given a BTS $\mathcal{B} = \langle \mathcal{S}, c, k \rangle$, a *battery status* $(x, y) \in \mathbb{R}^2_{>0}$ represents the current configuration of the battery. Intuitively, the values x and y represent the charge in the available-charge tank and the bound-charge tank of the battery, respectively (see Figure 9.4[Pg. 172]).

If the current battery status is (x, y), on a transition of weight w, we define the change in the battery status using a function $\texttt{Post} : \mathbb{R}^2 \times \mathbb{Z} \to \mathbb{R}^2$, letting the battery status after the transition $(x', y') = \texttt{Post}((x, y), w)$. Here, we follow the standard convention of energy transition systems and consider a positive (resp. negative) weight as adding (resp. drawing) a charge to (resp. from) the battery. In matrix notation, we have the following.

$$\texttt{Post}((x, y), w) = \left[\mathcal{A}_\mathcal{B} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \right]^\top + \begin{pmatrix} w \\ 0 \end{pmatrix}^\top, \text{ where}$$

$$\mathcal{A}_\mathcal{B} = \begin{pmatrix} 1 - \frac{k}{c} & \frac{k}{1-c} \\ \frac{k}{c} & 1 - \frac{k}{1-c} \end{pmatrix}$$

The matrix $\mathcal{A}_\mathcal{B}$ is called the *diffusion matrix*. Note that the $\texttt{Post}$ function indeed follows Equation 9.2[Pg. 173] except for the change in the sign of w. The values $\frac{k}{c} \cdot x$ and $\frac{k}{1-c} \cdot y$ denote the heights h_x and h_y of the two tanks, and hence, we get that $x' = x - k \cdot (h_x - h_y) + w$ and $y' = y + k \cdot (h_x - h_y)$. We abuse notation by

[1] All of our results equally hold for rational weights, by simply multiplying all weights by the product of all denominators.

defining $\texttt{Post}(\mathbf{t}, w_1 w_2 \ldots w_m)$ inductively as $\texttt{Post}(\texttt{Post}(\mathbf{t}, w_1), w_2 \ldots w_m)$ where each $w_i \in \mathbb{Z}$.

Given an *initial battery status* $\mathbf{t}_\iota \in \mathbb{R}^2_{>0}$, the *semantics* of a BTS $\mathcal{B} = \langle\langle S, \Delta, s_\iota, v\rangle, c, k\rangle$ is given by a (possibly infinite) transition system $\langle E, \rightarrow, e_\iota\rangle$ where $E = S \times \mathbb{R}^2_{>0}$ is the set of states, $\rightarrow \subseteq E \times E$ is the transition relation, and $e_\iota = (s_\iota, \mathbf{t}_\iota)$ is the initial state.

- We call each $(s, \mathbf{t}) \in E$ an *extended state* with $s \in S$ being its control state, and $\mathbf{t} \in \mathbb{R}^2_{>0}$ being its battery status [2].
- We have $((s, \mathbf{t}), (s', \mathbf{t}')) \in \rightarrow$ if and only if $\texttt{Post}(\mathbf{t}, w) = \mathbf{t}'$ and $(s, s') \in \Delta \wedge v((s, s')) = w$. We write $(s, \mathbf{t}) \rightarrow (s', \mathbf{t}')$ instead of $((s, \mathbf{t}), (s', \mathbf{t}')) \in \rightarrow$.

A weight w (and by extension, a transition with weight w) is *feasible* from battery status $\mathbf{t}$ if $\texttt{Post}(\mathbf{t}, w) \in \mathbb{R}^2_{>0}$, namely if $\texttt{Post}(\mathbf{t}, w)$ is a valid battery status. Similarly, a sequence of weights $w_0 w_1 \ldots w_n$ is feasible from $\mathbf{t}$ iff w_0 is feasible from $\mathbf{t}$ and each w_i is feasible from $\texttt{Post}(\mathbf{t}, w_0 \ldots w_{i-1})$. Extending the nomenclature, we call every $\mathbf{t} \in \mathbb{R}^2 \setminus \mathbb{R}^2_{>0}$ *infeasible*.

The *traces* of a BTS $\mathcal{B}$, denoted $\Pi(\mathcal{B})$, are given by (infinite or finite) paths of the form $\pi = (s_0, \mathbf{t_0})(s_1, \mathbf{t_1}) \ldots$ where $s_0 = s_\iota$ and $\mathbf{t_0} = \mathbf{t}_\iota$ and for every $i \geq 1$, we have $(s_{i-1}, \mathbf{t_{i-1}}) \rightarrow (s_i, \mathbf{t_i})$. The corresponding *control trace* is given by $\theta = \texttt{control}(\pi) = s_0 s_1 \ldots$, and the set of control traces by $\Theta(\mathcal{B}) = \{\texttt{control}(\pi) \mid \pi \in \Pi(\mathcal{B})\}$. We say that a (finite or infinite) sequence of control states $s_0 s_1 \ldots$ is feasible from a battery status $\mathbf{t}$ iff the weight sequence $w_0 w_1 \ldots$ is feasible from $\mathbf{t}$, where $(s_i, s_{i+1}) \in \Delta \wedge v((s_i, s_{i+1})) = w_i$.

Energy feasibility. We define an alternate set of semantics corresponding to the classical notion of ideal-energy systems. We say that $(x, y) \in \mathbb{R}^2$ is *energy-feasible* if $x + y > 0$. As for the term "feasible", we further extend the notion of energy-feasible as follows: 1. A sequence of weights $w_0 \ldots w_n$ is energy-feasible from (x, y) iff $\texttt{Post}((x, y), w_0 \ldots w_i)$ is energy-feasible for all $0 \leq i \leq n$; and 2. A sequence of control states $s_0 s_1 \ldots$ is energy-feasible from (x, y) iff $w_0 w_1 \ldots$ is energy-feasible from (x, y) where $w_i = v(s_i, s_{i+1})$.

Characteristic functions. For mathematical simplicity, we follow the approach taken in [123] and use an alternate representation for battery statuses. We represent (x, y) using two other numbers, denoting it $[\texttt{e}; \texttt{d}]$, where $\texttt{e}$ and $\texttt{d}$ are defined by the following *energy* and *deviation* functions.

- $\texttt{e} = \texttt{energy}((x, y)) = x + y$; and
- $\texttt{d} = \texttt{deviation}((x, y)) = x - y \cdot \dfrac{c}{1 - c}$.

Intuitively, $\texttt{e}$ is the total energy in the battery and $\texttt{d}$ is the difference between the heights of the two tanks multiplied by the factor c. The mathematical simplicity in using *energy* and *deviation* stems from the fact that they correspond to the eigenvectors of the diffusion matrix. Given $\texttt{e} = \texttt{energy}(\mathbf{t})$ and $\texttt{d} = \texttt{deviation}(\mathbf{t})$, the battery status $\mathbf{t} = (x, y)$ is uniquely determined: $x = c\texttt{e} + (1 - c)\texttt{d}$ and $y = \texttt{e} - x$. Hence, we use the notations $[\texttt{e}; \texttt{d}]$ and (x, y) interchangeably.

Proposition 9.1. *For any battery status* $[\texttt{e}; \texttt{d}]$ *and* $w \in \mathbb{N}$, *we have that* $\texttt{Post}([\texttt{e}; \texttt{d}], w) = [\texttt{e} + w; \lambda \cdot \texttt{d} + w]$ *where* $\lambda = 1 - \frac{k}{1-c} - \frac{k}{c}$.

[2] All of our results equally hold for the case that the element values should be non-negative, rather than strictly positive, but the proofs are marginally simpler in the strictly positive case.

The above proposition is a translation of the $\texttt{Post}$ function to the $[\mathtt{e};\mathtt{d}]$ notation. Intuitively, the energy is increased by the weight w as expected, while the difference in the tank heights is first reduced by a constant factor of λ and then increased due to the charge w added to the first column. The factor λ turns out to be the central parameter of the battery, playing a key role in how BTSs behaves. The following lemma formalizes the intuition that the bound-charge tank (y) cannot get empty before the available-charge tank (x) does.

Lemma 9.2. *Suppose battery status* (x, y) *is feasible, and let* $\texttt{Post}((x, y), w) = (x', y')$. *We have* $[\mathtt{e}'; \mathtt{d}'] = (x', y')$ *is feasible iff* $x' > 0$, *and, equivalently, if and only if* $c\mathtt{e}' + (1 - c)\mathtt{d}' > 0$.

Proof. The *only if* implication is obvious. As for the *if*, we have $x' = (1 - \frac{k}{c})x + \frac{k}{1-c}y + w$ and $y' = \frac{k}{c}x + (1 - \frac{k}{1-c})y$. Assuming $x > 0$ and $y > 0$, it easily follows that $y' > 0$. Hence, (x', y') is infeasible if and only if $x' \leq 0$ or equivalently, if $x' = c\mathtt{e}' + (1 - c)\mathtt{d}' \leq 0$ □

Model checking. The problems we consider ask for the existence of a control trace θ in the semantics of a BTS $\mathcal{B}$ with control states S given an initial battery status $\mathbf{t}$, such that $\theta \in \Phi$ for some given *objective set* $\Phi \subseteq S^* \cup S^\omega$. Specifically, we consider the following objective sets Φ:

- **Finite-automaton emptiness.** Asking if there exists a feasible trace to a set of target control states. Formally, given target control states $T \subseteq S$, we have $\Phi = \mathit{Reach}(T) = \{s_0 s_1 \ldots \mid \exists i : s_i \in T\}$, i.e., Φ is the set of control traces which visit T at least once.
- **Büchi emptiness.** Asking if there exists a feasible trace which visits a set of target Büchi states infinitely often. Formally, given Büchi control states $L \subseteq S$, we have $\Phi = \text{Büchi}(L) = \{s_0 s_1 \ldots \mid \forall j \exists i > j : s_i \in L\}$.
- **Streett emptiness.** The objective is specified by a set of request-grant pairs $\langle R_i, G_i \rangle$ (where each pair consists of a set $R_i \subseteq S$ of *request* control states and a set G_i of *grant* control states). The objective asks if there exists a feasible trace in the system such that for every request-grant pair, either G_i is visited infinitely often or R_i is visited finitely often. Formally, given a set of Streett pairs $P = \{\langle R_0, G_0 \rangle, \langle R_1, G_1 \rangle, \ldots, \langle R_m, G_m \rangle\}$, we have $\Phi = \texttt{Streett}(P) = \{s_0 s_1 \ldots \mid \forall 0 \leq i \leq m : [(\forall p \exists q > p : s_q \in G_i) \vee (\exists p \forall q > p : s_q \notin R_i)]\}$.

We call the traces which satisfy the finite-automaton, Büchi, and Streett conditions, accepting, Büchi, and Streett traces, respectively.

9.3.2 (Battery VS. Ideal-Energy) Transition Systems

Due to the recovery effect, BTSs behave qualitatively differently from a simple-energy transition systems. Nevertheless, in the domain where the energy in the battery is high, they do behave similarly. This lets us solve problems related to unlimited initial credit (referred to as "unknown initial credit") by reducing them to the simple-energy system problems.

Different Behavior. The BTS $\mathcal{B}_1$ of Fig. 9.1[Pg. 170] demonstrates a key difference from a simple-energy system – the total energy in the initial state is 32, while a transition of weight (-17) cannot be taken, since the available

energy is only 16. Yet, taking the cycle through states s_2 and s_3 reduces the total energy, but allows the (-17)-transition. After the cycle, the battery status is $(19\frac{3}{4}, 11\frac{1}{4})$, which becomes $(\frac{5}{8}, 13\frac{3}{8})$ following a (-17)-transition.

We formalize the difference in the theorem below. It is known that if an ideal-energy system contains an infeasible trace, it contains a memoryless infeasible trace [47]. A memoryless trace is one where a control state is always followed by the same control state. However, an infeasible trace in the system $\mathcal{B}_2$ of Fig. 9.1[Pg. 170] must make different choices at different visits in state s_0.

Theorem 9.3. *A battery transition system may have feasible (resp. infeasible) traces without having any memoryless feasible (resp. infeasible) traces.*

Proof. Consider the BTS $\mathcal{B}_1$ in Figure 9.1[Pg. 170]. There is a trace for reaching s_1, as well as an infinite trace, however both traces make non-uniform choices at different visits in s_0. Analogously, an infeasible trace in the system $\mathcal{B}_2$ of Figure 9.1[Pg. 170] must make different choices at s_0. We prove below the claim for $\mathcal{B}_2$. Analogous arguments and calculations can be made with respect to $\mathcal{B}_1$.

The only nondeterminism in $\mathcal{B}_2$ is in state s_0, allowing transitions to states s_1 and s_2. A trace that first chooses s_1 is legal, since the (-15)-transition is feasible from the initial status, after which there are only positive-weight transitions. The other memoryless option, of always choosing s_2, is also legal: Let (x, y) be the battery status when first reaching s_2. It can be shown that the first few cycles $s_2 \to s_3 \to s_0 \to s_2$ are feasible, after which the battery status will be (x', y'), such that $x' > x$ and $y' > y$. By the monotonicity of the `Post` function, it follows that the cycle can be repeated forever.

On the other hand, first choosing s_2 and then choosing s_1 makes an illegal trace: The battery status when returning to s_0 is $(12\frac{3}{4}, 20\frac{1}{4})$, which changes to $(-\frac{5}{8}, 18\frac{3}{8})$ after the (-15)-transition. □

High energy domain and unknown initial credit problems. In energy systems, one often considers "unknown-initial-credit problems", asking if there is some initial energy that allows accomplishing a task. It is clear that every control state of a battery system can be reached, at least once, if there is a path leading to it and enough initial energy to start with. This is formalized in the following lemma (which will also serve us in Theorem 9.5[Pg. 178] and in Section 9.4[Pg. 179]).

Lemma 9.4. *Consider a BTS $\mathcal{B} = \langle\langle S, \Delta, s_\iota, v\rangle, k, c\rangle$. There exist constants* $\texttt{HighEnergyConstant}(\mathcal{B}, i)$ *for every $i \in \mathbb{N}$ such that for every feasible extended state $(s_0, [\mathtt{e}; \mathtt{d}])$ with $\mathtt{e} > \texttt{HighEnergyConstant}(\mathcal{B}, i)$, every weight sequence $w_0 w_1 \ldots w_{i-1}$ of length i is feasible from $(s_0, [\mathtt{e}; \mathtt{d}])$.*

Proof. We define the constants inductively as follows:

(a) $\texttt{HighEnergyConstant}(\mathcal{B}, 0) = 0$ if $i = 0$; and

(b) $\texttt{HighEnergyConstant}(\mathcal{B}, i) =$
$\max\left(\texttt{HighEnergyConstant}(\mathcal{B}, i-1) + W, \frac{W}{c(1-\lambda)}\right)$ otherwise. Here, $W = \max_{(s,s') \in \Delta} |v((s, s'))|$.

We prove the theorem by induction. When $i = 0$, the weight sequence is empty and there is nothing to prove.

For the induction case, assume that we have shown the result up to $i-1$. Let $\texttt{Post}([\mathtt{e}; \mathtt{d}], w) = [\mathtt{e}'; \mathtt{d}']$. From Proposition 9.1[Pg. 175], we get that $\mathtt{e}' = \mathtt{e} + w$

and $\mathtt{d}' = \lambda \mathtt{d} + w$. If $[\mathtt{e}'; \mathtt{d}']$ is feasible and $\mathtt{e}' \geq \mathtt{HighEnergyConstant}(\mathcal{B}, i-1)$, we can apply the induction hypothesis on the weight sequence $w_1 \ldots w_{i-1}$ to prove the result. We show these facts below.

- We have $\mathtt{e}' = \mathtt{e} + w > \mathtt{HighEnergyConstant}(\mathcal{B}, i) + w \geq \mathtt{HighEnergyConstant}(\mathcal{B}, i-1) + W + w$. As $w \geq -W$, we have that $\mathtt{e}' > \mathtt{HighEnergyConstant}(\mathcal{B}, i-1) \geq 0$.
- Further, we have that $c\mathtt{e}' + (1-c)\mathtt{d}' = c\mathtt{e} + cw + (1-c)\lambda \mathtt{d} + (1-c)w$ or, equivalently, $c\mathtt{e}' + (1-c)\mathtt{d}' = (1-\lambda)c\mathtt{e} + w + \lambda(c\mathtt{e} + (1-c)\mathtt{d})$. As $[\mathtt{e}; \mathtt{d}]$ is feasible, by Proposition 9.1[Pg. 175], $c\mathtt{e} + (1-c)\mathtt{d} > 0$. Further, $\mathtt{e} > \mathtt{HighEnergyConstant}(\mathcal{B}, i) \geq \frac{W}{c(1-\lambda)}$. Using these, we get $c\mathtt{e}' + (1-c)\mathtt{d}' > -W + w \geq 0$. By Lemma 9.2[Pg. 176], $[\mathtt{e}'; \mathtt{d}']$ is feasible, completing the proof.

□

The following theorem states that the emptiness problems for battery systems reduce to the corresponding problems for energy systems if the initial energy is large enough.

Theorem 9.5. *Let $\mathcal{B} = \langle\langle S, \Delta, s_\iota, v\rangle, k, c\rangle$ be a BTS, $W = \max_{(s,s')\in\Delta} |v((s,s'))|$, and T, L and $\{\langle R_0, G_0\rangle, \ldots, \langle R_n, G_n\rangle\}$ be a set of target states, a set of Büchi states, and a set of Streett pairs, respectively. There exist constants $M_R = \mathtt{HighEnergyConstant}(\mathcal{B}, |S|)$, $M_B = \mathtt{HighEnergyConstant}(\mathcal{B}, 3|S| + 2W|S|^2)$, and $M_S = \mathtt{HighEnergyConstant}(\mathcal{B}, |S| + |S|^2 + W|S|^2 + W|S|^3)$ such that for any extended state $(s, [\mathtt{e}; \mathtt{d}])$: if $\mathtt{e} > M_R$ (resp. $\mathtt{e} > M_B$ and $\mathtt{e} > M_S$), a feasible accepting (resp. Büchi and Streett) trace starting from $(s, [\mathtt{e}; \mathtt{d}])$ exists iff an energy-feasible accepting (resp. Büchi and Streett) trace exists.*

Proof. In all three parts, the existence of an energy feasible trace is directly implied by the existence of a feasible trace. Therefore, we only deal with showing that the existence of an energy feasible trace implies the existence of a feasible trace.

The first part (finite-automaton emptiness) follows directly from Lemma 9.4[Pg. 177]. If there exists a path from s to the target set T, there exists a path of length at most $|S|$. Taking $M_R = \mathtt{HighEnergyConstant}(\mathcal{B}, |S|)$ is sufficient to give us the result.

For the second part (Büchi emptiness), the existence of an energy feasible trace implies that there exists a reachable cycle which visits a Büchi state and has non-negative total weight. We show below that the length of such a cycle can be bounded by $2|S| + 2W|S|^2$. Now, if $\mathtt{e} > \mathtt{HighEnergyConstant}(\mathcal{B}, |S| + 2|S| + 2W|S|^2)$, we can show, using Lemma 9.4[Pg. 177], that there is a feasible path from $(s, [\mathtt{e}; \mathtt{d}])$ to some $(s', [\mathtt{e}'; \mathtt{d}'])$ where s' is on the cycle and $\mathtt{e}' > \mathtt{HighEnergyConstant}(\mathcal{B}, 2|S| + 2W|S|^2)$. Now, the cycle is feasible from $(s', [\mathtt{e}'; \mathtt{d}'])$ and further, on returning to s', the energy is at least $\mathtt{e}'$ (as the total weight of the cycle is non-negative). Using such reasoning, it is easy to see that the cycle is repeatedly feasible from $(s', [\mathtt{e}'; \mathtt{d}'])$. Hence, a value of $M_B = \mathtt{HighEnergyConstant}(\mathcal{B}, |S| + 2|S| + 2W|S|^2)$ is sufficient.

Now, we show that the length of such a cycle can be bounded by $2|S| + 2W|S|^2$. Let $s_0 s_1 \ldots s_n s_0$ be any such cycle of non-negative total weight and visiting a Büchi state. Assume that the cycle has positive weight. The case with the cycle having 0 weight is similar. It is easy to see that in the part of

the UWTS defined by control states of this cycle, there exists a non-negative simple cycle (say θ_l). If the Büchi node is on θ_l, we are done (as the length of a simple cycle is bounded by $|S|$). Otherwise, we can find a path from a control trace on θ_l to the Büchi state in $s_0 s_1 \ldots s_n$ and back of length at most $2|S|$. Now, the maximum negative weight that can be accumulated on this path is $2W|S|$. Therefore, combining this path with $2W|S|$ iterations of the positive cycle θ_l will give us a positive cycle visiting a Büchi state. Since the length of θ_l is bounded by $|S|$, we get that the length of the whole cycle is bounded by $2|S| + 2W|S|^2$.

The proof for the third part is very similar to the proof of the second part except for the bound on the length of the cycle.

□

A straightforward consequence of the above theorem is that the unknown initial credit problems for BTSs are decidable. Furthermore, using the above theorem, it is easy to show that these BTS problems are equivalent to the corresponding energy-systems problems, which can be solved in polynomial time [41].

Corollary 9.6. *Given a BTS $\mathcal{B}$ and a finite-automaton, Büchi, or Streett condition, the problem of whether there is an initial battery status* $[\mathtt{e};\mathtt{d}]$*, such that there exists a feasible trace in $\mathcal{B}$ satisfying the condition is decidable in polynomial time.*

9.4 The Bounded-Energy Reachability Tree

Our algorithms for solving the emptiness problems are based on representing the infinite tree of all the possible traces in the BTS in a finite tree that summarizes all required information. The construction of the tree uses a "high-energy constant" – exploration from states whose energy is above the constant is stopped, as they can be further handled by a reduction to a simple-energy system using Theorem 9.5[Pg. 178]. Hence, the tree summarizes bounded-energy reachability, and we denote it `BERT`. As in Theorem 9.5[Pg. 178], the value of the high-energy constant depends on the problem to be solved. We describe how we construct the tree, taking the high-energy constant as a parameter. We start with some basic lemmata about a total order among battery statuses of equal energy. Then, we present the 0-cycle saturation lemma, which helps summarize unbounded iterations of cycles in a finite manner.

Feasibility order for battery statuses. The following lemma shows that there exists a total order on the set of battery statuses with the same energy such that every weight sequence feasible from a lower battery status is also feasible from a higher battery status.

Lemma 9.7. *Given two battery statuses* $[\mathtt{e};\mathtt{d}]$ *and* $[\mathtt{e};\mathtt{d}']$ *with* $\mathtt{d} > \mathtt{d}'$*, every weight sequence $w_0 w_1 \ldots w_{n-1}$ feasible from* $[\mathtt{e};\mathtt{d}']$ *is also feasible from* $[\mathtt{e};\mathtt{d}]$*. Furthermore, if* $\mathtt{Post}([\mathtt{e};\mathtt{d}], w_0 \ldots w_{n-1}) = [\mathtt{e}'';\mathtt{d}'']$ *and* $\mathtt{Post}([\mathtt{e};\mathtt{d}'], w_0 \ldots w_{n-1}) = [\mathtt{e}'';\mathtt{d}''']$*, we have* $\mathtt{d}'' > \mathtt{d}'''$*.*

Proof. We prove the result by induction. For the base case, we let the weight sequence be of length 1, i.e., w_0. From Proposition 9.1[Pg. 175], we know that

$\mathtt{Post}([\mathtt{e},\mathtt{d}], w_0) = [\mathtt{e} + w_0, \lambda\mathtt{d} + w_0]$ and $\mathtt{Post}([\mathtt{e},\mathtt{d}'], w_0) = [\mathtt{e} + w_0, \lambda\mathtt{d}' + w_0]$. Hence, $\mathtt{d}'' = \lambda\mathtt{d}+w_0$ and $\mathtt{d}''' = \lambda\mathtt{d}'+w_0$. As $\mathtt{d} > \mathtt{d}'$, we get that $\lambda\mathtt{d}+w_0 > \lambda\mathtt{d}'+w_0$ and $\mathtt{d}'' > \mathtt{d}'''$.

Now, assume that $[\mathtt{e} + w_0; \lambda\mathtt{d}' + w_0]$ is feasible. By Lemma 9.2[Pg. 176], we get that $c(\mathtt{e} + w_0) + (1 - c)(\lambda\mathtt{d}' + w_0) > 0$. As $d > d'$, we have $c(\mathtt{e} + w_0) + (1 - c)(\lambda\mathtt{d} + w_0) > 0$ and $c(\mathtt{e} + w_0) + (1 - c)\mathtt{d}'' > 0$. This gives us that $[\mathtt{e} + w_0; \mathtt{d}'']$ is feasible, hence completing the proof for the base case.

Now, assume that the required result holds for every weight sequence of length $n - 1$. Applying the induction hypothesis on the battery statuses $[\mathtt{e} + w_0; \lambda\mathtt{d} + w_0]$ and $[\mathtt{e} + w_0; \lambda\mathtt{d}' + w_0]$ and the weight sequence $w_1 \dots w_{n-1}$ gives us the required result. □

Guided by Lemma 9.7[Pg. 179] above, we define a partial order on the set of battery statuses as follows: $[\mathtt{e}'; \mathtt{d}'] \sqsubseteq [\mathtt{e}; \mathtt{d}]$ if $\mathtt{e} = \mathtt{e}'$ and $\mathtt{d}' \leq \mathtt{d}$, in which case we say that $[\mathtt{e}; \mathtt{d}]$ *subsumes* $[\mathtt{e}'; \mathtt{d}']$. We extend the partial order $\sqsubseteq$ to extended states (with both control states and battery statuses) by letting $(s', [\mathtt{e}'; \mathtt{d}']) \sqsubseteq (s, [\mathtt{e}; \mathtt{d}])$ if $s = s'$ and $[\mathtt{e}'; \mathtt{d}'] \sqsubseteq [\mathtt{e}; \mathtt{d}]$. Lemma 9.7[Pg. 179] can now be restated as follows: *If* $(s', [\mathtt{e}'; \mathtt{d}']) \sqsubseteq (s, [\mathtt{e}; \mathtt{d}])$, *every control path feasible from* $(s', [\mathtt{e}'; \mathtt{d}'])$ *is also feasible from* $(s, [\mathtt{e}; \mathtt{d}])$.

Zero-cycle saturation. We formalize below the key observation that 0-energy cycles can be finitely summarized: an infinite run along such a cycle monotonically converges to a fixed battery status. Moreover, the deviation in the limit is independent of the initial status.

Lemma 9.8 (Zero-cycle saturation)**.** *Let* $w_0 \dots w_{n-1}$ *be a sequence of weights such that* $\sum_{i=0}^{n-1} w_i = 0$ *and let* $\mathbf{t_0}$, $\mathbf{t_1}$, ... *be a sequence of tuples in* $\mathbb{R}^2$, *such that* $\mathbf{t_{i+1}} = \mathtt{Post}(\mathbf{t_i}, \mathbf{w_0} \dots \mathbf{w_{n-1}})$. *We have the following:*

1. *The sequence* $\mathbf{t_0}$, $\mathbf{t_1}$, ... *converges (say to* $\mathbf{t}^* = [\mathtt{e}^*; \mathtt{d}^*]$*). In other words,* $\forall \epsilon \geq 0. \exists m \in \mathbb{N} : |\mathbf{t}^* - \mathbf{t_m}|_1 \leq \epsilon$ *where* $|.|_1$ *denotes the maximum absolute component in a vector.*
2. *We have* $\forall i \in \mathbb{N} : \mathbf{t}^* \sqsubseteq \mathbf{t_i} \sqsubseteq \mathbf{t_0}$ *or* $\forall i \in \mathbb{N} : \mathbf{t_0} \sqsubseteq \mathbf{t_i} \sqsubseteq \mathbf{t}^*$. *In the latter case, if* $w_0 \dots w_{n-1}$ *is feasible from* $\mathbf{t_0}$ *it is feasible from each* $\mathbf{t_i}$.

Proof. Let $\mathbf{t_i} = [\mathtt{e}_i; \mathtt{d}_i]$. Obviously, $\forall i. \mathtt{e}_i = \mathtt{e}^*$. By repeated application of Proposition 9.1[Pg. 175] on the weights $w_0 w_1 \dots w_{n-1}$, starting with $\mathbf{t_i}$, we have $\mathtt{d}_{i+1} = \mathtt{d}_i \cdot \lambda^n + \sum_{p=0}^{n-1} w_p \cdot \lambda^{n-1-p}$. Hence, for all $i \in \mathbb{N}$, $\mathtt{d}_i = \mathtt{d}_0 \cdot \lambda^{i \cdot n} + \sum_{q=0}^{i-1} L \cdot \lambda^{n \cdot q}$, where $L = \sum_{p=0}^{n-1} w_p \cdot \lambda^{n-1-p}$. From this, it follows that the sequence $\mathbf{d_i}$ converges to $\mathbf{d}^* = \sum_{q=0}^{\infty} L \cdot \lambda^{q \cdot n} = L \cdot \frac{1}{1-\lambda^n}$. Further,

$$
\begin{aligned}
\mathtt{d}_{(i+1)} &= \mathtt{d}_0 \cdot \lambda^{i \cdot n + n} + L \cdot \sum_{q=0}^{i} \lambda^{n \cdot q} \\
&= \mathtt{d}_0 \cdot \lambda^{i \cdot n} + L \cdot \sum_{q=0}^{i-1} \lambda^{n \cdot q} + \mathtt{d}_0 \cdot [\lambda^{i \cdot n + n} - \lambda^{i \cdot n}] + L \cdot \lambda^{i \cdot n} \\
&= \mathtt{d}_i + \mathtt{d}_0 \cdot [\lambda^{i \cdot n + n} - \lambda^{i \cdot n}] + L \cdot \lambda^{i \cdot n} \\
&= \mathtt{d}_i + \lambda^{i \cdot n}(1 - \lambda^n) \cdot [\frac{L}{1 - \lambda^n} - \mathtt{d}_0] \\
&= \mathtt{d}_i + \lambda^{i \cdot n}(1 - \lambda^n) \cdot [d^* - \mathtt{d}_0]
\end{aligned}
$$

Since $\forall i. \lambda^{i \cdot n}(1 - \lambda^n)$ is positive, it follows that d_{i+1} is bigger, or not, than d_i based on whether $d^* < \mathtt{d}_0$ or not. If $d^* < \mathtt{d}_0$, we get $\mathtt{d}_0 > \mathtt{d}_1 > \ldots > d^*$, or, equivalently, $\mathbf{t_0} \sqsupseteq \mathbf{t_1} \sqsupseteq \ldots \sqsupseteq \mathbf{t}^*$. Similarly, if $d^* \geq \mathtt{d}_0$, we get $\mathtt{d}_0 \leq \mathtt{d}_1 \leq \ldots \leq d^*$, or, equivalently, $\mathbf{t_0} \sqsubseteq \mathbf{t_1} \sqsubseteq \ldots \sqsubseteq \mathbf{t}^*$. The feasibility of $w_0 \ldots w_n$ from each $\mathbf{t_i}$ follows from Lemma 9.7[Pg. 179] and $\mathbf{t_i} \sqsupseteq \mathbf{t_0}$. □

We denote the limit `deviation` $\mathbf{d}^*$ as $\mathtt{Saturate}(w_0 w_1 \ldots w_{n-1})$, i.e., $\mathtt{Saturate}(w_0 w_1 \ldots w_{n-1}) = \frac{1}{1-\lambda^n} \cdot \left(\sum_{p=0}^{n-1} w_p \cdot \lambda^{n-1-p}\right)$. Note that this deviation does not depend on the initial battery status $\mathbf{t_0}$. Accordingly, we extend the definition of the function `Saturate` to battery statuses as $\mathtt{Saturate}([\mathtt{e};\mathtt{d}], w_0 w_1 \ldots w_{n-1}) = [\mathtt{e}; \mathtt{Saturate}(w_0 w_1 \ldots w_n)]$.

Constructing the tree. For generating a finite tree with all the relevant bounded energy reachability information, we explore the feasible states and transitions starting from the initial state. However,

1. Extended states with high-enough energies are not explored further, and
2. If an extended state $\mathbf{q}$ that has an ancestor $\mathbf{q}'$ with the same control state and the same energy (but possibly a different deviation) is reached, we check the feasibility order, i.e., if $\mathbf{q} \sqsubseteq \mathbf{q}'$ or $\mathbf{q}' \sqsubseteq \mathbf{q}$. If $\mathbf{q} \sqsubseteq \mathbf{q}'$ we stop exploration from $\mathbf{q}$; otherwise, we saturate this 0-energy cycle from $\mathbf{q}'$ to $\mathbf{q}$, i.e., calculate the fixed battery status $\mathbf{t}^*$ to which an infinite run on that cycle will monotonically converge to (see Lemma 9.8[Pg. 180]). Then, we replace the battery status in $\mathbf{q}'$ with the maximum between battery status in $\mathbf{q}$ and $\mathbf{t}^*$.

The procedure `ComputeBERT` (Algorithm 9[Pg. 182]) computes the bounded-energy reachability tree `BERT`, given a BTS $\mathcal{B}$, an initial battery status $\mathbf{t}$, and an energy bound M. It is a rooted tree where each node is labelled with an extended state. During the procedure's execution, each node in the tree is either open (in `OpenNodes`) or closed, and exploration will only continue from open nodes. In addition, each node contains a Boolean field *star*, marking whether its label is a result of saturation. Initially, the root of `BERT` is labelled with the initial extended state $(s_\iota, \mathbf{t})$ and the root node is added to the set of `OpenNodes`.

In each step, one open node (`currNode`) is picked and removed from the set of `OpenNodes`. Let `currNode`.*label* $= (s, [\mathtt{e};\mathtt{d}])$. By default, we append to `currNode` children labelled by all feasible successors of $(s, [\mathtt{e};\mathtt{d}])$ (we call this an exploration step). If one of the following holds, we do not perform the exploration step.

- In case the energy (i.e., `e`) of `currNode` is greater than the given bound M, we stop exploration from it.
- In case an ancestor `ancestor` (with label $(s, [\mathtt{e};\mathtt{d}']))$ of `currNode` has the same control state and energy as `currNode`:
  - If $[\mathtt{e};\mathtt{d}] \sqsubseteq [\mathtt{e};\mathtt{d}']$ we stop exploration from `currNode`.
  - If $[\mathtt{e};\mathtt{d}'] \sqsubset [\mathtt{e};\mathtt{d}]$ we i) delete all the descendants of `ancestor`; and ii) replace the battery status in the label of `ancestor` with the $\sqsubset$-maximum between $[\mathtt{e};\mathtt{d}]$ and the zero-cycle saturation of $[\mathtt{e};\mathtt{d}']$, where $\mathtt{d}'$ is the 0-cycle saturation of the the weight sequence from `ancestor` to `currNode`. Note that if the weight sequence from `ancestor` to `currNode` is infeasible from $[\mathtt{e};\mathtt{d}']$ (which can happen if there is a saturation of another cycle between the `ancestor` and `currNode`),

we replace $[\mathtt{e};\mathtt{d}']$ by $[\mathtt{e};\mathtt{d}]$ and not the maximum.

When no open nodes are left, the procedure stops and returns `BERT`. We prove in a series of lemmata the properties of the procedure `ComputeBERT` and of the returned tree.

Algorithm 9 `ComputeBERT`: Computing the bounded-energy reachability tree

Input: Battery system $\mathcal{B} = \langle\langle S, \Delta, s_\iota, v\rangle, k, c\rangle$, initial battery status $\mathbf{t}$, energy bound M

```
BERT ← EmptyTree
BERT.root.label ← (s_ι, t);  BERT.root.star ← false
OpenNodes ← {BERT.root}
while OpenNodes ≠ ∅ do
  Pick and remove currNode from OpenNodes
  (s_0, [e; d]) ← currNode.label
  // If we found a good cycle, saturate that cycle
  if currNode has an ancestor ancestor with label (s_0, [e; d']) then
    if d > d' then
      s_0 ... s_n s_0 ← control state sequence in node labels from ancestor to
      currNode
      w_0 ... w_n ← v((s_0, s_1))v((s_1, s_2)) ... v((s_n, s_0))
      if w_0 w_1 ... w_n is feasible from [e; d'] and Saturate(w_0 ... w_n) > d
      then
        ancestor.label ←(s_0, [e; Saturate(w_0 ... w_n)]);
        ancestor.star ← true
      else {There was another cycle saturation between ancestor and
      currNode}
        ancestor.label ← [e; d]
      BERT.delete(all descendants of ancestor)
      OpenNodes ← OpenNodes ∪ {ancestor} \ all descendants of ancestor
    continue; // If d < d' there is no further exploration from the current
    node
  else {Explore one step further}
    for all (s_0, s') ∈ Δ do
      if Post([e; d], v(s_0, s')) is feasible then
        newNode ← new child of currNode
        newNode.label ← (s', Post([e; d], v(s_0, s')));
        newNode.star ← false
        if energy(Post([e; d], v(s_0, s'))) ≤ M then
          OpenNodes ← OpenNodes ∪ newNode
return BERT
```

Termination. We prove that Algorithm 9[Pg. 182] terminates for every input, by showing a bound on both the number of possible nodes in `BERT` and the number of node deletions in an execution. The latter bound follows from (i) every deletion event strictly increases a deviation value; and (ii) the number of possible values that a deviation can get in a deletion event is bounded. Note that this is in contradiction to the unbounded number of deviations that may occur in a trace of the BTS $\mathcal{B}$.

Lemma 9.9. *Algorithm 9[Pg. 182] terminates on all inputs.*

Proof. Termination follows from a bound on both the number of possible nodes in `BERT` and the number of node deletions in an execution.

The number of nodes in a tree depends on the fanout of the nodes and the length of the branches. The fanout of every node in `BERT` is bounded by the number of states in the BTS $\mathcal{B}$. As for the branches, the control state and energy of each node in a path is unique, except for the leaf, giving a bound of $|S| \times M + 1$ to the length of a path.

The bound on the number of deletions follows from (i) every deletion event strictly increases a deviation value; and (ii) the number of possible values that a deviation can get in a deletion event is bounded. Note that this is in contradiction to the unbounded number of deviations that may occur in a trace of the BTS $\mathcal{B}$.

Consider a node-deletion event in the execution of Algorithm 9[Pg. 182], and a new deviation value set to `ancestor`.*label*.

(i) The new value is either $\mathtt{Saturate}(w_0 \dots w_n)$ or $\mathtt{d}$, set in Line 14[Pg. 182] or Line 16[Pg. 182], which are in the scope of "if $\mathtt{Saturate}(w_0 \dots w_n) > \mathtt{d} > \mathtt{d}'$" or "if $\mathtt{d} > \mathtt{d}'$", respectively, while the old value is $\mathtt{d}'$.

(ii) The new value is uniquely determined by the following:

- The value of $\mathtt{Saturate}(w_0 \dots w_n)$, where $w_0 \dots w_n$ is the sequence of weights from `ancestor` to `currNode`; or
- The label of the last saturated (i.e., starred) node between `ancestor` and `currNode`, and the suffix $w_i \dots w_n$ of $w_0 \dots w_n$ corresponding to the segment from the last starred node to `currNode`, otherwise.

In the first case, resulting from Line 14[Pg. 182], the new value only depends on $w_0 \dots w_n$, which in turn is determined by $s_0 s_1 \dots s_n s_0$. In the second case, resulting from Line 16[Pg. 182], let $(s_i, [\mathtt{e}^{(i)}; \mathtt{d}^{(i)}])$ be the label of the last starred node. By Lemma 9.8[Pg. 180], $[\mathtt{e}^{(i)}; \mathtt{d}^{(i)}]$ only depends on the sequence of weights in a simple cycle of the BTS $\mathcal{B}$, thus may take a bounded number of possible values. The new value is then calculated by $\mathtt{Post}([\mathtt{e}^{(i)}; \mathtt{d}^{(i)}], w_i \dots w_n)$, which only depends on $[\mathtt{e}^{(i)}; \mathtt{d}^{(i)}]$ and the sequence $w_i \dots w_n$. □

Correctness. We now prove that `BERT` is a summarization of all extended states reachable through states of low-energy. Let $\mathit{Reach}(M)$ be the set of extended states reachable from the initial state of the BTS $\mathcal{B}$ through paths containing only extended states of energy less than M. In the lemmata below, we prove the following:

- *Soundness.* For every node `node` in `BERT` and for all $\epsilon > 0$, there is an extended state $\mathbf{q} \in \mathit{Reach}(M)$ such that $\mathbf{q} \sqsubseteq$ `node`.*label* and the difference between the deviations of $\mathbf{q}$ and `node`.*label* is smaller than ϵ.
- *Completeness.* For every extended state $\mathbf{q} \in \mathit{Reach}(M)$, there exists a node `node` in `BERT` such that $\mathbf{q} \sqsubseteq$ `node`.*label*.

Lemma 9.10 (Soundness)**.** *For every node* `node` *with label* $(s, [\mathtt{e}; \mathtt{d}])$ *encountered in an execution of Algorithm 9[Pg. 182] and* $\epsilon > 0$*, there exists an extended state* $(s, [\mathtt{e}; \mathtt{d} - \delta])$ *reachable from* $(s_\iota, \mathbf{t})$ *with* $0 \leq \delta < \epsilon$*.*

Proof. The claim can be proved by looking at all the points in the algorithm where a new label is created (lines 10[Pg. 182], 14[Pg. 182], and 25[Pg. 182]). The claim is trivially true for the initial label of the root. Assume as induction

hypothesis that the claim holds for every label encountered upto the current point of the execution. Fix $\epsilon \geq 0$.

Saturation. Let the new label be created during a deletion event, in line 10[Pg. 182] or line 14[Pg. 182]. In the case the label of `currNode` is copied to the label of `ancestor`, the proof follows immediately as we are just copying an existing label. Otherwise, we are taking the `Saturate` value, and the proof is based on Lemma 9.8[Pg. 180]. By iterating the path from `ancestor` to `currNode` a sufficient number of times, we can get as close as necessary (i.e., within ϵ) to the limit of the zero-weight cycle saturation. Lemma 9.8[Pg. 180] also gives us that every iteration is feasible.

Exploration. Let $(s', [\mathtt{e}'; \mathtt{d}'])$ be a new node label created in line 25[Pg. 182]. Choosing $\epsilon' < \min(\frac{\epsilon}{\lambda}, c\mathtt{e}' + (1-c)\mathtt{d}')$, by the induction hypothesis, there is a feasible path from $(s_\iota, \mathbf{t})$ to $(s, [\mathtt{e}; \mathtt{d} - \delta'])$ with $0 \leq \delta' < \epsilon'$.

Now, we have $[\mathtt{e}'; \mathtt{d}'] = \mathtt{Post}([\mathtt{e}; \mathtt{d}], w) = [\mathtt{e} + w; \lambda\mathtt{d} + w]$ and $\mathtt{Post}([\mathtt{e}; \mathtt{d} - \delta'], w) = [\mathtt{e}'; \mathtt{d}' - \lambda \cdot \delta']$. Letting $\delta = \lambda\delta'$, we get $0 \leq \delta = \lambda\delta' < \lambda\epsilon' \leq \epsilon$. If we prove that $[\mathtt{e}'; \mathtt{d}' - \delta]$ is feasible, we are done as we have shown that the path from $(s_\iota, \mathbf{t})$ to $(s, [\mathtt{e}; \mathtt{d} - \delta'])$ followed by the feasible transition from $(s, [\mathtt{e}; \mathtt{d} - \delta'])$ to $(s', [\mathtt{e}'; \mathtt{d}' - \delta])$ is a path from $(s_\iota, \mathbf{t})$ to $(s', [\mathtt{e}'; \mathtt{d}' - \delta])$.

As $[\mathtt{e}'; \mathtt{d}']$ is feasible, we get that $c\mathtt{e}' + (1-c)\mathtt{d}' > 0$. As we chose that $\epsilon' < c\mathtt{e}' + (1-c)\mathtt{d}'$, we get that $c\mathtt{e}' + (1-c)\mathtt{d}' - \delta = c\mathtt{e}' + (1-c)\mathtt{d}' - \lambda\delta' > c\mathtt{e}' + (1-c)\mathtt{d}' - \lambda\epsilon' > c\mathtt{e}' + (1-c)\mathtt{d}' - \lambda(c\mathtt{e}' + \mathtt{d}') > (1-\lambda)(c\mathtt{e}' + (1-c)\mathtt{d}') > 0$. This completes the proof for this case.

□

Lemma 9.11 (Completeness)**.** *Let $(s, [\mathtt{e}; \mathtt{d}])$ be an extended state with $\mathtt{e} < M$ that is reachable from $(s_\iota, \mathbf{t})$ through extended states with energy less than M. Then, there exists a node with label $(s, [\mathtt{e}; \mathtt{d}'])$ in* `BERT` *with $\mathtt{d}' \geq \mathtt{d}$.*

Proof. We prove the lemma by induction on the length of the path from $(s_\iota, \mathbf{t})$ to $(s, [\mathtt{e}; \mathtt{d}])$. For paths of length 0, it is trivial as $(s_\iota, \mathbf{t})$ is the initial label of the root. Suppose we have proven the claim for paths upto the length $n-1$.

It is easy to induct to length n. Suppose the path of length $n-1$ ending with $(s, [\mathtt{e}; \mathtt{d}])$ is extended to length n by adding $(s^\sharp, [\mathtt{e}^\sharp; \mathtt{d}^\sharp])$. The proof has two cases:

- If the node labelled with $(s, [\mathtt{e}; \mathtt{d}'])$ is a non-leaf node, then it has a successor labelled $(s^\sharp, [\mathtt{e}^\sharp; \mathtt{d}^{\sharp'}])$, with $\mathtt{d}^{\sharp'} > \mathtt{d}^\sharp$, which is the required node. This follows from lines 21[Pg. 182]–27[Pg. 182] of Algo. 9[Pg. 182].
- If the `BERT` node labelled with $(s, [\mathtt{e}; \mathtt{d}'])$ is a leaf node, it will have an ancestor labelled $(s, [\mathtt{e}; \mathtt{d}''])$ with $\mathtt{d}'' \geq \mathtt{d}$. This follows from line 19[Pg. 182] of Algorithm 9[Pg. 182]. Since the latter node is not a leaf, we comply with the previous case, and we are done.

□

9.5 Model Checking

We are now ready to tackle the finite-automaton, Büchi, and Streett emptiness problems for BTSs. We show that the problems are decidable and give suitable algorithms. The algorithms are based on Theorem 9.5[Pg. 178] and analysis of the bounded-energy reachability tree, as constructed in Section 9.4[Pg. 179].

Algorithm 10 Algorithm for the reachability problem

Input: Battery system $\mathcal{B} = \langle\langle S, W, T, s_\iota\rangle, k, c\rangle$
Input: Target control states $T \subseteq S$
Input: Initial battery status $\mathbf{t}$
Output: $\mathit{Reachable}((s_\iota, \mathbf{t}), T)$
1: $M \leftarrow \texttt{HighEnergyConstant}(\mathcal{B}, |S|)$
2: $\texttt{BERT} \leftarrow \texttt{ComputeBERT}(\mathcal{B}, (s_\iota, \mathbf{t}), M)$
3: **if** $\exists \texttt{node} \in \texttt{BERT} : [\texttt{node}.\mathit{label} = (s, [\texttt{e}; \texttt{d}]) \wedge s \in T] \vee$
$[\texttt{node}.\mathit{label} = (s, [\texttt{e}; \texttt{d}]) \wedge \texttt{e} > M \wedge \texttt{GraphReachability}(\mathcal{B}, s, T)]$
then
4: **return true**
5: **return false**

9.5.1 Finite-Automaton Emptiness

Combining the results from the previous section on bounded energy reachability tree and Theorem 9.5[Pg. 178], we can obtain a complete algorithm for the finite-automaton emptiness problem in a battery system. Algorithm 10[Pg. 185] solves the reachability problem for a given BTS $\mathcal{B}$ with initial battery status $\mathbf{t}$ and target set T. Given a BTS $\mathcal{B}$ with states S, an initial battery status $\mathbf{t}$, the algorithm works as follows:

- Build a bounded-energy reachability tree $\texttt{BERT} = \texttt{ComputeBERT}(\mathcal{B}, (s_\iota, \mathbf{t}), M)$, where $M = \texttt{HighEnergyConstant}(\mathcal{B}, |S|)$;
- If there is a node label $(s, [\texttt{e}; \texttt{d}])$ in BERT where s is in the target set T, return `true`;
- If there is a node label $(s, [\texttt{e}; \texttt{d}])$ in BERT where $\texttt{e} > M$, and some node in the target set is reachable from $(s, [\texttt{e}; \texttt{d}])$ through an energy-feasible path, return `true`;
- Otherwise, return `false`.

The correctness proof of the algorithm follows from Lemma 9.4[Pg. 177] and the soundness and completeness of the bounded-energy reachability tree (Lemmas 9.10[Pg. 183]–9.11[Pg. 184]). The following theorem states that this algorithm can be implemented in polynomial space in the inputs.

Theorem 9.12. *The finite-automaton emptiness problem for BTSs is decidable in polynomial space with respect to the number of control states in the BTS and a unary encoding of weights.*

Proof. The major part of the algorithm is the construction of the bounded-energy reachability tree. For a given energy bound M, this tree can contain an exponential number of nodes in M. However, using standard on-the-fly techniques, we can reduce the space complexity, only storing the current branch of the tree being explored. The corresponding space is the product of the number of nodes in each branch and the bits required for storing a node's label.

By the proof of Lemma 9.9[Pg. 183], the length of each branch in the tree is bounded by $|S| \times M + 1$, where S are the states of the given BTS $\mathcal{B}$. For the finite-automaton emptiness algorithm, we use $M = \texttt{HighEnergyConstant}(\mathcal{B}, |S|)$ and by the proof of Lemma 9.4[Pg. 177], we have $M \leq |S|W + \frac{W}{c(1-\lambda)}$, where W is the maximal negative weight in the BTS. With a unary encoding of the constants,

M is polynomial in the size of the input.

To complete the proof, we need to show that all the labels created in BERT can be represented in polynomial space in the energy bound M.

A label contains a control state, an energy, and a deviation. There are $|S| < M$ control states and up to M different energies.

As for the deviations, they are generated by a sequence of operations, involving two functions: Post (defined in Section 9.3[Pg. 174]) and Saturate (defined in Lemma 9.8[Pg. 180]). By Lemma 9.8[Pg. 180], the value of Saturate is independent of the deviation value before saturation. Hence, the deviation at each node in BERT is a result of the last Saturate operation in the branch of BERT leading to the node, followed by some Post operations. By the proof of Lemma 9.9[Pg. 183], the length of each branch in the tree is polynomial in M, implying up to M applications of Post. Hence, it is left to show that the Saturate function generates a deviation that can be stored in space polynomial in M, and that each application of the Post function adds up to b bits, where b is polynomial in M.

By Lemma 9.8[Pg. 180], given a sequence $w_0 w_1 \dots w_{n-1}$ of weights, $\texttt{Saturate}(w_0 w_1 \dots w_{n-1}) = \frac{1}{1-\lambda^n} \cdot \left(\sum_{p=0}^{n-1} w_p \cdot \lambda^{n-1-p} \right)$. The space required to store this value is polynomial in the constant λ and n, where $n \leq |S| < M$. In each application of Post on a battery status $[\mathtt{e};\mathtt{d}]$ and weight w, we have, by Proposition 9.1[Pg. 175], that $\texttt{Post}([\mathtt{e};\mathtt{d}], w) = [\mathtt{e}';\mathtt{d}']$, with $d' = \lambda \cdot \mathtt{d} + w$. Hence, storing $\mathtt{d}'$ requires up to b bits more than storing $\mathtt{d}$, where b is polynomial in the constant λ and $|w| < M$. □

9.5.2 Büchi and Streett Emptiness

Suppose we are given a BTS $\mathcal{B} = \langle\langle S, \Delta, s_\iota, v\rangle, k, c\rangle$ and a Büchi condition given by a set of Büchi states $B \subseteq S$. Our approach to Büchi emptiness consists of two major parts. If there exists a Büchi trace containing an extended state with energy more than $M_B = 3|S| + 2W|S|^2$, the problem can be reduced to the Büchi problem for simple-energy systems (Theorem 9.5[Pg. 178]). Therefore, we concentrate on the case where the energy of states is bounded by M_B. Here, the key idea is that if the energies of the extended states are bounded, then a BTS has a Büchi trace if and only if it has a Büchi trace of a special form.

First, we define the notion of an energy-unique path: we call a control trace $s_0 s_1 \dots s_n$ *energy-unique* if we have $\sum_{i=0}^{p} v((s_i, s_{i+1})) \neq \sum_{i=0}^{q} v((s_i, s_{i+1})) \vee s_p \neq s_q$ for $p \neq q$. Intuitively, $s_0 s_1 \dots s_n$ is energy-unique if no trace whose corresponding control trace is $s_0 \dots s_n$ has two extended states with the same control state and equal energy. Similarly, $s_0 s_1 \dots s_n$ is an *energy-unique* 0-*energy cycle* if $s_0 s_1 \dots s_n$ is energy-unique and $\sum_{i=0}^{n-1} v((s_i, s_{i+1})) + v((s_n, s_0)) = 0$.

The following theorem intuitively states that if there exists a bounded-energy Büchi trace in $\mathcal{B}$, then there exists a lasso-shaped bounded-energy Büchi trace where the first state of the cycle in the lasso is a Büchi state and the cycle in the lasso has one of the two following forms:

- the cycle is an energy-unique 0-energy cycle $s_0^l \dots s_n^l s_0^l$, such that the sequence $s_0^l s_1^l \dots s_n^l s_0^l$ is feasible from s_0^l with the battery status $\texttt{Saturate}(w_0 w_1 \dots w_n)$, where $w_i = v((s_i^l, s_{i+1}^l))$ for $i < n$ and $w_n = v((s_n^l, s_0^l))$; or
- the cycle is an energy-unique 0-energy cycle composed of an alternating

sequence of energy-unique paths and energy-unique 0-energy cycles. Here, every energy-unique 0-energy cycle in the sequence is unique.

Theorem 9.13. *Suppose a BTS $\mathcal{B}$ has a Büchi trace such that every extended state has energy less than some constant M. Then, $\mathcal{B}$ has a Büchi trace π such that the corresponding control trace θ has one of the following two forms:*

Form 1 $\theta = \theta_h(s_0^l s_1^l \dots s_n^l)^\omega$ *where s_0^l is a Büchi state, and $s_0^l s_1^l \dots s_n^l s_0^l$ is an energy-unique 0-energy cycle.*

Form 2 $\theta = \theta_h(\theta_l)^\omega$*, where* $\theta_h, \theta_l \in S^*$ *and* $\theta_l = (s_0^0 \dots s_{k_0}^0)(\theta_{l0})^{r_0} \dots (s_0^n \dots s_{k_1}^n) \dots (\theta_{ln})^{r_n}(s_0^{n+1} \dots s_{k_{n+1}}^{n+1})$ *and (a) each θ_{li} is a distinct energy-unique 0-energy cycle; (b) each $s_0^i \dots s_{k_i}^i$ is a energy-unique path; (c) θ_l is a 0-energy cycle; and (d) s_0^0 is a Büchi state.*

Intuitively, the above forms say the following:

Form 1 There is a reachable 0-energy cycle containing a Büchi node which is feasible from its `Saturate` value; or

Form 2 There exist 0-loop cycles θ_{li}'s where each $\theta_{l(i+1)}$ is feasibly reachable from the `Saturate` value of $\theta_{l(i)}$, and θ_{l0} is feasibly reachable from the `Saturate` value of θ_{ln}. Further, the paths $(s_0^i \dots s_{k_i}^i)$ from each θ_{li} to the next form a 0-energy cycle.

The proof proceeds by taking a witness Büchi trace and reducing it to one of the two forms by deleting parts of the trace where the initial and final energies and control states are the same, while the final deviation is less than the initial deviation.

Proof of Theorem 9.13[Pg. 187]. We do not prove the lemma formally, but instead provide a procedure that can take any bounded-energy Büchi trace and transform it into a Büchi trace in one of the two forms.

Suppose $\mathcal{B} = \langle\langle S, v, \Delta\rangle, k, c\rangle$ is a BTS, $(s_\iota, [\mathtt{e}; \mathtt{d}])$ be the initial extended state, and $B \subseteq S$ a set of Büchi states. Suppose the BTS contains a Büchi trace $\pi = (s_0, [\mathtt{e}_0; \mathtt{d}_0])(s_1, [\mathtt{e}_1; \mathtt{d}_1])(s_2, [\mathtt{e}_2; \mathtt{d}_2]) \dots$ where the energy of each extended state is bounded by M.

As π is a Büchi trace and the number of values $\mathtt{e}_i$ can take is bounded, there exists some s and $\mathtt{e}$ such that $s \in B$ and there exist infinitely many indices i such that $s_i = s$ and $\mathtt{e}_i = \mathtt{e}$. Let i_0, i_1, $\dots$ be the increasing sequence of all such indices. For every $p \in \mathbb{N}$, define π_p to be $(s_{i_p}, [\mathtt{e}_{i_p}; \mathtt{d}_{i_p}]) \dots (s_{i_{p+1}-1}, [\mathtt{e}_{i_{p+1}-1}; \mathtt{d}_{i_{p+1}-1}])$. Further, define $\pi^h = (s_0, [\mathtt{e}_0; \mathtt{d}_0]) \dots (s_{i_0-1}, [\mathtt{e}_{i_0-1}; \mathtt{d}_{i_0-1}])$. Let θ_p be the control trace corresponding to π_p and let θ^h be the control trace corresponding to π^h.

Our strategy for proving the theorem is as follows: (a) we categorize the infinite set of π_p into a finite number of classes; (b) therefore, we will have at least one class having an infinite number of π_p; and (c) we pick a representative from this class and construct a lasso-shaped Büchi trace using it.

We first define the `Simplify` function which simplifies each π_p. The key purpose of this simplification is to replace each π_p by an alternate control trace which satisfies the conditions for either Form 1's or Form 2's lasso. The `Simplify` function takes an initial battery status (say $[\mathtt{e}; \mathtt{d}]$) and a finite control trace (say $u_0 u_1 \dots u_n$), and produces an alternate control trace. It works as follows:

- If $u_0 u_1 \dots u_n$ is energy-unique, we return $u_0 u_1 \dots u_n$.

- If $u_0 u_1 \dots u_n$ is not energy-unique, we first check for energy-unique 0-energy cycles (say $u_i \dots u_j$). Now, we check if $\texttt{Post}([\texttt{e};\texttt{d}], u_0 \dots u_i) \geq \texttt{Post}([\texttt{e};\texttt{d}], u_0 \dots u_j)$. If so, we delete the segment $u_{i+1} \dots u_j$ from the control trace and start over with the simplification procedure.
- If there exist two energy-unique 0-energy cycles (say $u_i \dots u_j$ and $u_p \dots u_q$ and $p > j$) which are the same, i.e., $u_i \dots u_j = u_p \dots u_q$, we replace $u_i \dots u_q$ with $(u_i \dots u_j)^m$ where m is large enough such that $\texttt{Post}([\texttt{e};\texttt{d}], u_0 \dots u_{i-1}(u_i \dots u_j)^m) \geq \texttt{Post}([\texttt{e};\texttt{d}], u_0 \dots u_q)$ and start over. Such an m exists because of the zero-cycle saturation lemma. Since we have that the deviation increases on taking the $u_p \dots u_q$ cycle (as $u_p \dots u_q$ was not eliminated in the previous step), we have that $\texttt{Post}([\texttt{e};\texttt{d}], u_0 \dots u_q)$ is less that the `Saturate` value of the cycle. Hence, by taking a large enough m, we can get the deviation close as possible to the `Saturate` value of $u_p \dots u_q$ and hence, the deviation will become more than $\texttt{Post}([\texttt{e};\texttt{d}], u_0 \dots u_q)$.
- If none of the above apply, we return the control trace.

The `Simplify` function satisfies two key properties:

- The control traces that satisfy conditions imposed by either Form 1 and Form 2 on the cycle of the lasso, i.e., they are either energy-unique cycles or are made of alternating energy-unique segments and energy-unique 0-energy cycles.
- If the returned control trace is $\theta^\natural$, $\texttt{Post}([\texttt{e};\texttt{d}], u_0 \dots u_n) \sqsubseteq \texttt{Post}([\texttt{e};\texttt{d}], \theta^\natural)$.

Now, we iteratively transform the Büchi trace into a Büchi trace in Form 1 or Form 2 as follows:

- First, let $\theta^*_0 = \theta^h$. We define $\theta^*_i = \theta^*_{i-1} \cdot \texttt{Simplify}(\theta_i, \texttt{Post}([\texttt{e}_0;\texttt{d}_0], \theta^*_{i-1}))$. The simpler Büchi trace is the one corresponding to the limit of all θ^*_n. It can be shown to be feasible using the second property of the `Simplify` function.
- Now, the previous trace is made up of segments returned by the `Simplify` function. Each of these segments satisfy the conditions for either Form 1's or Form 2's lasso. For a Form 1 lasso segment $s^l_0 s^l_1 \dots s^l_n$, we say its class is itself. For a Form 2 lasso segment $(s^0_0 \dots s^0_{k_0})(\theta_{l0})^{r_0} \dots (s^n_0 \dots s^n_{k_1})(\theta_{ln})^{r_n}(s^{n+1}_0 \dots s^{n+1}_{k_{n+1}})$, we say its class is $(s^0_0 \dots s^0_{k_0})(\theta_{l0})^* \dots\ (s^n_0 \dots s^n_{k_1})(\theta_{ln})^*(s^{n+1}_0 \dots s^{n+1}_{k_{n+1}})$, i.e., we ignore the ri's. Note that there are only a finite number of possible classes. As there are only a finite number of classes, there is at least one class having an infinite number of segments in it.
  - If there are more than one such class, we define a new control trace with only the segments from one class with an infinite number of segments, and start over with the simplification procedure.
  - If there is only one such class, we define the final Form 1 or Form 2 trace based on it. If the class corresponds to Form 1, i.e., the class is defined by a single energy-unique 0-energy cycle (say θ_l), we return $\theta^h \cdot \theta^*_0 \dots \theta^*_{m-1}(\theta_l)^\omega$ where m^{th} segment is the first segment in the infinite class. If the class corresponds to Form 2, i.e., the class is of the form $s^0_0 s^0_1 \dots s^0_{k0}(\theta_{l0})^* \dots s^n_0 \dots s^n_{k_1}(\theta_{ln})^* s^{n+1}_0 \dots s^{n+1}_{k_{n+1}}$, we need to pick the constants $r0$, $r1$, ... to replace the $*$'s in the lasso. Here, we pick each ri high enough so that the trace $s^{i+1}_0 \dots s^{i+1}_{k_{i+1}} \cdot \theta_{l(i+1)})$ is feasible from any feasible state

reached after θ_{li}^{ri}. Using the proof of the zero-cycle saturation lemma, it can be shown that such ri's exist. We return $\theta^h \cdot \theta_0^* \ldots \theta_{m-1}^* (s_0^0 s_1^0 \ldots s_{k0}^0 (\theta_{l0})^{r0} \ldots s_0^n \ldots s_{k_1}^n (\theta_{ln})^{rn} s_0^{n+1} \ldots s_{k_{n+1}}^{n+1} \theta_l)^\omega$ where the mth segment is the first segment in the infinite class.

□

The algorithm. The Büchi-emptiness algorithm intuitively consists of two separate parts: (a) searching for high energy Büchi traces (where some extended state has energy more than $M_B = \texttt{HighEnergyConstant}(\mathcal{B}, 3|S| + 2W|S|^2)$); and (b) searching for low energy fair traces (where every extended state has energy less than M_B). The algorithm first constructs $\texttt{BERT} = \texttt{ComputeBERT}(\mathcal{B}, (s_\iota, \texttt{t}), M_B)$.

High energy. For every node label $(s, [\texttt{e}; \texttt{d}])$ in the BERT where $\texttt{e} > M_B$, we check (using techniques of [41]) whether there exists an energy-feasible fair trace from it.

Form 1 low energy. For every node label $(s, [\texttt{e}; \texttt{d}])$ in BERT with $\texttt{e} \leq M_B$ and $s \in B$, we check if there exists a fair trace of Form 1 starting from $(s, [\texttt{e}; \texttt{d}])$. Performing this check entails constructing energy-unique 0-energy cycles θ_l starting from s and examining if θ_l is feasible from $\texttt{Saturate}((s, [\texttt{e}; \texttt{d}]), \theta_l)$.

Form 2 low energy. For every $s \in B$ and $\texttt{e} < M_B$, we run Algorithm 11[Pg. 189] with initial state s and initial battery status $[\texttt{e}; \texttt{d}]$ to check if there exists a fair trace of Form 2. Here, d is the maximum deviation of a node label which has control state s and energy e.

Algorithm 11 Finding form 2 low energy fair traces

Input: Battery system $\mathcal{B} = \langle\langle S, \Delta, s_\iota, v\rangle, k, c\rangle$, Energy bound $M \in \mathbb{N}$, Control state s, Initial battery status $[\texttt{e}; \texttt{d}]$

1: $d^* \leftarrow \texttt{d}$
2: **while** true **do**
3: BERT $\leftarrow$ ComputeBERT$(\langle\langle S, \Delta, s, v\rangle, k, c\rangle, [\texttt{e}; \texttt{d}^*], M)$
4: $P \leftarrow \{\texttt{leaf}.label \mid$ BERT leaf leaf has label $(s, [\texttt{e}; \texttt{d}']) \wedge$
    leaf has a starred ancestor $\}$
5: **if** $P = \emptyset$ **then**
6: **return** false
7: **else if** $d^* = \max\{d' \mid (s, [\texttt{e}; \texttt{d}']) \in P\}$ **then**
8: **return** true
9: **else**
10: $d^* \leftarrow \max\{d' \mid (s, [\texttt{e}; \texttt{d}']) \in P\}$

Lemma 9.14. *Algorithm 11[Pg. 189] returns* true *if $\mathcal{B}$ contains a Büchi trace of Form 2, and* false *otherwise.*

Intuitively, Algorithm 11[Pg. 189] works by finding some deviation $\texttt{d}^*$ such that $(s, [\texttt{e}; \texttt{d}^*])$ is feasibly reachable from itself through some number of 0-energy cycle saturations (represented by starred nodes). In every iteration of the while-loop, it decreases the possible value for $\texttt{d}^*$ to the largest deviation for control-state s and energy e reachable from the previous value of $\texttt{d}^*$ through some starred nodes. If $\texttt{d}^*$ becomes so low that we are not able to saturate any 0-energy cycle starting from $(s, [\texttt{e}; \texttt{d}^*])$, then we return false.

Proof of Lemma 9.14[Pg. 189]. We start with showing that Algorithm 2 terminates on all inputs. First, if the value of $\mathtt{d}^*$ does not decrease in a particular iteration, the execution of the loop stops. Hence, suppose the value of $\mathtt{d}^*$ decreases in each iteration of the loop. Now, in every call to `ComputeBERT`, the value of d^* is less than in the previous call. As the value of the initial deviation d^* decreases in each iteration, the set of feasible traces is a subset of the set of feasible traces in the previous iteration. In particular, the set P in an iteration is a (not necessarily strict) subset of the set P for the previous iteration. If the set is equal to the previous one, termination is guaranteed due to line 8[Pg. 189]. Otherwise, we have a strictly decreasing set P. As P starts out being finite, it finally becomes empty in some iteration. Then, termination is guaranteed due to line 6[Pg. 189].

The correctness proof consists of two parts. First, we need to show that if the algorithm returns `true`, then there exists a Büchi trace of Form 2. This part follows easily from the correctness properties of `ComputeBERT`. Considering the last `BERT` constructed, we know that there exists a feasible path from $(s, [\mathtt{e}; \mathtt{d}^*])$ to some $(s, [\mathtt{e}; \mathtt{d}^* - \epsilon])$ for every $\epsilon > 0$. We can show that by taking ϵ small enough, the same path is feasible repeatedly. We do not show this here as the proof is straight-forward but tedious. Once we show this, it can be seen the trace made up of repetitions of this path is of Form 2 and that it is feasible.

Second, we need to show that if the algorithm returns false, then there does not exist a Büchi trace of Form 2. Suppose $\mathcal{B}$ contains a Büchi trace $\pi = (s_0, [\mathtt{e}_0; \mathtt{d}_0]) \ldots$ of Form 2 and let $(s, \mathtt{e})$ be such that $s \in B$ and $s_i = s \wedge \mathtt{e}_i = \mathtt{e}$ for infinitely many i. Let i_0, i_1, ... be the sequence of such i's. Now, consider the call to Algorithm 2 with the initial extended state $(s, [\mathtt{e}; \mathtt{d}])$. From the properties of the `ComputeBERT` algorithm, we have the following:

- As Algorithm 11[Pg. 189] is called with the maximum deviation possible for each of control state and energy pair, at the end of initial part of the trace (before the cycle), the deviation is less than d, i.e., $d_{i_0} \leq d$.
- After each iteration of the lasso in the trace, the deviation becomes less than the next value of d^* in the execution of the algorithm, i.e., d_{i_k} is at most the value of d^* in the kth iteration of the while-loop.
- In the last iteration (say q^{th}) of the while-loop, the d^* value is too low, i.e., the lasso of the trace is infeasible from it. As the d_{i_q} is less than d^* in this iteration, the lasso is infeasible from d_{i_q}.

Hence, the cycle of the Büchi trace will become infeasible from some $(s, [\mathtt{e}; \mathtt{d}_{i_k}])$ and hence, the trace is infeasible. This gives us a contradiction, completing the proof of the required lemma. □

Theorem 9.15. *Büchi emptiness for battery transition systems is decidable in polynomial space with respect to the number of states and a unary encoding of weights and constants.*

Equipped with Theorem 9.13[Pg. 187], the proof of Theorem 9.15[Pg. 190] follows in a similar fashion as in Theorem 9.12[Pg. 185].

Using similar techniques, we can construct an algorithm for Streett emptiness. In this case, also keeping track of the set of states visited along each branch of the reachability tree.

Theorem 9.16. *Street emptiness for battery transition systems is decidable in polynomial space with respect to the number of states and a unary encoding of*

weights and constants.

9.5.3 ω-Regular Model Checking

Equipped with a procedure for checking Büchi emptiness (Theorem 9.15[Pg. 190]), one can check whether a given BTS $\mathcal{B}$ satisfies any ω-regular constraint φ that is defined with respect to $\mathcal{B}$'s states. Such a constraint can be formalized, for example, by a linear temporal logic (LTL) formula, whose atomic propositions are the names of the states in $\mathcal{B}$. Indeed, any ω-regular constraint φ can be translated to a Büchi automaton $\mathcal{A}$, such that $\mathcal{A}$'s language is equivalent to the language of φ (or to the language of its negation, as is the common practice in the case of an LTL formula) [167]. Now, one can take the product of $\mathcal{A}$ and $\mathcal{B}$, defined in the usual way, getting a BTS $\mathcal{C}$ with a Büchi emptiness problem.

As Streett emptiness is a special case of ω-regular model checking, one may wonder why we bothered to have Theorem 9.16[Pg. 190]. The reason lies in the complexity – In Theorem 9.16[Pg. 190], we show that Streett emptiness can be solved in the same complexity class as the one for Büchi emptiness, while translating a Streett automaton into a Büchi automaton might involve an exponential state blowup [147].

9.6 Case Study

We conclude this chapter with a case study relating to controlling an energy-constrained robot. We first define a toy language for programming the robot controller, inspired by various real languages for programming robots, and define how the different constructs interact with the environment.

The setting. We model a semi-autonomous robot that operates in an *arena* D_L where each $l \in D_L$ is a possible location of the robot. For example, a location can be an (x, y) vector, providing the position of the robot in a plane of $1,000 \times 1,000$ squares.

We model the *environment* of the robot as a function that gives attributes to each location in the arena. Formally, the environment is $\mathcal{E} : D_L \to \langle D_{E_1}, D_{E_2}, \ldots, D_{E_m} \rangle$ where each D_{E_i} is a finite domain of some property. For example, the environment may define the terrain of each location and whether it lies in the sun or in the shade, in which case $\mathcal{E}(3, 5) = \langle$"smooth terrain", "sun"$\rangle$ means that the location $(3, 5)$ is a sunny place with a smooth terrain. Note that, in this case study, the environment is time invariant.

The actions of the robot are governed by its *control program*. In each time step, denoted by a 'tick', the control program computes *output actions* based on some *external inputs*, *sensor values*, and the values of the robot's *internal variables*.

The external input is given by *input variables* $\langle I_1, \ldots, I_k \rangle$, each over a finite domain D_{I_i}, and it comes from an external independent agent. The sensor values, given by *sensor variables* $\langle s_1, \ldots, s_r \rangle$, over the finite domains $D_{s_1}, \ldots, D_{s_r}$, are computed automatically based on the environment of the robot and its current location. Formally, for each sensor variable s_i there is a function $\xi_i : \mathcal{E} \times D_L \to D_{s_i}$. The robot also has some *internal variables*, $\langle N_1, \ldots, N_g \rangle$, over the finite domains $D_{N_1}, \ldots, D_{N_g}$, used for putting a logic in its behavior.

```
program := statements

statements := statement | statement; statements
statement := (label : tick) | action_var = expr
        | internal_var = expr | skip
        | if (expr == 0) statements else statements
        | while (expr == 0) statements
expr := sensor_var | input_var | internal_var
        | expr + expr | constant | expr * expr
        | (expr == 0) ? expr : expr
```

Figure 9.5: Syntax of the robot-control language

The output actions are given by *output variables* $\langle A_1, \ldots, A_l \rangle$, over the finite domains $D_{A_1}, \ldots, D_{A_l}$. Upon performing the actions, the current location, given in the variable L, is automatically computed based on the previous location and the actions; formally, by a function $\eta : D_L \times D_{A_1} \times \ldots \times D_{A_l} \to D_L$.

The *state* of the robot, $\mathcal{V}$, encapsulates the values of all the above variables. There is a *cost function* Energy which gives the energy gain (positive) or consumption (negative) of actions in the given environment, i.e., Energy is of type $D_{E_1} \times \ldots \times D_{E_m} \times D_{A_1} \times \ldots \times D_{A_l} \to \mathbb{Z}$. For the functions η, ξ_i, and Energy, we use the short-hand of applying the function to the whole state instead of the relevant variables. For example, instead of writing "$\xi_i(l) = v$ and value of L in state σ is l", we write "$\xi_i(\sigma) = v$".

The controller language. The language of the robot-control program is defined by the syntax shown in Figure 9.5[Pg. 192]. Most of the constructs in this language are standard, and will not be explained in detail. Note that the program cannot directly write to the location variables and sensor variables, but can only write to the internal variables and action variables. The most interesting construct in the syntax is the `tick` statement. Intuitively, the `tick` statement performs the actions described by the output variables (i.e., changes the location using the η function) and reads new values into the sensor variables (based on the environment and the current state, using the ξ_i functions) and into the input variables (non-deterministically). The formal semantics of the `tick` statement is described in the next paragraph.

We provide in Example 9.17[Pg. 192] a simple setting of an environment, a control program, and the finite domains of the various variables.

Example 9.17.

The environment (arena).

$x \backslash y$	1	2	3	4
1	-	-	×	-
2	×	/	×	-
3	-	/	/	/
4	-	×	/	/

Legends.

☐ : *Sun* ; ■ : *Shade*

- : *hard* ; / : *soft* ; ×: *obstacle*

The robot variables.

Location.	$D_L = \{(1,1),(1,2),\ldots,(4,4)\}$
Inputs.	$D_{I_1} = \{Move, None\}$
	$D_{I_2} = \{Front, Back, Left, Right\}$
Sensors.	$D_{s_1} = \{InTheSun, InTheShade\}$
	$D_{s_2} = \{SunOnFront, NoSunOnFront\}$
	. . . Sensors for sun and obstacles all around
	$D_{s_9} = \{ObstacleOnRight, NoObstacleOnRight\}$
Actions.	$D_{A_1} = \{Move, None\}$
	$D_{A_2} = \{Front, Back, Left, Right\}$
Internal.	$D_{N_1} = \{InTheSun, InTheShade\}$
	$D_{N_2} = \{WasInTheSun, WasInTheShade\}$
	$D_{N_3} = \{Was^2InTheSun, Was^2InTheShade\}$

The cost function. *(The direction does not matter.)*

$\mathsf{Energy}(Sun, Hard/Soft, None) = +12$
$\mathsf{Energy}(Sun, Hard, Move) = +1$
$\mathsf{Energy}(Sun, Soft, Move) = -1$
$\mathsf{Energy}(Shade, Hard/Soft, None) = -5$
$\mathsf{Energy}(Shade, Hard, Move) = -12$
$\mathsf{Energy}(Shade, Soft, Move) = -15$

The robot-control program. *The program, intuitively, defines the following behavior.*

- *Obey the external input, whenever it is legal. Otherwise, do nothing, if legal, or else check for a legal action.*
- *The constraints for a legal action:*
  - *Do not go into an obstacle. (A location out of the arena is considered as an obstacle.)*
  - *Do not stay in the sun for more than two consecutive steps.*
  - *Whenever staying for two consecutive steps in the sun, avoid the sun for at least two consecutive steps.*

The code is straightforward; we give below some of its fragments.

```
while(1) {
// Check if the input is legal
  // Moving into an obstacle?
  if (I1 = Move &&
       (   I2 = Front && S6 = ObstacleOnFront
        || I2 = Back && S7 = ObstacleOnBack
        || I2 = Left && S8 = ObstacleOnLeft
```

```
        || I₂ = Right && S₉ = ObstacleOnRight )
     )
      A₁ := None
  // Too much in the sun?
  if (N₂ = WasInTheSun
      && (N₁ = InTheSun || N₃ = Was²InTheSun) && ...
     )
     // Choose a legal action
     if (N₁ = InTheShadow)
        A₁ := None
     else if (S₂=NoSunOnFront && S₆=NoObstacleOnFront)
        A₁ := Move; A₂ := Front
      ...
  label1 : tick;
  N₃ := (N₂=WasInTheSun)? Was²InTheSun:Was²InTheShade
  N₂ := (N₁=InTheSun)? WasInTheSun : WasInTheShade
  N₁ := S₁
}
```

Semantics. Consider a robot-control program P, and fix diffusion constant k and a width constant c for a battery. We define the semantics of P in the standard small-step operational style. We summarize the state of the program as $(\sigma, \mathbf{t})$ where σ is a valuation of the variables, and $\mathbf{t}$ is a battery status. Therefore, the small-step semantics is given by a relation $\Rightarrow$ where intuitively, $(P, (\sigma, \mathbf{t})) \Rightarrow (P', (\sigma', \mathbf{t}'))$ holds if executing the first step from the program fragment P at state $(\sigma, \mathbf{t})$ leads to $(\sigma', \mathbf{t}')$ and the remaining program fragment is P'.

We assume that all the constructs except `tick` are executed instantaneously, and without any consumption of power; hence, the only construct that updates the battery status in the summary is the `tick`. Therefore, for all the other constructs, we do not explicitly present the semantics, but point out that the semantics are similar to a standard while-language. For the `tick` construct, we define the semantics using the proof rules from Figure 9.6[Pg. 195].

Intuitively, on executing a `tick`, the effects of the output actions are performed, the sensor variables are updated based on the new location and environment, the next valuation of the input variables is given, and then the battery status is updated based on the cost of the actions in the current environment.

Problem statement. We consider model-checking problems; that is, asking whether a given model satisfies a given specification. The model, in our case, is a robot and its environment; namely, a robot-control program, battery constants, an initial battery status, an environment with locations D_L, and an initial location. The specification is a regular or ω-regular language over the (finite or infinite) sequences of locations in D_L. The model-checking problem is affirmatively answered if the robot has a path, in the given setting, such that the sequence of locations along the path belongs to the language of the specification.

Consider, for example, the setting of Example 9.17[Pg. 192] together with an initial location $(1, 1)$, a battery width constant $\frac{1}{2}$, a battery diffusion constant $\frac{1}{8}$, and an initial battery status $(16, 16)$. A regular specification can ask, for

$$\frac{}{(\texttt{label}:\texttt{tick},(\sigma,\mathbf{t}))\Rightarrow(\texttt{effects};\texttt{sensors};\texttt{inputs};\texttt{battery},(\sigma,\mathbf{t}))}\ \text{Tick}$$

$$\frac{cost(\sigma)=w\quad \texttt{Post}(\mathbf{t},w)=\mathbf{t}'}{(\texttt{battery},(\sigma,\mathbf{t}))\Rightarrow(\texttt{skip},(\sigma,\mathbf{t}'))}\ \text{Battery}$$

$$\frac{v_1\in D_{I_1}\quad\ldots\quad v_l\in D_{I_l}}{(\texttt{inputs},(\sigma,\mathbf{t}))\Rightarrow(\texttt{skip},(\sigma[\forall k:I_k:=v_k],\mathbf{t}'))}\ \text{Inputs}$$

$$\frac{v_1=\xi_1(\sigma)\quad\ldots\quad v_r=\xi_r(\sigma)}{(\texttt{sensors},(\sigma,\mathbf{t}))\Rightarrow(\texttt{skip},(\sigma[\forall k:s_k:=v_k],\mathbf{t}'))}\ \text{Sensors}$$

$$\frac{v=\eta(\sigma)}{(\texttt{effects},(\sigma,\mathbf{t}))\Rightarrow(\texttt{skip},(\sigma[\forall k:L:=v],\mathbf{t}'))}\ \text{Effects}$$

Figure 9.6: Semantics of `tick`

instance, whether the robot has a finite path reaching the location $(3,2)$. An ω-regular specification can ask, say, whether the robot has an infinite path visiting the location $(1,4)$ infinitely often, while avoiding the locations $(3,1)$ and $(4,4)$.

Model-checking algorithm. Given a control-program P, an environment $\mathcal{E}$, a battery width constant c, a battery diffusion constant k, an initial battery status $\mathbf{t}_\iota$, and an initial variable valuation σ_ι, we define the equivalent battery transition system $\mathsf{BTS}[\![P,\mathcal{E},c,k,\mathbf{t}_\iota,\sigma_\iota]\!]=\langle\langle S,\Delta,s_\iota,v\rangle,c,k\rangle$ as follows.

Let L be the set of labels of the `tick` statements in the program.

- A state in the BTS is a pair (l,σ) where $l\in L$ is a label, and σ is a valuation of all the variables in the program.
- There exists a transition from (l_1,σ_1) to (l_2,σ_2) on weight w if for some program fragments P_1 and P_2:
  - there exist battery statuses $\mathbf{t_1}$ and $\mathbf{t_2}$ and a proof that $((\texttt{l}_1:\texttt{tick});P_1,(\mathbf{t_1},\sigma_1))\Rightarrow((\texttt{l}_2:\texttt{tick});P_2,(\mathbf{t_2},\sigma_2))$ where the *Tick* rule is applied exactly once; and
  - there exist battery statuses $\mathbf{t_1}$ and $\mathbf{t_2}$, such that there is a proof $(P,(\sigma_\iota,\mathbf{t_1}))\Rightarrow((\texttt{l}_1:\texttt{tick});P_1,(\sigma_1,\mathbf{t_2}))$.
- The cost of a transition from (l,σ) is w if applying the *cost* function on the valuation of the environment and action variables in σ is w.
- The initial state s_ι is given by (l,σ) such that there exists a program fragment P_1 and a battery status $\mathbf{t}$ such that there is a proof of $(P,(\sigma_\iota,\mathbf{t}))\Rightarrow(((\texttt{l}:\texttt{tick});P_1),(\sigma,\mathbf{t}))$ containing no applications of the *Tick* rule. Due to the determinism of our language, it is guaranteed that there exists only one such (l,σ).

A part of the BTS corresponding to Example 9.17[Pg. 192] is given in Figure 9.7[Pg. 196].

We have the following theorem.

Theorem 9.18. *Consider a robot model-checking problem consisting of a control-program P, an environment $\mathcal{E}$ with locations D_L, a battery width constant c, a battery diffusion constant k, an initial battery status $\mathbf{t}_\iota$, an initial variable valuation σ_ι, and a regular or ω-regular language ϕ over the sequences*

The battery diffusion constant $k = \frac{1}{8}$ and its width constant $c = \frac{1}{2}$.

Figure 9.7: A part of the BTS that corresponds to the robot and its environment, as described in Example 9.17[Pg. 192]. The best path to location $(3, 2)$ appears in boldface blue. This path is infeasible by the BTS semantics, while feasible by models that are based on an ideal-energy resource.

of locations in D_L.

Let $\mathcal{B} = \mathsf{BTS}[\![P, \mathcal{E}, c, k, \mathsf{t}_\iota, \sigma_\iota]\!]$. For a control state $b \in \mathcal{B}$, let $\mathsf{Location}(b)$ be the valuation of the robot location variable in b. Let ϕ' be a regular or ω-regular language over sequences of control locations in $\mathcal{B}$, such that a sequence $b_0, b_1, \ldots \in \Phi'$ iff $\mathsf{Location}(b_0), \mathsf{Location}(b_1), \ldots \in \Phi$.

Then, the robot model-checking problem is equivalent to the BTS model-checking of $\mathcal{B}$ and ϕ'.

Battery vs. ideal energy. Model-checking the robot behavior, taking into account the non-ideal aspects of the energy resource, is inherently different from considering the battery as an ideal energy resource, as demonstrated in Example 9.17[Pg. 192]. There, the robot cannot go with an initial battery status of $(16, 16)$ from location $(1, 1)$ to $(3, 2)$ (cf., Theorem 9.3[Pg. 177].) On the other hand, it is possible to go from location $(4, 4)$ to $(3, 4)$, starting with the same initial battery status. Note that such a situation is impossible with a model that is based on an ideal-energy resource, as the energy loss going from location $(4, 4)$ to $(3, 4)$ is 15, while from location $(1, 1)$ to $(3, 2)$ it is only 14! (The reason, as elaborated on in Sections 9.2[Pg. 171]–9.3[Pg. 174], lies in the influence of the energy changes along the path on the available charge of the battery.)

9.7 Summary

We presented the first discrete formal model of battery systems and showed that the standard automaton emptiness problems for this model are decidable. Further, these battery transition systems do not fall into the large class of well-structured transition systems. We also applied these model checking algorithms in a case study on energy-constrained robots.

Chapter 10

Discussion

We finish this part with a short discussion on the related work and possible directions of future work.

10.1 Future Work

Quantitative Synthesis for Concurrency. The approach presented in Chapter 6[Pg. 90] examines every correct strategy. There is thus the question whether there exists a practical algorithm that overcomes this limitation by eliminating partial strategies based on performance considerations. Also, we did not consider the question which solution(s) to present to the programmer in case there is a number of correct strategies with the same (or similar) performance. Furthermore, one could perhaps incorporate some information on the expected workload to the performance model as a usage model. One possible extension is to consider the synthesis of programs that access concurrent data structures rather than just finite state concurrent program, and another is to create benchmarks from which performance models can be obtained automatically.

Quantitative Abstraction Refinement. We intend to use quantitative abstraction to aid partial-program synthesis, as quantitative reasoning is necessary if the goal is not to synthesize any program, but rather the best performing program according to quantitative measures such as performance or robustness (as in Chapter 6[Pg. 90]). Furthermore, the anytime verification property of the refinements we proposed can lead to *anytime synthesis* methods, that is, methods that would synthesize correct programs, and refine these into more optimized versions if given more time. Further, our improvements to WCET estimation algorithms could also be used in synthesis of optimal programs.

On the practical side, extending our method with interpolation in first-order theories (e.g., the theory of arrays) could yield transition predicates over data structures, allowing us to analyze heap-accessing programs. Further, our segment-based abstraction refinement framework can be extended with other cost information, for example for analyzing the energy usage of programs.

Battery Transition Systems. In terms of future work, a natural direction is to explore standard program analysis and program synthesis questions for systems that use batteries. For example, to begin with, one could define an extension to standard imperative languages to allow programs to branch based on the status of the battery. For programs written in such a language, one could attempt to compute invariants about the combined program- and battery-state through abstract interpretation. Also, one could attempt battery-aware partial-program synthesis for such a language. This would be a generalization of the battery-aware scheduling problem studied in [108]. Another direction to explore is the possibility of solving two-player games for battery transition system, leading to battery-aware algorithms for synthesis of reactive systems.

10.2 Related Work

Performance-Aware Synthesis. Synthesis from specifications is a classical problem [57, 59, 133]. More recently, sketching, a technique where a partial implementation of a program is given and a correct program is generated automatically, was introduced [154] and was extended to concurrent programs in [153]. Synthesis for synchronization constructs is also an old problem and the celebrated paper [59] presented an algorithm for synthesis of synchronization skeletons. Recent study in this field includes topics like lock placement [53, 80] and fence insertion [160, 82]. However, none of the above approaches consider performance-aware algorithms for sketching; they focus on qualitative synthesis without any performance measure.

In [168, 53] fixed optimization criteria (such as preferring short atomic sections or fine-grained locks) are considered. Optimizing these measures may not lead to optimal performance on all architectures, and none of these works consider parametric performance models. Further, the synthesis problem for concurrent programs in general is equivalent to imperfect information games. However, none of the above works consider the general framework of games for synthesis, or the parametric performance model.

Recent works have considered quantitative synthesis [22, 49]; however the focus of these works has been the synthesis of sequential systems from temporal logic specifications. Moreover, all these works consider perfect information games. Neither imperfect information games nor quantitative objectives were considered before for synthesis for concurrent programs. We require imperfect information due to concurrency and quantitative objectives for performance measures.

Recently, another approach that has become popular for synthesis of efficient programs is stochastic super-optimization [149]. Here, the input is a small piece of straight-line assembly code and the optimizer searches through a large number of other equivalent programs to find the most efficient one. The optimizer is specific to each architecture and uses deep knowledge about the architecture to evaluate the performance of different versions of the program.

Quantitative Abstraction Refinement The theory of abstractions for programs was introduced in [65]. We build on the transition predicates of [136] and segment covers of [67] to construct our quantitative abstractions. The CEGAR algorithm was introduced in [60] and is widely used. Automated abstraction

refinement for transition predicates was presented in [62]. To the best of our knowledge, CEGAR-like algorithms for quantitative properties have not yet been studied.

However, quantitative abstractions and refinements have been introduced for *stochastic* systems [72, 111, 99, 104, 128], where the need for quantitative reasoning arises because of stochasticity. They are mainly directed towards the estimation of expected values, and the algorithms reflect this fact. The probabilistic work does not aim at handling accumulative properties like the limit-average property.

Abstractions (but not CEGAR-like algorithms) have been proposed for certain quantitative properties of non-probabilistic systems, such as cache abstractions for WCET analysis [84]. WCET analysis using interval abstraction was performed in [139, 138]. The power consumption analysis of software based on the costs of single instructions was presented in [158, 159], with improved models built subsequently (e.g., the model in [150] accounts for asynchronously consumed energy for peripheral devices). Our WCET analysis of executables based on quantitative abstraction refinement could be adapted for power consumption analysis using these models.

Worst-case Execution Time Analysis. In [135], transition invariants for proving program termination are derived. Following this framework, in [64] ranking functions as transition predicates are iteratively inferred and refined as lexicographic termination argument. The method relies on constraint constraint solving and exploiting the abstraction refinement approach of [62]. A method for computing loop bounds from loop invariants expressing transition predicates is described in [90]. This work uses templates for two-state invariants, whereas we infer segment predicates by interpolation. Furthermore, the approach uses patterns in conjunction with abstract interpretation to drive the invariant generation, and infers symbolic bounds on the number of loop iterations. While we also use loop bounds as symbolic parameters in our abstraction framework, our work is conceptually different from [90]. We rely on segment-based abstraction to reason about the time complexity of instruction sequences, and iteratively compute interpolants describing program properties under the current abstractions. Our interpolants are not necessarily inductive invariants, however they are derived automatically without using patterns or templates.

Most state-of-the-art static WCET approaches, see e.g. [113, 14], compute a constant WCET by using concrete inputs to the program and relying on numeric loop bounds. Unlike these methods, our WCET estimates are parametric, describing symbolic expressions over the program parameters. When using [113, 14] on different values of parameters, a new WCET analysis needs to be made for each set of values. This is not the case with our method, Our parametric WCET is computed only once and replacing parameters with their values yields the precise WCET for each set of concrete values, without rerunning the WCET analysis as in [113, 14].

Parametric WCET calculation is also described in [29, 102], where polyhedra-based abstract interpretation is used to derive integer constraints on program executions. Solving these constraints is done using parametric integer linear programming problem, and a parametric WCET is obtained. In [102] various heuristics are applied in order to approximate program paths by small

linear expressions over execution frequencies of program blocks. When compared to [29, 102], our segment-based abstraction allows us to reason about the WCET as a property of a sequence of instructions rather than a state property. To the best of our knowledge, interpolation-based algorithms and segment-based abstraction have not yet been studied for the WCET analysis of programs.

Battery Transition Systems. Batteries are involved devices, exhibiting various different physical phenomena. Accordingly, there are many different works considering these aspects, for example scheduling the load among several batteries [18, 54, 106, 108], optimizing the lifetime of a battery with respect to the "cycle aging effect" [2] and analyzing the thermal effects.

To the best of our knowledge, this is the first work to formally analyze an energy system with a non-ideal energy source. Previous work has either considered ideal energy sources (for example, [47]) or provides approximations for the battery life-time with respect to various discharge scenarios (for example, [106, 107]).

Our model checking algorithms follows the approach taken in the Karp-Miller tree [110] which can be used in general for well-structured transition systems [86]. However, our systems are not well-structured and a naive application of this technique does not entail termination of algorithms. We use ideas from flattable systems [15] and additional analysis of BTSs to produce a terminating version of these algorithms. In particular, we also use an intricate analysis of BTSs to get an algorithm for deciding Büchi and Streett properties. This kind of analysis is not possible for general flattable systems.

10.3 Summary

We extended three well-known verification and synthesis techniques to different quantitative specifications. While each of the extensions are similar to the original techniques at a high level, there are significant differences that are necessary to handle quantitative specifications. In Chapter 6[Pg. 90], new techniques were required for efficient evaluation of strategies and elimination of partial strategies. In Chapter 7[Pg. 110] and Chapter 8[Pg. 144], segment-based abstractions were required rather than standard state-based abstractions. In Chapter 9[Pg. 168], the infinite state systems that arose from modelling batteries were not a part of any previously known decidable class — several techniques were combined to prove that the model checking problems are decidable.

Chapter 11

Conclusion

The main aim of this dissertation was to explore the use of quantitative specifications in verification and synthesis of systems. We summarize our contributions towards this goal, present some ongoing work and possible directions of future work. Here, we present just a few directions of future work as it has already been in the discussed extensively in the discussion sections concluding each part.

Summary. In the first part, our focus was on reactive systems. We presented the simulation distances framework for specifying reactive systems. The basic concept in this framework is a simulation distance, which is an extension of the classical simulation relation. The simulation distances are measured using a quantitative extension of the simulation game where the simulating player may simulate actions using mis-matching actions, but pays a penalty for such mismatches. We showed how simulation distances can be used to capture quantitative versions of various relations between an implementation and specification such as correctness, coverage, and robustness. Further, we showed how simulation distances can be used to solve the synthesis from incompatible specifications problem where the goal is to synthesize an implementation that comes closest to being correct with respect to a number of possibly incompatible specifications. We presented a number of case studies showing how simulation distances can be used at various points during system development, such as test-case generation and robustness analysis. Further, we also proved properties of simulation distances that enable various verification techniques such as compositional analysis and analysis through abstractions.

In the second part, we focussed on extending some classical techniques in verification and synthesis to quantitative specifications. We presented an algorithm for counter-example guided inductive synthesis for synthesizing concurrent programs that are not only correct, but also perform optimally. We showed that our algorithm can synthesize optimal programs in cases such as optimistic concurrency where previous techniques based on heuristics such as minimizing critical sections do not apply. We also presented two kinds of abstraction and abstraction refinement techniques for quantitative properties — state-based and segment-based. We applied these techniques to the analysis of worst-case execution time of programs and showed that our abstraction-based techniques return tighter bounds on the worst-case execution than standard tools. Finally, we presented a formal model for systems that interact with a battery.

We discussed briefly the short-comings of the classical models (such as energy transition systems and energy games), showed how our model overcomes these, and presented novel model checking algorithms for battery transition systems for ω-regular properties.

Ongoing work. Work on a number of different aspects of the work presented in this thesis is ongoing. Related to the simulation distances framework, one direction being explored is the use of the coverage distance for test-case generation. This was briefly described in the example from Chapter 3[Pg. 30]. Another work currently ongoing is the rewriting of a large classical specification in the simulation distances framework to examine the practical advantages and disadvantages of the framework. The specification chosen is of the Advances High-Performance Bus (AHB) from the Advanced Micro-controller Bus Architecture (AMBA) [88, 23].

Related to the second part, work that grew out directly from the problem of synthesis of efficient programs was presented in [39] and [40]. In [39] and [40], the aim is not to synthesize the most efficient program in the given solution space, but instead to repair a given faulty concurrent program while avoiding expensive synchronization constructs such as locks and atomic sections.

Another line of work that is ongoing is the extension of the battery transition systems from Chapter 9[Pg. 168] to 2-player games. While the battery systems are not well structured transition systems, the model checking algorithms were possible as cycles in battery transitions systems can be accelerated precisely. However, we know of no game solving algorithm based on acceleration. Therefore, novel techniques will be required to augment standard game solving algorithms with acceleration-based methods. Game solving algorithms for 2-player battery games will enable solving schedulability problems (without any approximation) with energy sources [108].

Future work. In the long term, there are many possible directions of work based on the techniques presented in this thesis.

Related to simulation distances framework, one interesting direction is to see if it is possible to define a high-level specification language based on the framework. In practice, specifications are rarely written directly in a formal framework (such as automata or temporal logics), but are instead written in a convenient high-level language (such as Lustre [31], Scade, etc) whose semantics are defined in terms of a low-level formal framework. Defining such a high-level language for specifications and error models in the simulation distances framework would bring us closer to practical adoption of quantitative frameworks.

Related to segment-based abstractions presented in Chapter 7[Pg. 110] and interpolation for segment-based abstractions from Chapter 8[Pg. 144], one possible direction is to apply these techniques to termination analysis. While the theoretical aspects has been explored in [67], the practical issues have not addressed yet. If successful, the segment-based abstraction techniques for termination will subsume techniques based on summarization [63] and trace abstraction [124].

Another possible application of quantitative abstractions is in the domain of partial-program synthesis. The technique of abstraction-guided synthesis (presented in [168]) was introduced to make synthesis feasible for large partial-programs. In abstraction-guided synthesis certain specializations of the partial-

program are preemptively eliminated based on abstract counter-examples, i.e., errors in an abstract program (which might not be present in the concrete program). Using quantitative abstractions for this technique would entail the same for "performance errors", i.e., some programs may be eliminated as they potentially have bad performance even if they might be correct.

The material presented in Chapter 9[Pg. 168] is a small step in the study of physics-based formal models for systems interacting with energy sources. There are several other, more complex, physical models for batteries [141, 142]. It is to be seen whether these models can be discretized and formalized, and whether they have decidable model checking properties. Applying the model presented in Chapter 9[Pg. 168] to practical schedulability problems is another important goal for future work.

Bibliography

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. In *LICS*, pages 165–175, 1988.

[2] Ron Adany and Tami Tamir. Online algorithm for battery utilization in electric vehicles. In *FedCSIS*, pages 349–356, 2012.

[3] Aws Albarghouthi and Kenneth L. McMillan. Beautiful interpolants. In *CAV*, pages 313–329, 2013.

[4] Shaull Almagor, Udi Boker, and Orna Kupferman. Discounting in LTL. In *TACAS*, pages 424–439, 2014.

[5] Shaull Almagor and Orna Kupferman. Max and sum semantics for alternating weighted automata. In *ATVA*, pages 13–27, 2011.

[6] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking for real-time systems. In *LICS*, pages 414–425, 1990.

[7] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, pages 209–229, 1992.

[8] Rajeev Alur, Loris D'Antoni, Jyotirmoy V. Deshmukh, Mukund Raghothaman, and Yifei Yuan. Regular functions and cost register automata. In *LICS*, pages 13–22, 2013.

[9] Rajeev Alur and David L. Dill. The theory of timed automata. In *REX Workshop*, pages 45–73, 1991.

[10] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. In *FOCS*, pages 164–169, 1989.

[11] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. In *COMPOS*, pages 23–60, 1997.

[12] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. Alternating refinement relations. In *CONCUR*, pages 163–178, 1998.

[13] Guy Avni and Orna Kupferman. Parameterized weighted containment. In *FoSSaCS*, pages 369–384, 2013.

[14] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: An open toolbox for adaptive WCET analysis. In *SEUS*, pages 35–46, 2010.

[15] Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Ph. Schnoebelen. Flat acceleration in symbolic model checking. In *ATVA*, pages 474–488, 2005.

[16] Sebastian S. Bauer, Uli Fahrenberg, Line Juhl, Kim G. Larsen, Axel Legay, and Claus R. Thrane. Weighted modal transition systems. *Formal Methods in System Design*, 42(2):193–220, 2013.

[17] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal - a tool suite for automatic verification of real-time systems. In *Hybrid Systems*, pages 232–243, 1995.

[18] Luca Benini, Giuliano Castelli, Alberto Macii, Enrico Macii, Massimo Poncino, and Riccardo Scarsi. Extending lifetime of portable systems by battery scheduling. In *DATE*, pages 197–203, 2001.

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

[20] Henrik Björklund, Sven Sandberg, and Sergei G. Vorobyov. A discrete subexponential algorithm for parity games. In *STACS*, pages 663–674, 2003.

[21] Roderick Bloem, Roberto Cavada, Ingo Pill, Marco Roveri, and Andrei Tchaltsev. RAT: A tool for the formal analysis of requirements. In *CAV*, pages 263–267, 2007.

[22] Roderick Bloem, Krishnendu Chatterjee, Thomas A. Henzinger, and Barbara Jobstmann. Better quality in synthesis through quantitative objectives. In *CAV*, pages 140–156, 2009.

[23] Roderick Bloem, Stefan J. Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Specify, compile, run: Hardware from psl. *Electr. Notes Theor. Comput. Sci.*, 190(4):3–16, 2007.

[24] Bard Bloom. *Ready simulation, bisimulation, and the semantics of CCS-like languages*. PhD thesis, MIT, 1989.

[25] Mark S. Boddy. Anytime problem solving using dynamic programming. In *AAAI*, pages 738–743, 1991.

[26] Udi Boker, Thomas A. Henzinger, and Arjun Radhakrishna. Battery transition systems. In *POPL*, pages 595–606, 2014.

[27] Gerardine G. Botte, Venkat R. Subramanian, and Ralph E. White. Mathematical modeling of secondary lithium batteries. *Electrochimica Acta*, 45(15-16):2595 – 2609, 2000.

[28] J. Richard Büchi and Lawrence H. Landweber. Definability in the monadic second-order theory of successor. *J. Symb. Log.*, 34(2):166–170, 1969.

[29] Stefan Bygde and Björn Lisper. Towards an automatic parametric wcet analysis. In *WCET*, 2008.

[30] Paul Caspi and Albert Benveniste. Toward an approximation theory for computerised control. In *EMSOFT*, pages 294–304, 2002.

[31] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *POPL*, pages 178–188, 1987.

[32] Pavol Cerný, Krishnendu Chatterjee, Thomas A. Henzinger, Arjun Radhakrishna, and Rohit Singh. Quantitative synthesis for concurrent programs. In *CAV*, pages 243–259, 2011.

[33] Pavol Cerný, Martin Chmelik, Thomas A. Henzinger, and Arjun Radhakrishna. Interface simulation distances. In *GandALF*, pages 29–42, 2012.

[34] Pavol Cerný, Sivakanth Gopi, Thomas A. Henzinger, Arjun Radhakrishna, and Nishant Totla. Synthesis from incompatible specifications. In *EMSOFT*, pages 53–62, 2012.

[35] Pavol Cerný, Thomas A. Henzinger, and Arjun Radhakrishna. Quantitative simulation games. In *Essays in Memory of Amir Pnueli*, pages 42–60, 2010.

[36] Pavol Cerný, Thomas A. Henzinger, and Arjun Radhakrishna. Simulation distances. In *CONCUR*, pages 253–268, 2010.

[37] Pavol Cerný, Thomas A. Henzinger, and Arjun Radhakrishna. Simulation distances. *Theor. Comput. Sci.*, 413(1):21–35, 2012.

[38] Pavol Cerný, Thomas A. Henzinger, and Arjun Radhakrishna. Quantitative abstraction refinement. In *POPL*, pages 115–128, 2013.

[39] Pavol Cerný, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. Efficient synthesis for concurrency by semantics-preserving transformations. In *CAV*, pages 951–967, 2013.

[40] Pavol Cerný, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. Regression-free synthesis for concurrency. In *CAV (to appear)*, 2014.

[41] Arindam Chakrabarti, Luca de Alfaro, Thomas A. Henzinger, and Mariëlle Stoelinga. Resource interfaces. In *EMSOFT*, pages 117–133, 2003.

[42] Krishnendu Chatterjee. *Stochastic Omega-Regular Games*. PhD thesis, EECS Department, University of California, Berkeley, October 2007.

[43] Krishnendu Chatterjee, Luca de Alfaro, Rupak Majumdar, and Vishwanath Raman. Algorithms for game metrics. In *FSTTCS*, pages 107–118, 2008.

[44] Krishnendu Chatterjee and Laurent Doyen. Energy parity games. *Theor. Comput. Sci.*, 458:49–60, 2012.

[45] Krishnendu Chatterjee, Laurent Doyen, and Thomas A. Henzinger. Quantitative languages. In *CSL*, pages 385–400, 2008.

[46] Krishnendu Chatterjee, Laurent Doyen, and Thomas A. Henzinger. Expressiveness and closure properties for quantitative languages. In *LICS*, pages 199–208, 2009.

[47] Krishnendu Chatterjee, Laurent Doyen, Thomas A. Henzinger, and Jean-François Raskin. Generalized mean-payoff and energy games. In *FSTTCS*, pages 505–516, 2010.

[48] Krishnendu Chatterjee, Thomas A. Henzinger, and Barbara Jobstmann. Environment assumptions for synthesis. In *CONCUR*, pages 147–161, 2008.

[49] Krishnendu Chatterjee, Thomas A. Henzinger, Barbara Jobstmann, and Rohit Singh. Measuring and synthesizing systems in probabilistic environments. In *CAV*, pages 380–395, 2010.

[50] Krishnendu Chatterjee, Thomas A. Henzinger, and Marcin Jurdzinski. Mean-payoff parity games. In *LICS*, pages 178–187, 2005.

[51] Krishnendu Chatterjee, Marcin Jurdzinski, and Thomas A. Henzinger. Simple stochastic parity games. In *CSL*, pages 100–113, 2003.

[52] Krishnendu Chatterjee, Mickael Randour, and Jean-François Raskin. Strategy synthesis for multi-dimensional quantitative objectives. *Acta Inf.*, 51(3-4):129–163, 2014.

[53] Sigmund Cherem, Trishul M. Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. In *PLDI*, pages 304–315, 2008.

[54] Carla-Fabiana Chiasserini and Ramesh R. Rao. Energy efficient battery management. *IEEE Journal on Selected Areas in Communications*, 19(7):1235–1245, 2001.

[55] Hana Chockler, Orna Kupferman, and Moshe Y. Vardi. Coverage metrics for formal verification. *STTT*, 8(4-5):373–386, 2006.

[56] Hana Chockler, Orna Kupferman, and Moshe Y. Vardi. Coverage metrics for temporal logic model checking*. *Formal Methods in System Design*, 28(3):189–212, 2006.

[57] Alonzo Church. Logic, arithmetic, and automata. In *Proceedings of the International Congress of Mathematicians*, 1962.

[58] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *CAV*, pages 359–364, 2002.

[59] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71, 1981.

[60] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[61] J. Cochet-Terrasson, G. Cohen, S. Gaubert, M. McGettrick, and J.-P. Quadrat. Numerical computation of spectral elements in max-plus algebra, 1998.

[62] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction refinement for termination. In *SAS*, pages 87–101, 2005.

[63] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Summarization for termination: no return! *Formal Methods in System Design*, 35(3):369–387, 2009.

[64] Byron Cook, Abigail See, and Florian Zuleger. Ramsey vs. lexicographic termination proving. In *TACAS*, pages 47–61, 2013.

[65] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[66] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.

[67] Patrick Cousot and Radhia Cousot. An abstract interpretation framework for termination. In *POPL*, pages 245–258, 2012.

[68] Siddhartha R. Dalal, Ashish Jain, Nachimuthu Karunanithi, J. M. Leaton, Christopher M. Lott, Gardner C. Patton, and Bruce M. Horowitz. Model-based testing in practice. In *ICSE*, pages 285–294, 1999.

[69] Luca de Alfaro, Marco Faella, and Mariëlle Stoelinga. Linear and branching system metrics. *IEEE Trans. Software Eng.*, 35(2):258–273, 2009.

[70] Luca de Alfaro, Thomas A. Henzinger, and Rupak Majumdar. Discounting the future in systems theory. In *ICALP*, pages 1022–1037, 2003.

[71] Luca de Alfaro, Rupak Majumdar, Vishwanath Raman, and Mariëlle Stoelinga. Game refinement relations and metrics. *Logical Methods in Computer Science*, 4(3), 2008.

[72] Luca de Alfaro and Pritam Roy. Magnifying-lens abstraction for markov decision processes. In *CAV*, pages 325–338, 2007.

[73] Aldric Degorre, Laurent Doyen, Raffaella Gentilini, Jean-François Raskin, and Szymon Torunczyk. Energy and mean-payoff games with imperfect information. In *CSL*, pages 260–274, 2010.

[74] Josee Desharnais, Vineet Gupta, Radha Jagadeesan, and Prakash Panangaden. Metrics for labelled markov processes. *Theor. Comput. Sci.*, 318(3):323–354, 2004.

[75] Sumesh Divakaran, Deepak D'Souza, and Raj Mohan Matteplackel. Conflict-tolerant specifications in temporal logic. In *ISEC*, pages 103–110, 2010.

[76] Marc Doyle, Thomas F Fuller, and John Newman. Modeling of galvanostatic charge and discharge of the lithium/polymer/insertion cell. *Journal of the Electrochemical Society*, 140(6):1526–1533, 1993.

[77] Manfred Droste and Paul Gastin. Weighted automata and weighted logics. *Theor. Comput. Sci.*, 380(1-2):69–86, 2007.

[78] Deepak D'Souza and Madhu Gopinathan. Conflict-tolerant features. In *CAV*, pages 227–239, 2008.

[79] Calvin C. Elgot and Michael O. Rabin. Decidability and undecidability of extensions of second (first) order theory of (generalized) successor. *J. Symb. Log.*, 31(2):169–181, 1966.

[80] Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In *POPL*, pages 291–296, 2007.

[81] Uli Fahrenberg, Axel Legay, and Claus R. Thrane. The quantitative linear-time–branching-time spectrum. In *FSTTCS*, pages 103–114, 2011.

[82] Xing Fang, Jaejin Lee, and Samuel P. Midkiff. Automatic fence insertion for shared memory multiprocessing. In *ICS*, pages 285–294, 2003.

[83] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach, Revised (Paperback)*. Course Technology, 1998.

[84] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache behavior prediction by abstract interpretation. *Sci. Comput. Program.*, 35(2):163–189, 1999.

[85] Jerzy Filar and Koos Vrieze. *Competitive Markov decision processes*. Springer-Verlag New York, Inc., New York, NY, USA, 1996.

[86] Alain Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.

[87] Thomas F Fuller, Marc Doyle, and John Newman. Relaxation phenomena in lithium-ion-insertion cells. *Journal of the Electrochemical Society*, 141(4):982–990, 1994.

[88] Yashdeep Godhal, Krishnendu Chatterjee, and Thomas A. Henzinger. Synthesis of amba ahb from formal specification: a case study. *STTT*, 15(5-6):585–601, 2013.

[89] Sean Gold. A pspice macromodel for lithium-ion batteries. In *Battery Conference on Applications and Advances*, pages 215–222, 1997.

[90] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In *PLDI*, pages 292–304, 2010.

[91] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen wcet benchmarks: Past, present and future. In *WCET*, pages 136–146, 2010.

[92] Steven C Hageman. Simple PSPICE models let you simulate common battery types. *EDN*, 38(22):117, 1993.

[93] Richard W Hamming. Error detecting and error correcting codes. *Bell System technical journal*, 29(2):147–160, 1950.

[94] Constance L. Heitmeyer, Myla Archer, Ramesh Bharadwaj, and Ralph D. Jeffords. Tools for constructing requirements specifications: the scr toolset at the age of nine. *Comput. Syst. Sci. Eng.*, 20(1), 2005.

[95] Thomas A. Henzinger, Orna Kupferman, and Sriram K. Rajamani. Fair simulation. In *CONCUR*, pages 273–287, 1997.

[96] Thomas A. Henzinger, Rupak Majumdar, and Vinayak S. Prabhu. Quantifying similarities between timed systems. In *FORMATS*, pages 226–241, 2005.

[97] Thomas A. Henzinger, Jan Otop, and Roopsha Samanta. Lipschitz robustness of finite-state transducers. *CoRR*, abs/1404.6452, 2014.

[98] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Elsevier Inc., 2008.

[99] Holger Hermanns, Björn Wachter, and Lijun Zhang. Probabilistic CEGAR. In *CAV*, pages 162–175, 2008.

[100] Krystof Hoder, Laura Kovács, and Andrei Voronkov. Playing in the grey area of proofs. In *POPL*, pages 259–272, 2012.

[101] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley, 2003.

[102] Benedikt Huber, Daniel Prokesch, and Peter P. Puschner. A formal framework for precise parametric wcet formulas. In *WCET*, pages 91–102, 2012.

[103] Milka Hutagalung, Martin Lange, and Étienne Lozes. Revealing vs. concealing: More simulation games for büchi inclusion. In *LATA*, pages 347–358, 2013.

[104] Michael Huth. On finite-state approximants for probabilistic computation tree logic. *Theor. Comput. Sci.*, 346(1):113–134, 2005.

[105] Ranjit Jhala and Kenneth L. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, pages 459–473, 2006.

[106] Marijn R. Jongerden. *Model-based energy analysis of battery powered systems.* PhD thesis, University of Twente, 2010.

[107] Marijn R. Jongerden and Boudewijn R. Haverkort. Which battery model to use? *IET Software*, 3(6):445–457, 2009.

[108] Marijn R. Jongerden, Alexandru Mereacre, Henrik C. Bohnenkamp, Boudewijn R. Haverkort, and Joost-Pieter Katoen. Computing optimal schedules for battery usage in embedded systems. *IEEE Trans. Industrial Informatics*, 6(3):276–286, 2010.

[109] Richard M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23(3):309 – 311, 1978.

[110] Richard M. Karp and Raymond E. Miller. Parallel program schemata. *J. Comput. Syst. Sci.*, 3(2):147–195, 1969.

[111] Mark Kattenbelt, Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Abstraction refinement for probabilistic software. In *VMCAI*, pages 182–197, 2009.

[112] Raimund Kirner. The wcet analysis tool calcwcet167. In *ISoLA (2)*, pages 158–172, 2012.

[113] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. r-tubound: Loop bounds for wcet analysis (tool paper). In *LPAR*, pages 435–444, 2012.

[114] Robert Könighofer, Georg Hofferek, and Roderick Bloem. Debugging formal specifications using simple counterstrategies. In *FMCAD*, pages 152–159, 2009.

[115] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In *CAV*, pages 1–35, 2013.

[116] Lawrence H. Landweber. Synthesis algorithms for sequential machines. In *IFIP Congress (1)*, pages 300–304, 1968.

[117] Kim G. Larsen. Priced timed automata: Theory and tools. In *FSTTCS*, pages 417–425, 2009.

[118] Kim G. Larsen, Simon Laursen, and Jirí Srba. Action investment energy games. In *MEMICS*, pages 155–167, 2012.

[119] Kim Guldstrand Larsen and Arne Skou. Bisimulation through probabilistic testing. In *POPL*, pages 344–352, 1989.

[120] Rüdiger Lincke, Jonas Lundberg, and Welf Löwe. Comparing software metrics tools. In *ISSTA*, pages 131–142, 2008.

[121] R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM J. Res. Dev.*, 6(2):200–209, 1962.

[122] Rupak Majumdar and Indranil Saha. Symbolic robustness analysis. In *RTSS*, pages 355–363, 2009.

[123] James F Manwell and Jon G McGowan. Lead acid battery storage model for hybrid energy systems. *Solar Energy*, 50(5):399–405, 1993.

[124] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In M. Sagiv, editor, *European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20. Springer-Verlag, 2005.

[125] Richard Mayr and Lorenzo Clemente. Advanced automata minimization. In *POPL*, pages 63–74, 2013.

[126] Robert McNaughton. Infinite games played on finite graphs. *Ann. Pure Appl. Logic*, 65(2):149–184, 1993.

[127] Robin Milner. An algebraic definition of simulation between programs. In *IJCAI*, pages 481–489, 1971.

[128] David Monniaux. Abstract interpretation of programs as markov decision processes. *Sci. Comput. Program.*, 58(1-2):179–205, 2005.

[129] Lawrence C. Paulson. The foundation of a generic theorem prover. *J. Autom. Reasoning*, 5(3):363–397, 1989.

[130] Ingo Pill, Simone Semprini, Roberto Cavada, Marco Roveri, Roderick Bloem, and Alessandro Cimatti. Formal analysis of hardware requirements. In *DAC*, pages 821–826, 2006.

[131] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. In *VMCAI*, pages 364–380, 2006.

[132] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.

[133] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *POPL*, pages 179–190, 1989.

[134] Amir Pnueli and Roni Rosner. On the synthesis of an asynchronous reactive module. In *ICALP*, pages 652–671, 1989.

[135] Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *LICS*, pages 32–41, 2004.

[136] Andreas Podelski and Andrey Rybalchenko. Transition predicate abstraction and fair termination. In *POPL*, pages 132–144, 2005.

[137] EJ Podlaha and HY Cheh. Modeling of cylindrical alkaline cells. *Journal of the Electrochemical Society*, 141(1):28–35, 1994.

[138] Adrian Prantl, Jens Knoop, Markus Schordan, and Markus Triska. Constraint solving for high-level wcet analysis. *CoRR*, abs/0903.2251, 2009.

[139] Adrian Prantl, Markus Schordan, and Jens Knoop. TuBound - a conceptually new tool for worst-case execution time analysis. In *WCET*, 2008.

[140] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Symposium on Programming*, pages 337–351, 1982.

[141] Daler N. Rakhmatov and Sarma B. K. Vrudhula. An analytical high-level battery model for use in energy management of portable electronic systems. In *ICCAD*, pages 488–493, 2001.

[142] Venkat Rao, Gaurav Singhal, Anshul Kumar, and Nicolas Navet. Battery model for embedded systems. In *VLSI Design*, pages 105–110, 2005.

[143] John H. Reif. The complexity of two-player games of incomplete information. *J. Comput. Syst. Sci.*, 29(2):274–301, 1984.

[144] David Romero-Hernández and David de Frutos-Escrig. Defining distances for all process semantics. In *FMOODS/FORTE*, pages 169–185, 2012.

[145] David Romero-Hernández and David de Frutos-Escrig. Distances between processes: A pure algebraic approach. In *WADT*, pages 265–282, 2012.

[146] David Romero-Hernández and David de Frutos-Escrig. Coinductive definition of distances between processes: Beyond bisimulation distances. In *FORTE*, pages 249–265, 2014.

[147] Shmuel Safra and Moshe Y. Vardi. On omega-automata and temporal logic (preliminary report). In *STOC*, pages 127–137, 1989.

[148] Roopsha Samanta, Jyotirmoy V. Deshmukh, and Swarat Chaudhuri. Robustness analysis of string transducers. In *ATVA*, pages 427–441, 2013.

[149] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ASPLOS*, pages 305–316, 2013.

[150] Simon Schubert, Dejan Kostic, Willy Zwaenepoel, and Kang G. Shin. Profiling software for energy consumption. In *GreenCom*, pages 515–522, 2012.

[151] Natarajan Shankar. A tool bus for anytime verification. Usable Verification, 2010.

[152] Aravinda Prasad Sistla. *Theoretical Issues in the Design and Verification of Distributed Systems*. PhD thesis, Harvard University, Cambridge, MA, USA, 1983. AAI8403047.

[153] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. Sketching concurrent data structures. In *PLDI*, pages 136–148, 2008.

[154] Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, pages 281–294, 2005.

[155] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

[156] Hsin-Hao Su, Chin Lung Lu, and Chuan Yi Tang. An improved algorithm for finding a length-constrained maximum-density subtree in a tree. *Inf. Process. Lett.*, 109(2):161–164, 2008.

[157] Paulo Tabuada, Ayca Balkan, Sina Y. Caliskan, Yasser Shoukry, and Rupak Majumdar. Input-output robustness for discrete systems. In *EMSOFT*, pages 217–226, 2012.

[158] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. Power analysis of embedded software: a first step towards software power minimization. In *ICCAD*, pages 384–390, 1994.

[159] Vivek Tiwari, Sharad Malik, Andrew Wolfe, and Mike Tien-Chien Lee. Instruction level power analysis and optimization of software. *VLSI Signal Processing*, 13(2-3):223–238, 1996.

[160] Viktor Vafeiadis and Francesco Zappa Nardelli. Verifying fence elimination optimisations. In *SAS*, pages 146–162, 2011.

[161] Franck van Breugel. An introduction to metric semantics: operational and denotational models for programming and specification languages. *Theor. Comput. Sci.*, 258(1-2):1–98, 2001.

[162] Franck van Breugel and James Worrell. An algorithm for quantitative verification of probabilistic transition systems. In *CONCUR*, pages 336–350, 2001.

[163] Franck van Breugel and James Worrell. A behavioural pseudometric for probabilistic transition systems. *Theor. Comput. Sci.*, 331(1):115–142, 2005.

[164] Franck van Breugel and James Worrell. Approximating and computing behavioural distances in probabilistic transition systems. *Theor. Comput. Sci.*, 360(1-3):373–385, 2006.

[165] Rob J. van Glabbeek. The linear time-branching time spectrum (extended abstract). In *CONCUR*, pages 278–297, 1990.

[166] Rob J. van Glabbeek. The linear time - branching time spectrum II. In *CONCUR*, pages 66–81, 1993.

[167] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Banff Higher Order Workshop*, pages 238–266, 1995.

[168] Martin T. Vechev, Eran Yahav, and Greta Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, pages 327–338, 2010.

[169] Sigal Weiner, Matan Hasson, Orna Kupferman, Eyal Pery, and Zohar Shevach. Weighted safety. In *ATVA*, pages 133–147, 2013.

[170] Reinhard Wilhelm, Sebastian Altmeyer, Claire Burguière, Daniel Grund, Jörg Herter, Jan Reineke, Björn Wachter, and Stephan Wilhelm. Static timing analysis for hard real-time systems. In *VMCAI*, pages 3–22, 2010.

[171] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

[172] Uri Zwick and Mike Paterson. The complexity of mean payoff games on graphs. *Theor. Comput. Sci.*, 158(1&2):343–359, 1996.