On Verification and Controller Synthesis for Probabilistic Systems at Runtime

by

Mateusz Ujma

A thesis submitted for the degree of
Doctor of Philosophy in Computer Science

Wolfson College, Oxford
Hilary Term 2015

Abstract

Probabilistic model checking is a technique employed for verifying the correctness of computer systems that exhibit probabilistic behaviour. A related technique is controller synthesis, which generates controllers that guarantee the correct behaviour of the system. Not all controllers can be generated offline, as the relevant information may only be available when the system is running, for example, the reliability of services may vary over time.

In this thesis, we propose a framework based on controller synthesis for stochastic games at runtime. We model systems using stochastic two-player games parameterised with data obtained from monitoring of the running system. One player represents the controllable actions of the system, while the other player represents the hostile uncontrollable environment. The goal is to synthesize, for a given property specification, a controller for the first player that wins against all possible actions of the environment player. Initially, controller synthesis is invoked for the parameterised model and the resulting controller is applied to the running system. The process is repeated at runtime when changes in the monitored parameters are detected, whereby a new controller is generated and applied. To ensure the practicality of the framework, we focus on its three important aspects: *performance*, *robustness*, and *scalability*.

We propose an *incremental model construction* technique to improve performance of runtime synthesis. In many cases, changes in monitored parameters are small and models built for consecutive parameter values are similar. We exploit this and incrementally build a model for the updated parameters reusing the previous model, effectively saving time.

To address robustness, we develop a technique called *permissive controller synthesis*. Permissive controllers generalise the classical controllers by allowing the system to choose from a set of actions instead of just one. By using a permissive controller, a computer system can quickly adapt to a situation where an action becomes temporarily unavailable while still satisfying the property of interest.

We tackle the scalability of controller synthesis with a *learning-based* approach. We develop a technique based on real-time dynamic programming which, by generating random trajectories through a model, synthesises an approximately optimal controller. We guide the generation using heuristics and can guarantee that, even in the cases where we only explore a small part of the model, we still obtain a correct controller.

We develop a full implementation of these techniques and evaluate it on a large set of case studies from the PRISM benchmark suite, demonstrating significant performance gains in most cases. We also illustrate the working of the framework on a new case study of an open-source stock monitoring application.

Acknowledgements

First and foremost I would like to thank my supervisors, Marta Kwiatkowska and Dave Parker. Without their support and guidance during my time in Oxford, this thesis would have never materialized. None of the research presented in this thesis would be possible without my Oxford collaborators with who I had a pleasure to work with. Those include Klaus Dräger, Vojtěch Forejt, and Hongyang Qu. When I was not occupied with research my lovely friends were always a source of great company. To name a few Aistis, Alistair, Daniel, Dipali, Frits, Gido, Henry, Kacper, Kamil, Kasia, Klemens, Matteo, Matt Bilton, Nathan, Penny, Radek, and Ruth. My interests in science and technology were greatly encouraged by my parents Ryszard and Elżbieta, who not only supported me but also showed a great interest in all aspects of my time in Oxford. I would also thank my two sisters Ania and Marta for always taking the time to care for me even when I am away. Lastly, I would like to thank my partner, Yvonne, for her support, encouragement, and understanding throughout my DPhil. This work has been supported by ERC Advanced Grant VERIWARE.

Rodzicom za wsparcie,
troskę i cierpliwość

Contents

1 Introduction **1**

2 Review of Related Work **5**

2.1 Probabilistic Model Checking 5
2.2 Quantitative Verification at Runtime 7
2.3 Incremental Techniques in Probabilistic Model Checking 8
2.4 Permissivity in Games 10
2.5 Learning-based Techniques for Probabilistic Models 11
2.6 Tools 12

3 Background Material **15**

3.1 Probabilistic Models 15
3.1.1 Discrete-Time Markov Chains 15
3.1.2 Markov Decision Processes 17
3.1.3 Stochastic Two-Player Games 19
3.2 Properties of Probabilistic Models 21
3.2.1 Probabilistic Reachability 21
3.2.2 Expected Total Reward 22
3.3 Model Checking and Controller Synthesis 23
3.4 Model Checking of Properties for Probabilistic Models 24
3.4.1 Model Checking of Properties for DTMCs 24
3.4.2 Model Checking of Properties for MDPs 26
3.4.3 Model Checking of Properties for Stochastic Games 31
3.5 The PRISM Modelling Language 33
3.5.1 Discrete-Time Markov Chains 34
3.5.2 Markov Decision Processes 35
3.5.3 Stochastic Games 35
3.6 Difference Bound Matrices 36
3.7 Mixed-Integer Linear Programming 40

3.8 Real-Time Dynamic Programming . . . 41

4 Verification and Controller Synthesis at Runtime 45
4.1 Components of the Framework . . . 46
4.1.1 Computer System . . . 46
4.1.2 Environment . . . 47
4.1.3 The Monitoring Module . . . 48
4.1.4 The Verification Module . . . 48
4.2 Key Characteristics of the Verification Module . . . 49
4.3 Summary . . . 50

5 Incremental Model Construction 51
5.1 Introduction . . . 51
5.2 Parameters in PRISM Models . . . 52
5.3 Non-Incremental Model Construction . . . 53
5.4 Incremental Model Construction . . . 56
5.5 Symbolic Representation . . . 60
5.6 Experimental Results . . . 62
5.7 Summary . . . 67

6 Permissive Controller Synthesis 69
6.1 Introduction . . . 69
6.2 Permissive Controller Synthesis . . . 70
6.2.1 Multi-Strategies . . . 70
6.2.2 Penalties and Permissivity . . . 71
6.2.3 Permissive Controller Synthesis . . . 73
6.3 MILP-Based Synthesis of Multi-Strategies . . . 76
6.3.1 Deterministic Multi-Strategies with Static Penalties . . . 76
6.3.2 Deterministic Multi-Strategies with Dynamic Penalties . . . 79
6.3.3 Approximating Randomised Multi-Strategies . . . 80
6.3.4 Optimisations . . . 83
6.4 Experimental Results . . . 85
6.5 Summary . . . 91

7 Learning-based Controller Synthesis 93
7.1 Introduction . . . 93
7.2 Learning-based Algorithms for Stochastic Games . . . 94
7.3 Stochastic Games without End Components . . . 95

7.3.1 Unbounded Reachability with BRTDP . . . 95
7.3.2 Step-bounded Reachability with BRTDP . . . 98
7.4 Stochastic Games with one-player ECs . . . 98
7.4.1 Identification and Processing of *one-player ECs* . . . 99
7.4.2 BRTDP and *one-player ECs* . . . 103
7.5 Experimental Results . . . 105
7.6 Summary . . . 113

8 Case Study and Implementation **115**
8.1 Introduction . . . 115
8.2 *StockPriceViewer* Application . . . 116
8.3 Controller Synthesis for *StockPriceViewer* . . . 117
8.3.1 Computer System . . . 118
8.3.2 Monitoring Module . . . 118
8.3.3 Verification Module . . . 118
8.4 Experimental Results . . . 119
8.4.1 Incremental Model Construction . . . 119
8.4.2 Learning-based Controller Synthesis . . . 122
8.4.3 Permissive Controller Synthesis . . . 126
8.5 Summary . . . 129

9 Conclusions **133**
9.1 Summary and Evaluation . . . 133
9.2 Future Work . . . 135
9.3 Conclusion . . . 137

A Benchmarks **139**
A.1 DTMC Case Studies . . . 139
A.2 MDP Case Studies . . . 139
A.3 Stochastic Game Case Studies . . . 141

B Proofs for Chapter 6 **143**
B.1 Proof of Theorem 4 . . . 143
B.2 Proof of Theorem 5 . . . 146
B.3 Proof of Theorem 6 . . . 148
B.4 Proof of Theorem 7 . . . 149
B.5 Proof of Theorem 8 . . . 152
B.6 Proof of Theorem 9 . . . 153

B.7 Proof of Theorem 10 155
B.8 Proof of Theorem 11 157

C Proofs for Chapter 7 **159**
C.1 Proof of Theorem 12 159
C.2 Proof of Lemma 4 160
C.3 Proof of Theorem 13 161

D PRISM model of *StockPriceViewer* Application **165**

List of Figures

3.1 Example of Discrete-Time Markov Chain (DTMC) D. 17
3.2 Example of a Markov-Decision Process (MDP) M. 19
3.3 Example of a Stochastic Two-Player Game (STPG) G. 21
3.4 The MDP M from Figure 3.2 with MECs collapsed. 28
3.5 PRISM model description of the DTMC from Figure 3.1. 35
3.6 PRISM model description of the MDP from Figure 3.2. 35
3.7 PRISM model description of the stochastic game from Figure 3.3. 36
3.8 DBM representation of a constraint $(x_1 \leqslant 0 \wedge x_2 > 1)$. 37
3.9 Conjunction of DBMs $\mathcal{D}^1$ and $\mathcal{D}^2$ (top), and the result DBM $\mathcal{D}^3$ (bottom). 38
3.10 DBM $\mathcal{D}$ and its complement $\overline{\mathcal{D}^1}$ and $\overline{\mathcal{D}^2}$. 39
3.11 CPLEX output for the MILP encoding from Example 17. 40
3.12 Example of an MDP M satisfying the assumptions (1), (2), and (3). 43

4.1 Verification and Controller Synthesis at Runtime. 47

5.1 PRISM model M with s_max as the undefined constant. 54
5.2 Stochastic games G^1(left) and G^2(right) for PRISM model from Figure 5.1. 55
5.3 Profiling non-incremental model construction. 56
5.4 DBM representation of the guard g_1 from Example 21. 61
5.5 Multiple runs of incremental model construction. 65
5.6 Profiling incremental model construction. 67

6.1 A stochastic game G for Example 22. 72
6.2 MILP encoding for deterministic multi-strategies with static penalties. . . . 77
6.3 MILP encoding for deterministic multi-strategies with dynamic penalties. . 77
6.4 MILP encoding for the model from Figure 6.1. 79
6.5 A node in the original game (left), and the corresponding nodes in the transformed game for approximating randomised multi-strategies (right). . 82
6.6 Counterexample for Theorem 10 in case of dynamic penalties. 82
6.7 An approximation gadget for state $\overline{s}$ from Figure 6.1 and Example 26. . . . 84

7.1 Stochastic game G for Example 27. 97
7.2 Stochastic game G with an EC. 99
7.3 Executing COLLAPSE function on a stochastic game G. 105
7.4 Influence of the length of the explored paths on the computation time . . . 112

8.1 *StockPriceViewer* application. 116
8.2 Verification and Controller Synthesis at Runtime. 117
8.3 Stochastic game model of *StockPriceViewer* application parameterised with t_1 from Figure 8.2. 120
8.4 Running incremental model construction for *StockPriceViewer*. 123
8.5 Running learning-based controller synthesis for *StockPriceViewer*. 125
8.6 Running permissive controller synthesis for *StockPriceViewer*. 127
8.7 Improved robustness of the *StockPriceViewer* application. 130

B.1 The game for the proof of Theorem 6. 148

D.1 PRISM model *android_3* of *StockPriceViewer* application supporting three different providers. 166
D.2 PRISM model *android_4* of *StockPriceViewer* application supporting four different providers (continued in Figure D.3). 167
D.3 PRISM model *android_4* of *StockPriceViewer* application supporting four different providers. 168
D.4 Rewards, penalties and the property for the model of *StockPriceViewer* application. 168

List of Tables

5.1 Experimental results for incremental model construction (Algorithm 3) for MDPs and DTMCs. 64
5.2 Experimental results for incremental model construction (Algorithm 3) for stochastic games. 66

6.1 Details of the models and properties used for experiments with deterministic multi-strategies, and execution times shown for classical (single) strategy synthesis. 86
6.2 Experimental results for synthesis of optimal deterministic multi-strategies. 86
6.3 Details of models and properties for approximating optimal randomised multi-strategies. 90
6.4 State space growth for approximating optimal randomised multi-strategies. 90
6.5 Experimental results for deterministic and approximated optimal randomised multi-strategies. 91

7.1 Details of models and properties for running the BRTDP algorithm. 107
7.2 Verification times using BRTDP (three different heuristics) and PRISM. . 108
7.3 Number of visited states using BRTDP (three different heuristics). 110
7.4 Number of reachable states under ε-optimal strategy using BRTDP (three different heuristics). 111

8.1 Size of the stochastic game model of the *StockPriceViewer* application. . . 121
8.2 Experimental results for incremental model construction for *StockPrice-Viewer*. 122
8.3 Experimental results for learning-based controller synthesis for *StockPrice-Viewer*. 125

List of Algorithms

1 Real-Time Dynamic Programming [9, 14] 42
2 *ConstructModel* 54
3 *ConstructModelIncr* 57
4 Learning algorithm (for stochastic game with no ECs) 96
5 Identification of one-player ECs for BRTDP 101
6 Identification and processing of one-player ECs 102
7 Learning algorithm (for stochastic game with one-player ECs) 104

Chapter 1

Introduction

Computer systems are prevalent in our daily lives. From smartphones in our pockets to cloud systems delivering services to a range of industries, we have grown dependent on the correct behaviour of computer systems. *Formal verification* is a branch of computer science that aims to provide formal guarantees about the correctness of computer systems. *Model checking* is one of the techniques offered by formal verification. In model checking, we are given a formal model that exhibits all possible behaviours of the system, at a certain level of abstraction, and a property that our system has to satisfy. The model checking problem is to exhaustively check that the model always satisfies the property of interest.

Probabilistic systems are an important class of computer systems. These are systems that use physical devices that may fail, communicate over lossy channels, or use protocols that take advantage of randomisation. Formally verifying that such systems behave correctly and reliably requires quantitative approaches, such as *probabilistic model checking*.

Probabilistic model checking is used to verify systems with stochastic behaviour. Systems are modelled as, for example, Markov chains, Markov decision processes, or stochastic games, and analysed algorithmically using numerical computation to establish their quantitative properties.

A closely related problem to probabilistic model checking is that of *controller synthesis* for probabilistic systems. This entails constructing a model of some entity that can be controlled (e.g., a robot, a cloud administrator or a machine) and its environment, formally specifying the desired behaviour of the system, and then generating, through an analysis of the model, a controller that will guarantee the required behaviour.

In many cases, the same techniques that underly probabilistic model checking can be used for controller synthesis. For example, we can model the system as a stochastic two-player game, specify a property and then apply probabilistic model checking. This yields a *strategy* (also called a policy) for the first player in a stochastic game, which instructs

the controller as to which action should be taken in each state of the model in order to guarantee that the property will be satisfied against all actions of the second player.

Recently [19, 52, 65, 66], probabilistic model checking has been applied to the verification of computer systems at runtime. Verification results are used to reconfigure the running system such that the system is steered to ensure a given quantitative property. In [20] the authors propose a framework comprising a computer system exhibiting probabilistic behaviour, a monitoring module that observes the system behaviour, and a reconfiguration component. A model of the system is parameterised using data from the monitoring module and the results of probabilistic model checking are forwarded to the reconfiguration module, which directs the system accordingly.

In this thesis, we focus on a variant of this problem, where we are interested in generating a controller at runtime that can be employed to steer the running system. In Chapter 4, we describe our framework for controller synthesis at runtime. The main contribution of this thesis addresses three different challenges of the controller synthesis at runtime. We focus on *performance*, *robustness* and *scalability* of the controller synthesis process.

We improve the performance of the controller synthesis process by developing an *incremental model construction* technique. Continuous verification of a running system often boils down to analysing parametric models that share similar structure. We exploit this property and propose a model construction technique that, given a model built for a certain set of parameters of the system, incrementally builds a new model for the new set of parameters. During the incremental step we often only have to visit a small subset of the original model, yielding considerable performance gains.

We address the robustness of the controller synthesis problem by proposing a type of synthesis that we call *permissive controller synthesis*. While applying the generated controller on the running system, it could be the case that some action specified by the controller might become temporarily unavailable. The permissive controller synthesis algorithm generates controllers that allow several actions at one time, so that, if one of them becomes unavailable, the system may switch to a different one while still ensuring the property is satisfied.

In order to enhance the scalability of the controller synthesis problem, we build on *learning-based methods* used in fields like planning or reinforcement learning. We synthesise a controller by generating random trajectories through the probabilistic model. These trajectories allow us to compute an approximation of the value of the property, while potentially only exploring a small part of the model. These techniques are known to work for models with a certain structure and for non-verification purposes. We develop new methods that give guarantees that the process converges to the correct values and are able to support a wider range of probabilistic models.

We aim to make the techniques presented in the thesis practical, and to this end have developed a full implementation for each method. All implementations are built on top of PRISM-games [31], an extension of the PRISM model checker [88]. PRISM is a probabilistic model checker widely used for analysis and verification of probabilistic systems. We evaluated our techniques on a range of case studies coming from the PRISM Benchmark Suite [89], as well as PRISM-based stochastic games coming from the publications [21, 30, 33, 100]. We decided to choose the PRISM Benchmark Suite as it is a widely accepted set of case studies used for evaluation of probabilistic model checking and controller synthesis methods. To evaluate our runtime framework we developed a new case study. The case study considers an open-source stock monitoring application called *StockPriceViewer*, for which we synthesise controllers at runtime that specify which stock information provider the application should use.

Layout of the Thesis

A review of existing literature is presented in Chapter 2. We outline the relevant prior work that has been done in the area, and discuss how the contributions in this thesis differ from the literature. In Chapter 3, we introduce the basic definitions that are used throughout this thesis. These, among others, comprise probabilistic models, property specification formalisms, model checking and controller synthesis, as well as techniques that are specific to the new methods that we developed in this thesis. Our framework for controller synthesis at runtime can be found in Chapter 4. We address the specific challenges of the framework in the following chapters. In Chapter 5, we present incremental model construction techniques. In Chapter 6, we introduce permissive controller synthesis, and, in Chapter 7, we describe learning-based controller synthesis. The case study that employs all mentioned techniques within the framework described in Chapter 4 can be found in Chapter 8. In Chapter 9, we summarise work completed in this thesis and propose new possible directions for future work.

Other Publications

The methods presented in Chapter 5, Chapter 6 and Chapter 7 have been published or submitted for publication as jointly-authored papers. The preliminary work on incremental model construction presented in Chapter 5 appeared as [61] (a longer Technical Report is available in [62]). In this thesis, we present an extended version of this work. Extensions over [61] include support for difference-bound matrices, a more efficient state storage method, and a more robust implementation. A paper that describes the usefulness of incremental methods and verification at runtime has been published as [95]. The author of

this thesis collaborated with the co-authors on developing the incremental model construction algorithm as well as proving its correctness. The implementation and experimental evaluation was solely done by the author. In [49, 50], the permissive controller synthesis method has been introduced. The author collaborated with the co-authors on developing the concept, and the corresponding problem formulation, while proofs of the theorems were done by Klaus Dräger and Vojtěch Forejt. The author was solely responsible for implementation and experimental evaluation. The approach to improve the scalability by means of learning-based methods has been published in [17]. The methods presented in [17] work both for the case where only partial information about the Markov decision process (MDP) structure is known, as well as the case of full information. The full information case has resulted from collaboration between the author and co-authors. The author took part in both developing the algorithms as well as extending the proofs for stochastic games. The original proofs were done for MDPs by the co-authors. The implementation and experimental evaluation of the method has been carried out solely by the author.

Chapter 2

Review of Related Work

In this chapter, we summarise the literature related to the topics considered in this thesis. The chapter is divided into several sections, where each section is devoted to a different aspect of the thesis. We start, in Section 2.1, with a brief review of the model checking and probabilistic model checking literature. In Section 2.2, we discuss available frameworks for quantitative verification at runtime. A literature review of incremental methods has been considered in Section 2.3. An overview of various notions of permissivity for games can be found in Section 2.4. Literature on learning-based methods has been summarised in Section 2.5. In Section 2.6, we discuss available tools for probabilistic model checking.

2.1 Probabilistic Model Checking

Since the early 1980s, model checking [39, 108, 7] has been an important technique employed for verifying the correctness of computer systems. Given a formal model of a computer system and a temporal logic property that the system should satisfy, the model checking problem asks if the system satisfies the given property. In recent years, a variety of model checkers [10, 37, 71, 73, 76, 80, 88] has been proposed for different types of computer systems. A model checker is a piece of software that exhaustively checks if the system satisfies the property by visiting each relevant state of the system model.

Two of the main temporal logics used for specifying properties of computer systems are Linear Time Logic (LTL) [105] and Computation Tree Logic (CTL) [40]. The main difference between them is how each logic perceives a run of the system. LTL formulae consider a run of the system where each element of the run has one possible successor (linear time). In comparison, in CTL, when considering a run, we take into account all possible successors (branching time). A model checking algorithms for LTL have been introduced in [113], and in [38] for CTL.

Many computer systems contain elements that exhibit probabilistic behaviour. Such systems can be modelled by annotating transitions between states with probabilities. Examples of probabilistic models include discrete-time Markov Chains (DTMCs), Markov decision processes (MDPs), and stochastic two-player games (stochastic games). For probabilistic models we distinguish between two main types of properties: *qualitative* and *quantitative* properties.

For *qualitative* properties, the aim is to verify if a temporal property holds with probability 1 or 0. This can done using graph algorithms as the transition probabilities do not influence the satisfaction. For DTMCs this has been considered in [44, 70, 97] and for MDPs in [45, 119]. MDPs represent a generalisation of DTMCs and include both probabilistic and non-deterministic behaviour.

For *quantitative* properties, we are interested in computing the probability with which a given property holds or whether it exceeds some bound. For DTMCs this has been considered in [44, 45] and for MDPs in [44, 106, 119]. To formalise quantitative properties of probabilistic systems, extensions of CTL have been proposed. Probabilistic Computation Tree Logic (PCTL) [70] is a temporal logic that adds a probabilistic operator to CTL, allowing us to express a probability bound that the property should satisfy. Model checking of PCTL properties can be done using techniques from [44, 45].

Modelling many types of computer systems requires a notion of cost or reward associated with system states and actions. In the context of probabilistic systems, costs and rewards have been used to model, for example, energy consumption [86], discrete time passage [93], or the number of collisions that happen during wireless communication [92]. When using rewards, we are typically interested in computing the total reward, which is defined as the sum of rewards achieved on a run of the system. Assuming we know how to compute the reward achieved on all runs of the system, we can consider an *expected total reward*. For DTMCs computing the expected total reward can be done by solving a set of linear equations [107], and for MDPs by solving a linear program [55].

For MDPs, which exhibit non-deterministic and probabilistic choices, a problem related to model checking – controller synthesis – can be considered. The *controller synthesis* problem is: given an MDP and a property, synthesise a controller that resolves the non-deterministic choices such that the property is always satisfied. There are many applications of controller synthesis, including control strategies for robots [96], power management strategies for hardware [60], or efficient PIN guessing attacks against hardware security modules [114]. Controller synthesis for PCTL properties has been considered in [6, 59] and can be done using model checking techniques for MDPs [44, 45].

A model that generalises MDPs by adding a second type of non-determinism is stochastic games. Introduced by Shapley [111], stochastic games divide the state space between

two players. In this thesis, we are interested in games where the actions of one player are controllable, while the action of the other player are deemed uncontrollable. Uncontrollable actions are used to model adversarial behaviour of the environment. The controller synthesis problem for stochastic games is to synthesise a controller for the first player such that the probabilistic reachability or expected total reward property is satisfied for any resolution of the second player actions. The problem is known to be in NP $\cap$ co-NP [41, 55].

There are numerous applications of stochastic games, including analysis of smart energy management [30], autonomous urban driving [34], or decision making in self-adaptive systems [26].

2.2 Quantitative Verification at Runtime

Modern computer systems often work in a dynamic environment and have to *adapt* to changes at runtime. To ensure the correct behaviour of adaptive systems, authors of [24] advocated the use of *formal methods at runtime*. The idea is to build and verify a system model at runtime. The model is parameterised with data from the running system and the results of verification can be used to adapt the system to changes in the environment. This concept is related to *models at runtime* [13] that suggests keeping a model of the system during system execution. For probabilistic systems, a similar method called *quantitative verification at runtime* [19] has been developed.

There are numerous cases when quantitative verification at runtime proved to be successful. Early work includes [25, 52], where the authors considered scenarios such as dynamic power management, cluster management, and web service orchestration. In [23], a telehealth service-based system has been considered, and in [66] a dynamic adaptation of webservice-based application. Cloud deployment and provisioning have been considered in [102]. Lastly, in [65], quantitative verification has been used in the context of unmanned underwater vehicles (UUV).

A prerequisite for a successful application of formal methods at runtime is having a precise model of the system. The *Keep Alive Models with Implementations* (KAMI) [52] framework uses quantitative verification for estimating the value of the system parameters at runtime. In KAMI, a model of the system is given as a DTMC, with some of its parameters unknown until the system is running. The authors observe the running system and use Bayesian estimation to obtain the value of the parameters. The computed values are used to parameterise the DTMC model, which is then verified for property violations.

In [20], the authors use quantitative verification at runtime in the context of a *Monitor Analyse Plan Execute* (MAPE) [83] loop. The MAPE loop formalises the adaptation process for computer systems. The authors provide a full implementation of each element

of the loop. KAMI is used in the monitoring phase to obtain an accurate model of the system. PRISM [88], a probabilistic model checker, is incorporated in the analysis and planning phases. More specifically, PRISM is employed to verify that the running system does not violate any properties. If results of the verification suggest that the system should change its behaviour, the output of PRISM can be used in the planning phase. Execution of the system adaptation is done using GPAC [18].

A slightly different approach has been presented in [57] by Filieri et al. Instead of using a MAPE loop, the framework controls system adaptations using a control-theoretic approach. A DTMC model of the running system is translated into a discrete-time dynamic system formalism [98]. This formalism is well known in control theory and methods for obtaining a controller for such systems exist. The authors consider a case study, where their approach is shown to improve the reliability of an image processing application.

2.3 Incremental Techniques in Probabilistic Model Checking

Kwiatkowska et al. [94] consider a scenario where transition probabilities in a probabilistic model change over time. This is a common scenario in a runtime setting. For example, hardware elements can degrade over time, causing changes in failure probabilities that are encoded in the system model. The authors present incremental methods that re-use the results of verification performed for old values of transition probabilities when re-verifying the model. The presented methods divide the state space of the model into strongly connected components (SCCs) and run verification for each SCC separately. When a change in transition probability happens, the subset of affected SCCs is computed and verification is carried out for each affected SCC. For all SCCs that are not affected we can work incrementally and re-use the old values, which leads to a significant speed-up.

In the context of robotic systems, incremental techniques have been studied in a series of papers [117, 118, 122, 123]. In [122], the authors present an incremental algorithm for synthesis of control strategies for probabilistic systems. Models are encoded as MDPs and properties are expressed in a subset of LTL known as co-safe LTL [85]. The authors consider the problem of a robot interacting with a dynamic number of independent agents. The incremental methods synthesise a control strategy for a model consisting of only a small number of agents. In the incremental step the number of agents is incremented and a new strategy is computed based on the results for the smaller model. The process is continued until the model includes all the agents or the available memory is exhausted. In [117, 118], the authors consider a similar scenario but develop a new and more effi-

cient solution method. While they still build the model of the robot and a set of agents incrementally, each step is followed by a state space reduction procedure. This results in a potential speed-up over the techniques presented in [122]. In [123], the authors generalise the results obtained in [117, 118, 122]. Instead of considering only a subset of LTL, methods supporting full LTL are shown. Moreover, the authors consider a more general scenario, where agents are allowed to affect each others behaviour.

In [58], authors consider a method for incremental verification at runtime. Properties to be verified are specified in PCTL, and probabilistic models contain parameters that are unknown until runtime. The method is run off-line and precomputes the probabilistic reachability value as a function of those parameters. Later, at runtime, when the value of parameters is known, the computed function can be evaluated and the probabilistic reachability value obtained. The evaluation time is typically very short, allowing the method to significantly decrease the verification time. In [56], the authors extend their approach to support verification of reward-based PCTL properties.

In [48], the authors present incremental methods for computing counterexamples for probabilistic models defined in the PRISM modelling language. The presented methods are based on computing the minimal subset of commands of the PRISM model such that the underlying probabilistic model defined by those commands violates the probabilistic reachability property. The problem is translated into a SAT encoding where the assignment to the encoding variables yields a possible counterexample. After obtaining a counterexample, PRISM is run to check if the counterexample violates the property. If the counterexample is proved spurious, a new encoding is constructed. This is done incrementally based on the previous encoding and disallows the spurious counterexample.

Our work, presented in Chapter 5, differs significantly from the methods we just described. In [94], the authors do not support structural changes to the model and assume that changes in probability values are always between 0 and 1. In [56, 58], structural changes are possible but are limited to a small number of states and need to be known at design time. In comparison, we focus on structural changes that can affect a large numbers of states and are not known until runtime. The work presented in [56, 58] works on models consisting of up to a thousand states, whereas our methods can work with models that contain millions of states. In [117, 118, 122, 123], the authors assume a variable number of agents that are part of the system model. In the PRISM modelling language, such agents can be described using modules. In our work, we assume a fixed number of modules and focus on changes to model parameters which are assumed to be fixed in [117, 118, 122, 123]. Similarly to our work, the authors of [48] consider the PRISM modelling language. However, they do not run their method in a runtime scenario, and assume that all model parameters are known and do not change.

2.4 Permissivity in Games

In [11], the authors consider synthesis of permissive strategies for parity games. A winning strategy is considered to be more permissive if it allows more actions than some other strategy. The authors motivate such a definition by suggesting that a permissive strategy may be useful in case of temporary unavailability of the controller's actions. It is shown that permissive strategies exist in every parity game and require only finite memory. Subsequently, a synthesis method for permissive strategies is presented. This is achieved by providing a reduction from computing the most permissive strategy in a parity game into finding the most permissive strategy in a safety game.

Permissive strategies for turn-based two-player games and reachability objectives have been considered in [15]. Similarly to [11], the authors assume that a permissive strategy will allow more behaviour than a classical strategy. The authors quantify the permissivity by introducing a notion of penalties. A penalty is incurred every time an action in a game is disallowed. Three methods of aggregating penalties are shown: summing the penalties, computing the discounted sum, and computing the mean value. In the case of sum of penalties, synthesizing the optimally permissive memoryless strategy can be done in PTIME. For discounted sum, synthesising the optimally permissive strategy is in NP $\cap$ co-NP. In [16], the authors extend the results of [15] to the setting of parity games.

Our work on permissive controller synthesis from Chapter 6 significantly differs from methods that we found in the literature. In contrast to [11, 15, 16] that focuses on non-stochastic games and deterministic strategies, we work in a stochastic games setting and develop methods for synthesizing randomised strategies. In [15], the authors show that optimally permissive strategies exist for reachability objectives and expected penalties; in contrast, in our setting they may not. One of the aims of our work was to develop an implementation of our algorithms. None of the publications [11, 15, 16] provide a practical implementation.

To synthesise permissive strategies for stochastic games in Chapter 6 we developed a Mixed-Integer Linear Programming (MILP) encoding of our problem. In [121], while solving a different problem of finding the minimal critical subsystem of an MDP, the authors present an MILP encoding that is similar to ours. Minimal critical subsystem is the smallest set of states of an MDP that violates the property. Such a set of states can be returned to represent a counterexample refuting the property. The encoding in [121] is constructed based on the explicit state representation of the MDP. The constraints encode the probabilistic reachability property, while the objective function ensures that a minimal critical subsystem is computed. The methods are evaluated on a set of PRISM case studies and CPLEX [131] has been used as one of the MILP solvers.

The MILP encoding presented in [121] considers MDPs, a less general model than the stochastic games we work with. Additionally, in our encoding, we work with properties defined as expected total reward, while in [121] the encoding supports only reachability properties. Lastly, we study a different problem of permissive strategy synthesis, whereas in [121] the authors study minimal critical subsystem generation.

2.5 Learning-based Techniques for Analysis of Probabilistic Models

Learning-based methods have a long history in fields such as planning [99], reinforcement learning [115], and optimal control [12]. Real-Time Dynamic Programming (RTDP) [9] is one example of the methods studied in those fields that are relevant to our work in Chapter 7. RTDP is a method for computing the expected reward for so called stochastic shortest path MDPs [12]. The method is based on generating a series of random trajectories through an MDP, where each trajectory contributes to computing the value of the expected reward. During the run, RTDP produces a series of values that constitute a lower bound of the value of the expected reward. Given enough runs, RTDP is proven to converge to the correct value despite the fact that it may only visit a small subset of the states. Guiding the trajectory generation is done using a heuristic which picks states that have a high probability of being reached.

RTDP has been extended in numerous ways, with two extensions being the most prominent. Labelled RTDP (LRTDP) [14] defines a method of labelling states that speeds up the convergence of RTDP. A state is labelled as solved if the value of the expected reward has converged in that state. If a random trajectory visits such a state, the computation of RTDP is stopped for that trajectory, resulting in a speed-up. The authors of [101] suggest another extension of RTDP called Bounded RTDP (BRTDP). The method keeps both lower and upper values of the expected reward in a state. By keeping those values, we always know the precise interval in which the value of the expected reward lies. In BRTDP, the size of the interval is also used for guiding the trajectory generation. States with a larger gap between the lower and upper bounds will be visited more often, resulting in quicker convergence.

One of the main limitations of algorithms based on RTDP is the fact that they are only guaranteed to converge on a subclass of MDPs called stochastic shortest path MDPs. In [84], the authors studied a more general class of MDPs called Generalised Stochastic Shortest Path (GSSP). The main difference between those two models is that GSSP allow MDPs that contain cycles. None of the previously mentioned algorithms converges on an

MDP with cycles, as randomly generated trajectories can get stuck in loops, forbidding the termination of the algorithm. The authors of [84] propose a new algorithm called FRET. The algorithm does a similar random exploration as RTDP but is able to eliminate the cycles on-the-fly. FRET is compared to the value iteration, and outperforms it both in terms of speed and memory consumption.

Heuristics that are used by RTDP-based algorithms, as well as those considered in Chapter 7, play an important role in obtaining good practical results. For the problem of counterexample generation for probabilistic models, heuristic-based state space exploration has been considered in [1, 2, 4]. In [4], the authors used an extension of the *k-shortest-path* algorithm to generate paths that constitute a counterexample. The authors generate an MDP on-the-fly and use a heuristic to guide the search into parts of the MDP that are more likely to contain a counterexample. The methods presented in [1, 2, 4], have been implemented in [3].

The methods developed in Chapter 7 are based on the RTDP [9] and BRTDP [101] algorithms. Similarly to BRTDP, we provide both upper and lower bounds on the value of the property and we use heuristics defined for RTDP in the experimental section. The main difference between our work and RTDP and BRTDP is that our method supports arbitrary MDPs, whereas [9, 101] only work with stochastic shortest path MDPs. In contrast to FRET [84], we support both upper and lower bounds and provide a range of heuristics for guiding the trajectory generation. The work presented in [1, 2, 4] uses heuristics similarly to this thesis, but focuses on counterexample generation.

2.6 Tools

Several model checkers have been proposed for verification of probabilistic models. The two most popular include PRISM [88] and MRMC [80]. In this thesis, all presented methods have been implemented as an extension of PRISM. We discuss PRISM later in this section. The MRMC [80] model checker has been developed for verification of DTMCs and continuous-time Markov chains (CTMCs). The input of MRMC is a file explicitly specifying the transition probability function. Properties are defined in logics PCTL and CSL logics extended with a reward operator. Other model checkers include LiQuor [35], which is able to verify MDPs specified in the probabilistic extension of the PROMELA language called PROBMELA [5]. The properties are defined as ω-regular linear time properties. For parametric DTMCs, PARAM has been developed [68]. The input language of PARAM is an extension of PRISM modelling language and the tool allows verifying probabilistic reachability properties. PARAM has been incorporated in a recent PRISM 4.2.1 release [32, 46].

PRISM [88] allows model checking of several types of probabilistic models, including DTMCs, CTMCs, MDPs, and Probabilistic Timed Automata (PTAs). Properties can be specified in PCTL, CSL, LTL, and PCTL*, where each logic can be extended with a reward operator. PRISM has multiple internal implementations called *engines*. Available engines include the "mtbdd" engine that uses symbolic (BDD-based) data structures, the "explicit", and the "sparse" engines that use explicit-state data structures, as well as the "hybrid" engine that uses a mix of symbolic and explicit representation. Several well known model checking techniques are available within PRISM, to name a few, quantitative abstraction refinement [81], statistical model checking [74, 124], and multi-objective verification [60]. There exist numerous case studies where PRISM proved its usefulness. These include formal analysis of the Bluetooth protocol [51], analysis of biological pathways [72], strategy synthesis for an efficient PIN guessing attack [114], and many more [87, 90, 91, 93].

All methods that we present later in this thesis have been developed as part of PRISM-games [31], an extension of PRISM that supports turn-based stochastic games. The games analysed by PRISM-games are defined in an extension of the PRISM modelling language that allows specification of players, as well as of which actions those players control. Properties are given using an extension of the logic ATL called rPATL [30]. The rPATL logic supports specification of reward-based properties, where each property is preceded with information about which player will try to satisfy the property. Internally, PRISM-games uses PRISM's "explicit" engine to implement model checking algorithms for stochastic games. Case studies analysed by PRISM-games include smart energy management [30], autonomous urban driving [34], and collective decision making in sensor networks [30].

Chapter 3

Background Material

In this chapter, we introduce the basic background information that is needed throughout the thesis. We start in Section 3.1 by defining the probabilistic models that we work with. For completeness we present three models, where each model is a generalisation of the previous one. In Section 3.2, we describe a formalism to reason about the properties of those models. In Section 3.3, we introduce controller synthesis and show how it relates to model checking. The background information about probabilistic model checking, i.e. the process of checking that a given property holds for a probabilistic model, is presented in Section 3.4. The PRISM modelling language for describing probabilistic models can be found in Section 3.5. Starting in Section 3.6, we describe techniques and data structures that do not originate from probabilistic model checking but are used by the methods that we have developed. This includes Difference Bound Matrices, Mixed-Integer Linear Programming, and learning-based MDP solution methods.

3.1 Probabilistic Models

We start by introducing the notation used throughout this chapter. We use $\mathbb{N}$, $\mathbb{Q}_{\geqslant 0}$, and $\mathbb{R}_{\geqslant 0}$ to denote the sets of all non-negative integers, rational numbers, and real numbers, respectively. $\mathit{Dist}(X)$ is the set of all rational probability distributions over a finite or countable set X, i.e., the functions $f : X \to [0,1] \cap \mathbb{Q}_{\geqslant 0}$ such that $\sum_{x \in X} f(x) = 1$, and $\mathit{supp}(f)$ denotes the *support* of f. We call f a *Dirac* distribution if $f(x) = 1$ for some $x \in X$.

3.1.1 Discrete-Time Markov Chains

Discrete-Time Markov Chains (DTMCs) are the simplest probabilistic model considered in this thesis. Formally, a DTMC D is a tuple $\mathsf{D} = \langle S, \overline{s}, \Delta, \mathcal{L} \rangle$, where:

- S is a finite set of states
- $\overline{s} \in S$ is the initial state
- $\Delta : S \times S \to [0,1]$ is the transition probability function
- $\mathcal{L} : S \to 2^{AP}$ is the labelling function.

The transition probability function Δ, given states $s, s' \in S$ as arguments, returns the probability of transitioning from state s to state s'. We require that such probabilities sum up to 1, i.e. for any state $s \in S$ we have that $\sum_{s' \in S} \Delta(s, s') = 1$. A state with only one successor such that $\Delta(s, s) = 1$ is called a *terminal* state. The set AP consists of atomic propositions that can be used to label the states. We use the function $\mathcal{L}$ to define the labelling. The function $Sat : AP \to 2^S$ returns the set of states labelled with a given atomic proposition.

A path through DTMC D, representing a possible execution of the system that D models that starts in a state $s \in S$, is a (finite or infinite) sequence $\omega = s_0 s_1 s_2 \ldots$ where $s_0{=}s$, and $\Delta(s_i, s_{i+1}) > 0$ for all $i \geqslant 0$. To reference the i-th element of the path we use $\omega(i)$, for example $\omega(1) = s_1$. We denote by $IPath_s$ ($FPath_s$) the set of all infinite (finite) paths starting in s. We omit the subscript s when s is the initial state $\overline{s}$, that is, we use just $IPath$ and $FPath$ for $IPath_{\overline{s}}$ and $FPath_{\overline{s}}$. For an infinite path ω, by $Prefix(\omega, n)$ we denote the finite prefix of ω of length n, i.e. $Prefix(\omega, 2) = s_0 s_1$. We use $last(\omega)$ to denote the last state of a finite path ω.

To reason about the probabilities of the paths in a DTMC, we work with a standard definition of probability measure [82]. For each state $s \in S$, we define a probability measure $Pr_{\mathsf{D},s}$ on $IPath_s$. We start by formalizing the probability of a finite path. Given a finite path $\omega = s_0 s_1 \ldots s_n$, we define the probability of the path $\Delta(\omega) = \Delta(s_0, s_1)\Delta(s_1, s_2) \cdot \cdots \cdot \Delta(s_{n-1}, s_n)$.

A *cylinder set* $Cyl(\omega)$ is the set of all infinite paths with a finite prefix ω. The family of sets of infinite paths Σ_s is the smallest σ-algebra defined on $IPath_s$ that contains cylinder sets $Cyl(\omega)$ for all finite paths ω starting in s. Probability measure $Pr_{\mathsf{D},s}$ on Σ_s is now defined as the unique measure with $Pr_{\mathsf{D},s}(Cyl(\omega)) = \Delta(\omega)$.

A *bottom strongly connected component* (BSCC) of a DTMC D is a set of states $S' \subseteq S$ such that:

- if $\Delta(s, s') > 0$ for some $s \in S'$, then $s' \in S'$; and
- for all $s, s' \in S'$ there is a path $\omega = s_0 s_1 \ldots s_n$ such that $s_0 = s$, $s_n = s'$.

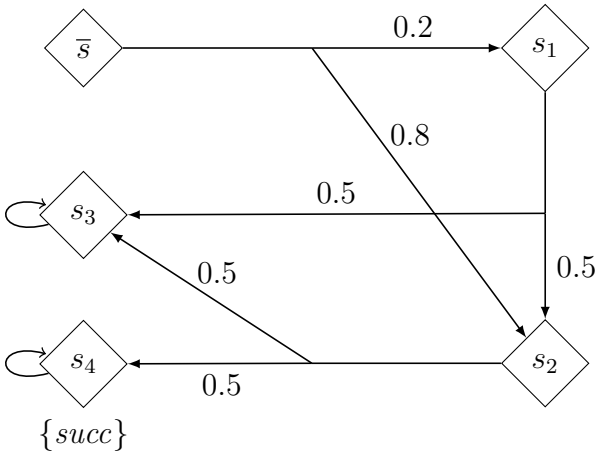


Figure 3.1: Example of Discrete-Time Markov Chain (DTMC) D.

Example 1 The DTMC D from Figure 3.1 consists of five states $S = \{\overline{s}, s_1, s_2, s_3, s_4\}$, with the initial state $\overline{s}$. In the initial state, the transition probability function returns $\Delta(\overline{s}, s_1) = 0.2$ and $\Delta(\overline{s}, s_2) = 0.8$. States s_3 and s_4 are considered as terminal states as well as BSCCs because they only have self-loops. We mark s_4 with atomic proposition *succ* and the labelling function returns $\mathcal{L}(s_4) = \{succ\}$. Two possible executions of the system can be represented by the two finite paths $\omega_1 = \overline{s}s_1s_2s_4$ and $\omega_2 = \overline{s}s_2s_3$, where the probability of the paths is $\Delta(\omega_1) = 0.2 \cdot 0.5 \cdot 0.5 = 0.05$ and $\Delta(\omega_2) = 0.8 \cdot 0.5 = 0.4$. An infinite path $\omega_3 = \overline{s}s_2s_4s_4s_4\ldots$ visits the state marked with *succ* infinitely often. A finite prefix of ω_3 yields probability $Pr_{\mathsf{D},\overline{s}}(Cyl(\overline{s}s_2s_4)) = 0.8 \cdot 0.5 = 0.4$.

3.1.2 Markov Decision Processes

Markov Decision Processes (MDPs) are probabilistic models that describe both probabilistic and non-deterministic behaviour. MDPs can be considered a more general model than DTMCs. Non-determinism is typically used to capture concurrency, adversarial behaviour of the environment, or to represent actions that are under the system control.

An MDP M is a tuple $\mathsf{M} = \langle S, \overline{s}, A, \delta, \mathcal{L}\rangle$, where:

- S is a finite set of states
- $\overline{s} \in S$ is the initial state
- A is a finite set of actions
- $\delta : S \times A \to Dist(S)$ is the (partial) transition probability function
- $\mathcal{L} : S \to 2^{AP}$ is the labelling function.

We define the set S, the initial state $\overline{s}$ and the labelling function $\mathcal{L}$ in the same way as for DTMCs. The transition probability function Δ is now replaced by δ. Given a state $s \in S$ and action $a \in A$, δ returns a distribution over the set of successors of s. Each state s of an MDP has a set of *enabled* actions, given by $A(s) = \{a \in A \mid \delta(s,a) \text{ is defined}\}$. We assume that $|A(s)| \geqslant 1$ for every state s. A state s is *terminal* if all actions $a \in A(s)$ satisfy $\delta(s,a)(s) = 1$.

For simplicity of presentation we assume that all actions in the model are *unique*. For every action $a \in A$, there is at most one state s such that $a \in A(s)$, i.e., $A(s) \cap A(s') = \emptyset$ for $s \neq s'$. If there *are* states s, s' such that $a \in A(s) \cap A(s')$, we can always rename the actions as $(s, a_1) \in A(s)$, and $(s', a_2) \in A(s')$, so that the MDP satisfies our assumption.

An *infinite path* through an MDP M is an infinite sequence $\omega = s_0 a_0 s_1 a_1 \cdots$ such that $a_i \in A(s_i)$ and $\delta(s_i, a_i)(s_{i+1}) > 0$ for every $i \geqslant 0$. Similarly to DTMCs, a *finite path* is a finite prefix of an infinite path ending in a state. We use the same notation as for DTMCs to denote sets of all infinite (finite) paths.

Before we can define the probability measure on the paths in an MDP, we need to resolve the non-determinism coming from multiple actions available in a state. A *strategy* (also called controller, adversary, or policy) is a function $\sigma : \mathit{FPath} \to \mathit{Dist}(A)$ that, based on the execution so far, resolves the choices between actions in each state.

A strategy σ is *deterministic* if $\sigma(\omega)$ is a Dirac distribution for all ω, and *randomised* otherwise. A strategy that only depends on the last state of ω is called a *memoryless* strategy, and *history-dependent* otherwise. For memoryless strategies, we simplify the function σ to $\sigma : S \to \mathit{Dist}(A)$. We write Σ_{M} for the set of all (memoryless) strategies in M.

The strategy σ and initial state $\overline{s}$ induce a possibly infinite DTMC on which we can define the probability measure over IPath_s in the standard fashion [8]. We denote the probability measure as $Pr^{\sigma}_{\mathsf{M},s}$. As above, we simply write Pr^{σ}_{M} for $Pr^{\sigma}_{\mathsf{M},\overline{s}}$.

An *end component* (EC) of an MDP M is a pair (S', A') where $S' \subseteq S$ and $A' \subseteq \bigcup_{s \in S'} A(s)$ such that:

- if $\delta(s,a)(s') > 0$ for some $s \in S'$ and $a \in A'$, then $s' \in S'$; and
- for all $s, s' \in S'$ there is a path $\omega = s_0 a_0 \ldots s_n$ such that $s_0 = s$, $s_n = s'$ and for all $0 \leqslant i < n$ we have $a_i \in A'$.

A *maximal end component* (MEC) is an EC that is maximal with respect to the point-wise subset ordering. We call an EC a *non-trivial EC* if it contains more than one state.

Example 2 Throughout the rest of the thesis, we consistently use examples drawn from one theme. We assume a simple model, where a robot or multiple robots move between

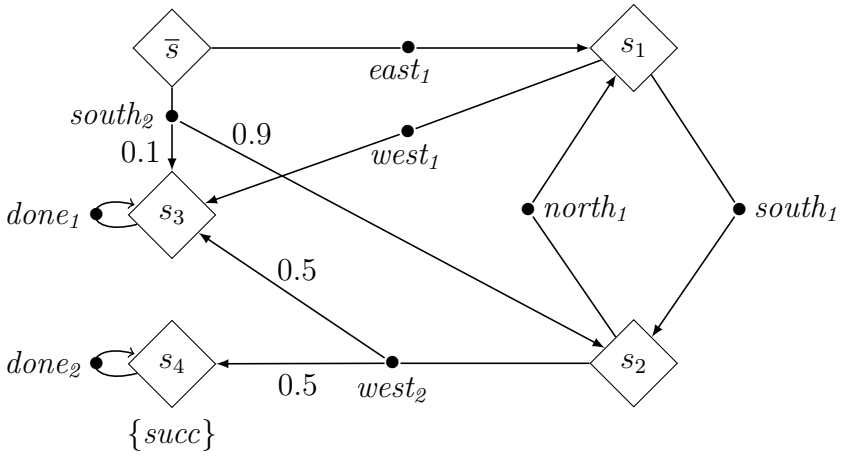


Figure 3.2: Example of a Markov-Decision Process (MDP) M.

locations. Movements are controlled using actions that are marked with geographical directions (east, west, north, south). Each action may have a distribution on the possible destination locations, representing uncertainty in the robot movements.

The MDP M from Figure 3.2 consists of five states $S = \{\overline{s}, s_1, s_2, s_3, s_4\}$; $\overline{s}$ is the initial state, and the labelling function $\mathcal{L}$ returns $\mathcal{L}(s_4) = \{succ\}$. In contrast to DTMCs, in some states we now have a choice between multiple actions. In the state s_1, the robot may move either in the $south_1$ or $west_1$ direction; in other words, $A(s_1) = \{south_1, west_1\}$. The states s_3 and s_4 are the only terminal states in M.

Let us consider a memoryless deterministic strategy σ such that $\sigma(\overline{s}) = east_1$, $\sigma(s_1) = south_1$, $\sigma(s_2) = west_2$, $\sigma(s_3) = done_1$ and $\sigma(s_4) = done_2$. One possible execution of the system under σ yields an infinite path $\omega = \overline{s}east_1 s_1 south_1 s_2 west_2 s_4 done_2 s_4 \ldots$. The probability of such a path would be $Pr^{\sigma}_{\mathsf{M},\overline{s}}(Cyl(Prefix(\omega, 4))) = 1 \cdot 1 \cdot 0.5 = 0.5$. The MDP M contains one non-trivial EC formed by the pair $(\{s_1, s_2\}, \{south_1, north_1\})$.

3.1.3 Stochastic Two-Player Games

The last probabilistic model that we consider in this thesis are turn-based *Stochastic Two-Player Games* (STPGs) or simply stochastic games. Similarly to MDPs, which can be considered a generalisation of DTMCs, stochastic games can be considered a generalisation of MDPs. In contrast to MDPs, in stochastic games the state space is divided between two players and each of the players controls actions in their states.

The stochastic game G is a tuple $\mathsf{G} = \langle S_{\Diamond}, S_{\Box}, S, \overline{s}, A, \delta, \mathcal{L}\rangle$, where:

- $S_{\Diamond}$ is a finite set of controller states
- $S_{\Box}$ is a finite set of environment states

- $S = S_\Diamond \cup S_\Box$ is the finite set of all states
- $\overline{s} \in S$ is the initial state
- A is a finite set of actions
- $\delta : S \times A \to Dist(S)$ is the (partial) transition probability function
- $\mathcal{L} : S \to 2^{AP}$ is the labelling function.

$S_\Diamond$ is a finite set of states controlled by the first player; we will call this player the *controller* player. $S_\Box$ is a finite set of states controlled by the second player; we call the second player the *environment* player. The set S denotes the set of all states. The definitions of the initial state, the transition probability function, the set of actions, the labelling function, and terminal states are identical to the ones for MDPs. Similarly to MDPs, we will also assume that actions in the stochastic game are *unique*. Please note that, in the case where $S_\Box = \emptyset$, the stochastic game G becomes an MDP. We use the same definition and notation as for MDPs to denote the set of finite and infinite paths in a stochastic game.

In the case of MDPs, we have only one strategy σ; for stochastic games we define a strategy for each player. The strategy of the controller player is described using a function $\sigma : FPath \to Dist(A)$, and we use $\pi : FPath \to Dist(A)$ for the strategy of the environment player. Similarly to MDPs, σ or π can be deterministic or randomised, and memoryless or history-dependent. We write $\Sigma^\circ_{\mathsf{G}}$ for the set of all (memoryless) strategies of player $\circ$ in G, where $\circ \in \{\Box, \Diamond\}$.

A pair of strategies (σ, π) and a state s induce a possibly infinite DTMC on which we can define the probability measure over $IPath_s$ in the same way as for MDPs and DTMCs. We denote the probability measure as $Pr^{\sigma,\pi}_{\mathsf{G},s}$. As for previous models, we write $Pr^{\sigma,\pi}_{\mathsf{G}}$ for $Pr^{\sigma,\pi}_{\mathsf{G},\overline{s}}$.

We re-use the definition of an end component, maximal end component and non-trivial end component as presented for MDPs. To determine if all states in an end component belong to one player, we define the $player()$ function. The function takes a set of states as an input and returns the player that owns those states. Formally, $player(\{s_1, s_2, s_3\})$ returns $\Diamond$ (or $\Box$) if *all* the states belong to the $\Diamond$ ($\Box$ respectively) player and $\perp$ otherwise.

Example 3 We extend the single robot example from Example 2 to a two-robot scenario modelled using stochastic games. The first robot tries to move to a location labelled with the *succ* label, while the other robot may either allow it to pass or block the movement.

The stochastic game G from Figure 3.3 consists of five states with $\overline{s}$ as the initial state. The controller player controls the states from the set $S_\Diamond = \{\overline{s}, s_2, s_4\}$, whereas the

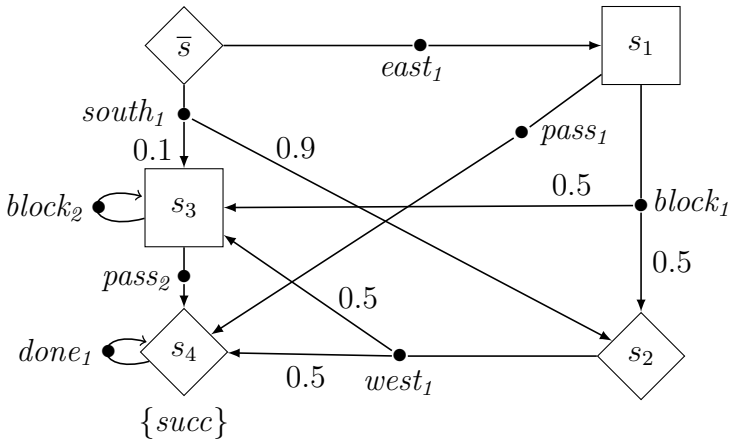


Figure 3.3: Example of a Stochastic Two-Player Game (STPG) G.

environment player controls $S_\Box = \{s_1, s_3\}$. In the initial state, the controller player can choose whether to go east or south, i.e. $A(\overline{s}) = \{east_1, south_1\}$. The environment player in the state s_1 can either decide to block or allow the movement of the first robot, i.e. $A(s_1) = \{pass_1, block_1\}$. The state s_4 is the only terminal state.

An example of a pair of controller and environment memoryless deterministic strategies is (σ, π) where: $\sigma(\overline{s}) = east_1$, $\sigma(s_2) = west_1$, $\sigma(s_4) = done_1$, and $\pi(s_1) = block_1$, $\pi(s_3) = block_2$. A possible infinite path under (σ, π) is $\omega = \overline{s} east_1 s_1 block_1 s_2 west_1 s_4 done_1 s_4 \ldots$. The probability of the path ω would be $Pr^{\sigma,\pi}_{\mathsf{G},\overline{s}}(Cyl(Prefix(\omega, 4))) = 1 \cdot 0.5 \cdot 0.5 = 0.25$.

3.2 Properties of Probabilistic Models

The purpose of this section is to formalise properties that we use to describe the required behaviour of the controllers. In this thesis, we focus on two types of properties, *probabilistic reachability* and *expected total reward*, that are defined over the models from Section 3.1.

3.2.1 Probabilistic Reachability

Reachability is the most basic type of property typically considered for probabilistic models. Given a DTMC D, a set of target states $T \subseteq S$, and the initial state $\overline{s} \in S$, we denote the probabilistic reachability property by $\phi = \mathtt{P}_{\bowtie p}\,[\,\mathtt{F}\ T\,]$, with a bound $p \in [0, 1]$ and an operator $\bowtie \in \{\leqslant, \geqslant, <, >\}$, meaning that, in the DTMC D, the probability of reaching any state from the set T, having started from the initial state $\overline{s}$, satisfies $\bowtie p$.

To check if the property is satisfied, we need to compute the value of probabilistic reachability. Formally, we denote this value by $\mathtt{P}_{=?}\,[\,\mathtt{F}\ T\,]$, where $\mathtt{P}_{=?}\,[\,\mathtt{F}\ T\,] = Pr_{\mathsf{D},\overline{s}}(\{s_0 s_1 s_2 \cdots \in IPath_{s_0} \mid s_i \in T \text{ for some } i \text{ and } s_0 = \overline{s}\})$.

A step-bounded probabilistic reachability property is denoted by $\mathtt{P}_{\bowtie p}\,[\,\mathtt{F}^{\leqslant k}\ T\,]$, where $k \in \mathbb{N}$, meaning that, in the DTMC D, the probability of reaching any state from the set T within k steps from the initial state $\overline{s}$ satisfies $\bowtie p$. As previously, we define $\mathtt{P}_{=?}\,[\,\mathtt{F}^{\leqslant k}\ T\,] = Pr_{\mathsf{D},\overline{s}}(\{s_0 s_1 s_2 \cdots \in IPath_{s_0} \mid s_i \in T \text{ for } i \leqslant k \text{ and } s_0 = \overline{s}\})$.

For MDPs and stochastic games, the probabilistic reachability value will depend on the strategies that are used to resolve the non-determinism. To check if property $\phi = \mathtt{P}_{\bowtie p}\,[\,\mathtt{F}\ T\,]$ is satisfied, we will need to compute either the maximal or the minimal value of probabilistic reachability.

Formally, we define the maximal probabilistic reachability value as $\mathtt{P}_{\max=?}\,[\,\mathtt{F}\ T\,] = \sup_{\sigma\in\Sigma_{\mathsf{M}}}\{Pr^{\sigma}_{\mathsf{M},\overline{s}}(\{s_0 a_0 s_1 a_1 \cdots \in IPath_{s_0} \mid s_i \in T \text{ for some } i \text{ and } s_0 = \overline{s}\})\}$. For example, to check if $\mathtt{P}_{\leqslant p}\,[\,\mathtt{F}\ T\,]$ is satisfied, it is enough to check if $\mathtt{P}_{\max=?}\,[\,\mathtt{F}\ T\,] \leqslant p$. The definition of the minimal value of probabilistic reachability can be constructed similarly.

For a stochastic game G, we denote the maximal value of probabilistic reachability $\mathtt{P}_{max=?}\,[\,\mathtt{F}\ T\,] = \sup_{\sigma\in\Sigma^{\Diamond}_{\mathsf{G}}} \inf_{\pi\in\Sigma^{\Box}_{\mathsf{G}}}\{Pr^{\sigma,\pi}_{\mathsf{G},\overline{s}}(\{s_0 a_0 s_1 a_1 \cdots \in IPath_{s_0} \mid s_i \in T \text{ for some } i \text{ and } s_0 = \overline{s}\})\}$. The definition of the minimal value of probabilistic reachability follows similarly.

For both MDPs and stochastic games, we also consider step-bounded probabilistic reachability properties denoted by $\mathtt{P}_{\bowtie p}\,[\,\mathtt{F}^{\leqslant k}\ T\,]$. Definitions of those properties can be obtained in a straightforward manner from the definitions of the unbounded properties.

Sometimes, instead of the target set T, we use an atomic proposition that marks the target states, for example, for an atomic proposition *succ* we write $\mathtt{P}\,[\,\mathtt{F}\ \mathit{succ}\,]$, meaning $\mathtt{P}\,[\,\mathtt{F}\ \mathit{Sat}(\mathit{succ})\,]$.

An example of a probabilistic reachability property can be "a web server successfully delivers a response with probability at least 0.999" or "the probability of the robot reaching a dead-end is lower than 0.001".

3.2.2 Expected Total Reward

The second type of property that we consider in this thesis is the expected total reward. Before we specify how it can be computed, we introduce the notion of rewards attached to a state or an action in a probabilistic model. A *reward structure* is a function of the form $r : S \to \mathbb{R}_{\geqslant 0}$ mapping states to non-negative reals. For MDPs and stochastic games, the reward structure can be extended to actions, i.e. $r : S \times A \to \mathbb{R}_{\geqslant 0}$. We use the term "reward" but, these often represent "costs" (e.g. elapsed time or energy consumption).

The total reward for reward structure r along an infinite path $\omega = s_0 s_1 s_2 \ldots$ is $r(\omega) = \sum_{j=0}^{\infty} r(s_j)$, which can be infinite. The expected total reward is defined as $E_{\mathsf{D},s}(r) = \int_{\omega\in IPath_s} r(\omega)\, dPr_{\mathsf{D},s}$. For technical reasons, we will always assume that the

maximum possible reward $E_{\mathsf{D},s}(r)$ is finite for any $s \in S$ (which can be checked with an analysis of the underlying graph).

An expected reward property is written as $\phi = \mathtt{R}^{r}_{\bowtie b}\,[\,\mathtt{C}\,]$ (where $\mathtt{C}$ stands for cumulative), meaning that, for DTMC D, the expected total reward for r satisfies $\bowtie b$ in the initial state and $\bowtie \in \{\leqslant, \geqslant, <, >\}$. To check if the property ϕ is satisfied we need to compute the value of the expected total reward. We denote the value by $\mathtt{R}^{r}_{=?}\,[\,\mathtt{C}\,] = E_{\mathsf{D},\overline{s}}(r)$. Subsequently, we need to check if $\mathtt{R}^{r}_{=?}\,[\,\mathtt{C}\,] \bowtie b$.

For an MDP M, we first define the reward on the path $\omega = s_0 a_0 s_1 a_1 \ldots$ by $r(\omega) = \sum_{j=0}^{\infty}[r(s_j) + r(s_j, a_j)]$. Then, for a strategy σ, we can define the expected total reward in state $s \in S$ as $E^{\sigma}_{\mathsf{M},s}(r) = \int_{\omega \in IPath_s} r(\omega)\, dPr^{\sigma}_{\mathsf{M},s}$. Similarly to probabilistic reachability, we will be interested in computing either the maximal or the minimal value of the total expected reward. We denote the maximal value by $\mathtt{R}^{r}_{\max=?}\,[\,\mathtt{C}\,] = \sup_{\sigma \in \Sigma_{\mathsf{M}}}\{E^{\sigma}_{\mathsf{M},\overline{s}}(r)\}$; the minimal value can be defined similarly.

For a stochastic game G and a pair of strategies (σ, π), we define the expected total reward in state $s \in S$ by $E^{\sigma,\pi}_{\mathsf{G},s}(r) = \int_{\omega \in IPath_s} r(\omega)\, dPr^{\sigma,\pi}_{\mathsf{G},s}$. Subsequently, we define the maximal value of expected total reward by $\mathtt{R}^{r}_{\max=?}\,[\,\mathtt{C}\,] = \sup_{\sigma \in \Sigma^{\Diamond}_{\mathsf{G}}} \inf_{\pi \in \Sigma^{\Box}_{\mathsf{G}}}\{E^{\sigma,\pi}_{\mathsf{G},\overline{s}}(r)\}$. The definition of the minimal value follows similarly.

Probabilistic reachability can be easily reduced to expected total rewards. We can encode it by replacing any outgoing transitions from states in the target set T with a single transition to a sink state labelled with a reward of 1.

An example of a property that can be expressed using expected total reward is "the expected time for a web server to successfully deliver a response is within 200 ms" or "the expected number of moves for a robot to reach the target state is below 5".

3.3 Model Checking and Controller Synthesis

Model checking and controller synthesis for MDPs and stochastic games are closely related problems. We will assume the property of the form $\phi = \mathtt{P}_{\bowtie p}\,[\,\mathtt{F}\ T\,]$ (respectively, for expected total reward $\phi = \mathtt{R}^{r}_{\bowtie b}\,[\,\mathtt{C}\,]$).

The *model checking* problem asks whether *for all* strategies in the model the property ϕ is satisfied. The *controller synthesis* problem asks whether there *exists* a strategy in the model that satisfies ϕ.

The solution to the model checking problem will yield a strategy that serves as a solution to the controller synthesis problem. For example, in order to synthesise the controller for property $\phi_1 = \mathtt{P}_{\geqslant b}\,[\,\mathtt{F}\ T\,]$, it is enough to compute the value of $\mathtt{P}_{min=?}\,[\,\mathtt{F}\ T\,]$ and subsequently check if value satisfies $\geqslant b$. The strategy that is obtained when computing $\mathtt{P}_{min=?}\,[\,\mathtt{F}\ T\,]$ is typically included in the output of the model checking algorithm and can

be used as a controller. The case for expected total reward and properties other than $\geqslant$ follows trivially.

Example 4 Consider the MDP from Figure 3.2 and the property $\phi = \mathtt{P}_{\geqslant 0.5}\,[\,\mathtt{F}\ \mathit{succ}\,]$. There are two memoryless deterministic strategies that obtain non-zero probability of reaching the state labelled with *succ*. The first one is $\sigma_1(\overline{s}) = \mathit{east}_1$, $\sigma_1(s_1) = \mathit{south}_1$, $\sigma_1(s_2) = \mathit{west}_2$, $\sigma_1(s_3) = \mathit{done}_1$, $\sigma_1(s_4) = \mathit{done}_2$, and achieves 0.5. The second strategy is $\sigma_2(\overline{s}) = \mathit{south}_2$, $\sigma_2(s_1) = \mathit{south}_1$, $\sigma_2(s_2) = \mathit{west}_2$, $\sigma_2(s_3) = \mathit{done}_1$, $\sigma_2(s_4) = \mathit{done}_2$ and achieves 0.45. Model checking property ϕ against MDP M would return a negative answer as only strategy σ_1 satisfies the property ϕ. The answer to the controller synthesis problem would be positive with the strategy σ_1 being a solution.

Example 5 We use the stochastic game from Figure 3.3 and consider the property $\phi = \mathtt{P}_{\geqslant 0.4}\,[\,\mathtt{F}\ \mathit{succ}\,]$. Consider the strategy $\sigma_1(\overline{s}) = \mathit{east}_1$, $\sigma_1(s_2) = \mathit{west}_1$, $\sigma_1(s_4) = \mathit{done}_1$. This strategy reaches the target state with probability 1 if the environment picks pass_1 in s_1, but if it picks block_1 we only achieve 0.25, which does not satisfy property ϕ. A strategy that satisfies ϕ against all environment player strategies is $\sigma_2(\overline{s}) = \mathit{south}_1$, $\sigma_2(s_2) = \mathit{west}_1$, $\sigma_2(s_4) = \mathit{done}_1$; this strategy achieves probability 0.45. The strategy σ_2 is a solution to the controller synthesis problem; similarly to the MDPs example, the answer to the model checking problem would be negative due to strategy σ_1.

3.4 Model Checking of Properties for Probabilistic Models

This section presents algorithms for computing both the probabilistic reachability and the expected total reward properties for probabilistic models from Section 3.1.

3.4.1 Model Checking of Properties for DTMCs

Probabilistic Reachability

Model checking probabilistic reachability properties for DTMCs can be reduced to solving a set of linear equations [44]. The solution of this set of linear equations will then define the reachability probability value for every state in the DTMC. Linear equations can be solved in polynomial time using Gaussian elimination or using iterative methods such as Gauss-Seidel or Jacobi.

Before we present the algorithm, we fix a DTMC $\mathsf{D} = \langle S, \overline{s}, \Delta, \mathcal{L}\rangle$ and the set of target states $T \subseteq S$, and use x_s to denote the reachability probability in state $s \in S$. The

probabilistic reachability algorithm can be divided into two steps. In the first step, we compute three disjoint sets S^{yes}, S^{no} and $S^{?}$, where $S = S^{yes} \cup S^{no} \cup S^{?}$. The set S^{yes} contains all the states for which the probability value is equal to one; this includes all the target states. The set S^{no} contains states that never reach the target set, and $S^{?}$ all other states. Both S^{yes} and S^{no} sets can be computed using simple graph traversal algorithms that can be found in [59]. For the states in S, we obtain the following set of linear equations:

$$x_s = \begin{cases} 1 & \text{if } s \in S^{yes} \\ 0 & \text{if } s \in S^{no} \\ \sum_{s' \in S} \Delta(s, s') \cdot x_{s'} & \text{if } s \in S^{?} \end{cases}$$

whose solution yields the reachability probability value in every state of the DTMC.

Example 6 We consider the DTMC presented in Figure 3.1, with the property being $\mathtt{P}_{=?}\,[\,\mathtt{F}\ \mathit{succ}\,]$. The sets $S^{yes} = \{s_4\}$, $S^{no} = \{s_3\}$ and the probabilistic reachability value for the states in the set $S^{?}$ can be obtained as a solution of the set of linear equations:

$$\begin{aligned} x_{\overline{s}} &= 0.2 \cdot x_{s_1} + 0.8 \cdot x_{s_2} \\ x_{s_1} &= 0.5 \cdot x_{s_2} + 0.5 \cdot x_{s_3} \\ x_{s_2} &= 0.5 \cdot x_{s_3} + 0.5 \cdot x_{s_4} \\ x_{s_3} &= 0 \\ x_{s_4} &= 1 \end{aligned}$$

Solving the above yields $x_{\overline{s}} = 0.45, x_{s_1} = 0.25$, $x_{s_2} = 0.5$, and $\mathtt{P}_{=?}\,[\,\mathtt{F}\ \mathit{succ}\,] = 0.45$.

Expected Total Reward

As in the case of probabilistic reachability, the expected total reward can also be computed as a solution of a set of linear equations. We fix a DTMC $\mathsf{D} = \langle S, \overline{s}, \Delta, \mathcal{L} \rangle$ and use x_s to denote the expected total reward in state $s \in S$. We divide the state space into two disjoint sets, S^{no} and $S^{?}$, such that $S = S^{no} \cup S^{?}$. The set S^{no} will now contain all the states with expected reward equal to zero. Those states can be identified by looking for BSCCs containing only states with zero reward assigned to them. After obtaining the set S^{no}, we can construct the set of linear equations, whose solution yields the expected reward value for $S^{?}$ states:

$$x_s = \begin{cases} 0 & \text{if } s \in S^{no} \\ r(s) + \sum_{s' \in S} \Delta(s, s') \cdot x_{s'} & \text{if } s \in S^{?}. \end{cases}$$

Example 7 Consider the DTMC from Figure 3.1, with the reward structure r_1 assigning $r_1(\overline{s}) = r_1(s_1) = r_1(s_2) = 1$ and the property $\mathtt{R}^{r_1}_{=?}[\,\mathtt{C}\,]$. The set $S^{no} = \{s_3, s_4\}$, and to compute the value for the set $S^?$ we have the following set of linear equations:

$$\begin{aligned} x_{\overline{s}} &= 1 + 0.2 \cdot x_{s_1} + 0.8 \cdot x_{s_2} \\ x_{s_1} &= 1 + 0.5 \cdot x_{s_2} + 0.5 \cdot x_{s_3} \\ x_{s_2} &= 1 + 0.5 \cdot x_{s_3} + 0.5 \cdot x_{s_4} \\ x_{s_3} &= 0 \\ x_{s_4} &= 0. \end{aligned}$$

The solution to the above set of linear equations is $x_{\overline{s}} = 2.1, x_{s_1} = 1.5$, $x_{s_2} = 1$, and $\mathtt{R}^{r_1}_{=?}[\,\mathtt{C}\,] = 2.1$.

3.4.2 Model Checking of Properties for MDPs

As we have seen in Section 3.2, the value of a probabilistic reachability or expected reward property depends on the chosen strategy. In this thesis, we are interested in properties where the probabilistic reachability or expected total reward satisfies a given bound. For those properties, it is enough to consider strategies that minimise or maximise the property value.

In Section 3.1.2, several classes of strategies have been introduced. A natural question therefore is which type of strategies are sufficient to compute the minimising/maximising strategy? For MDPs, a class of *memoryless deterministic* strategies is sufficient for both minimising or maximising the value of probabilistic reachability and expected total reward [107]. In [45, 47, 107], a linear programming (LP) encoding of the problem is presented. Linear programs can be solved in polynomial time [78], yielding polynomial time complexity for the problem of computing the optimal strategy in an MDP.

We describe two common solution methods for MDPs. The first one is the previously mentioned linear programming (LP), while the second is value iteration. Value iteration, starting from an under-approximation of the optimal values, iteratively computes a sequence of values getting closer to the actual value of the optimal strategy. The method may require an exponential number of iterations to converge, but in many cases we are able to terminate the computation after a small number of steps.

As a technicality, in the case of computing a maximising strategy, a precomputation step is applied before using any solution method. In the precomputation step, we collapse all MECs of an MDP. Collapsing a MEC merges all its states into a single state while preserving all outgoing and incoming transitions. The method has been described in

detail in [36]. MEC collapsing is not necessary for model checking purposes, but it is relevant for the controller synthesis problem.

In the following sections we present methods for computing the maximal value of the property; techniques for computing the minimal value are similar. We fix an MDP $\mathsf{M} = \langle S, \overline{s}, A, \delta, \mathcal{L} \rangle$ and target set $T \subseteq S$, and denote the value of the optimal strategy in state s by x_s.

Maximal Probabilistic Reachability

Before we define the encoding, we divide the state space into three disjoint sets; S^{yes}, S^{no}, and $S^{?}$, where $S = S^{yes} \cup S^{no} \cup S^{?}$. The set S^{yes} contains all the states for which there exists a strategy that reaches the target set with probability one; this set also includes all the target states. Please note that computing the set S^{yes} is not necessary for correctness, but can considerably improve the performance of the algorithm. The set S^{no} represents all the states for which, for all strategies, we never reach the target state. Similarly to DTMCs, states in S^{yes} and S^{no} sets can be identified using simple graph traversal algorithms presented in [59].

Linear Programming

The values for the remaining states in $S^{?}$ will be given by the solution of the following linear program:

Minimise: $\sum_{s \in S} x_s$ subject to:

$$
\begin{aligned}
x_s &\geqslant \sum_{s' \in S} \delta(s,a)(s') \cdot x_{s'} && \text{for all } s \in S^{?},\ a \in A(s) \\
x_s &= 1 && \text{for all } s \in S^{yes} \\
x_s &= 0 && \text{for all } s \in S^{no}.
\end{aligned}
$$

Example 8 Consider the MDP M from Figure 3.2 and property $\mathtt{P}_{max=?}[\,\mathtt{F}\ succ\,]$. The MDP M contains one non-trivial MEC: $(\{s_1, s_2\}, \{south_1, north_1\})$, which in the precomputation step is collapsed into a state $s_{1,2}$ such that: $\delta(s_{1,2}, west_1) = \delta(s_1, west_1)$, $\delta(s_{1,2}, west_2) = \delta(s_2, west_2)$, $\delta(\overline{s}, east_1)(s_1) = 0$, $\delta(\overline{s}, east_1)(s_{1,2}) = 1$, $\delta(\overline{s}, south_2)(s_2) = 0$, and the last element being $\delta(\overline{s}, south_2)(s_{1,2}) = 0.9$. The MDP M with the collapsed MEC can be seen in Figure 3.4. For all examples in this section we will assume that all MECs have been collapsed.

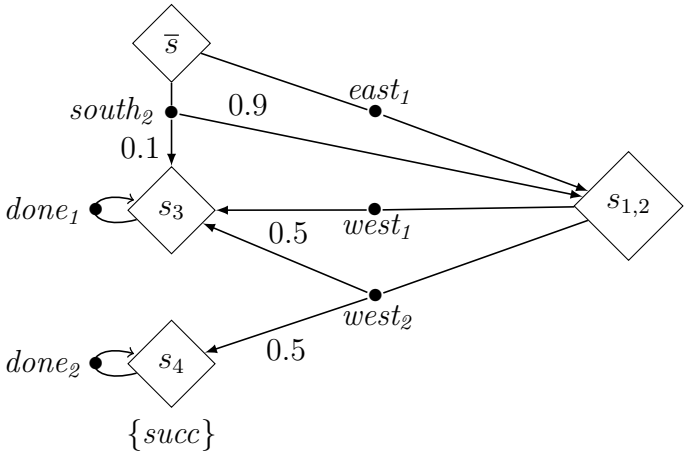


Figure 3.4: The MDP M from Figure 3.2 with MECs collapsed.

We begin by computing the sets S^{yes} and S^{no}. The set $S^{yes} = \{s_4\}$ and $S^{no} = \{s_3\}$, and for the states in $S^?$ we build the linear program:

Minimise: $x_{\overline{s}} + x_{s_{1,2}} + x_{s_3} + x_{s_4}$ subject to:

$$\begin{aligned} x_{\overline{s}} &\geqslant x_{s_{1,2}} \\ x_{\overline{s}} &\geqslant 0.9 \cdot x_{s_{1,2}} + 0.1 \cdot x_{s_3} \\ x_{s_{1,2}} &\geqslant x_{s_3} \\ x_{s_{1,2}} &\geqslant 0.5 \cdot x_{s_3} + 0.5 \cdot x_{s_4} \\ x_{s_3} &= 0 \\ x_{s_4} &= 1. \end{aligned}$$

The solution of the linear program yields values $x_{\overline{s}} = 0.5$ and $x_{s_{1,2}} = 0.5$, and hence $\mathtt{P}_{max=?}[\,\mathtt{F}\ succ\,] = 0.5$.

Value Iteration

We denote by x_s^k the probabilistic reachability value obtained in the k-th iteration of value iteration. We start with an under-approximation of the optimal value and set x_s^0 to 1 if $s \in S^{yes}$ and $x_s^0 = 0$, otherwise. The value in the k-th iteration, for every $s \in S$, is defined

by:

$$x_s^k := \begin{cases} 1 & s \in S^{yes} \\ 0 & s \in S^{no} \\ \max\limits_{a \in A(s)} \sum\limits_{s' \in S} \delta(s,a)(s') \cdot x_{s'}^{k-1} & s \in S^?. \end{cases}$$

When $k \to \infty$, values of x_s^k converge towards the optimal value x_s. The number of required steps until convergence might be exponential, but in practice we terminate the computation when a specified convergence criterion has been met. We use two convergence criteria: the first checks if the *absolute* difference between the successive values of x_s^{k-1} and x_s^k is smaller than specified threshold ε:

$$\max_{s \in S} |x_s^k - x_s^{k-1}| < \varepsilon$$

and the second uses the *relative* difference:

$$\max_{s \in S} |(x_s^k - x_s^{k-1})/x_s^k| < \varepsilon.$$

Example 9 We consider the same model and property as in the Example 8. Below, we present the value of x_s for each state for every iteration of value iteration.

$$\begin{array}{lllllll} k=0: & x_{\overline{s}}^0 = 0, & x_{s_1}^0 = 0, & x_{s_2}^0 = 0, & x_{s_3}^0 = 0, & x_{s_4}^0 = 1 \\ k=1: & x_{\overline{s}}^1 = 0, & x_{s_1}^1 = 0, & x_{s_2}^1 = 0.5, & x_{s_3}^1 = 0, & x_{s_4}^1 = 1 \\ k=2: & x_{\overline{s}}^2 = 0.45, & x_{s_1}^2 = 0.5, & x_{s_2}^2 = 0.5, & x_{s_3}^2 = 0, & x_{s_4}^2 = 1 \\ k=3: & x_{\overline{s}}^3 = 0.5, & x_{s_1}^3 = 0.5, & x_{s_2}^3 = 0.5, & x_{s_3}^3 = 0, & x_{s_4}^3 = 1 \\ k=4: & x_{\overline{s}}^4 = 0.5, & x_{s_1}^4 = 0.5, & x_{s_2}^4 = 0.5, & x_{s_3}^4 = 0, & x_{s_4}^4 = 1. \end{array}$$

Values between the iteration for $k = 3$ and $k = 4$ do not change, causing the convergence criterion (both absolute and relative) to be satisfied, subsequently terminating the computation.

Expected Total Reward

As in the previous section, assume a fixed MDP $\mathsf{M} = \langle S, \overline{s}, A, \delta, \mathcal{L} \rangle$ and let x_s denote the value of the maximising strategy in state s. For the computation of the expected total reward, we divide the state space into two disjoint subsets; S^{no} and $S^?$, where S^{no} contains states for which we never reach a state with non-zero reward. Similarly to probabilistic

reachability, these states can be computed using graph algorithms [59].

Linear Programming

To obtain the expected total reward values for the states in $S^?$, we build the linear program:

Minimise: $\sum_{s \in S} x_s$ subject to:

$$\begin{aligned} x_s &\geqslant r(s) + \sum_{s' \in S} \delta(s,a)(s') \cdot x_{s'} + r(s,a) && \text{for all } s \in S^?,\ a \in A(s) \\ x_s &= 0 && \text{for all } s \in S^{no}. \end{aligned}$$

Example 10 Consider the MDP in Figure 3.4. We use the reward structure r_2 that assigns rewards to actions as follows: $r_2(\overline{s}, east_1) = r_2(\overline{s}, south_2) = r_2(s_{1,2}, west_1) = r_2(s_{1,2}, west_2) = 1$. The property is $\mathtt{R}^{r_2}_{max=?}[\mathtt{C}]$.

The set $S^{no} = \{s_3, s_4\}$, and for the $S^?$ states we construct the linear program:

Minimise: $x_{\overline{s}} + x_{s_{1,2}} + x_{s_3} + x_{s_4}$ subject to:

$$\begin{aligned} x_{\overline{s}} &\geqslant x_{s_{1,2}} + 1 \\ x_{\overline{s}} &\geqslant 0.9 \cdot x_{s_{1,2}} + 0.1 \cdot x_{s_3} + 1 \\ x_{s_{1,2}} &\geqslant x_{s_3} + 1 \\ x_{s_{1,2}} &\geqslant 0.5 \cdot x_{s_3} + 0.5 \cdot x_{s_4} + 1 \\ x_{s_3} &= 0 \\ x_{s_4} &= 0. \end{aligned}$$

The solution of the linear program yields values $x_{\overline{s}} = 2, x_{s_{1,2}} = 1$, thus $\mathtt{R}^{r_2}_{max=?}[\mathtt{C}] = 2$.

Value Iteration

The value x_s is defined as the limit of:

$$x_s^k := \begin{cases} 0 & s \in S^{no} \\ r(s) + \max_{a \in A(s)} \sum_{s' \in S} \delta(s,a)(s') \cdot x_{s'}^{k-1} + r(s,a) & s \in S^? \end{cases}$$

as $k \to \infty$. The convergence criterion is the same as in the probabilistic reachability case.

Example 11 We illustrate the value iteration algorithm on the same model, reward

structure, and property as for Example 10. For every iteration of the algorithm and each $s \in S$, we have:

$$\begin{aligned}
k = 0: &\quad x^0_{\overline{s}} = 0, \quad x^0_{s_{1,2}} = 0, \quad x^0_{s_3} = 0, \quad x^0_{s_4} = 0 \\
k = 1: &\quad x^1_{\overline{s}} = 1, \quad x^1_{s_{1,2}} = 1, \quad x^1_{s_3} = 0, \quad x^1_{s_4} = 0 \\
k = 2: &\quad x^2_{\overline{s}} = 2, \quad x^2_{s_{1,2}} = 1, \quad x^2_{s_3} = 0, \quad x^2_{s_4} = 0 \\
k = 3: &\quad x^3_{\overline{s}} = 2, \quad x^3_{s_{1,2}} = 1, \quad x^3_{s_3} = 0, \quad x^3_{s_4} = 0
\end{aligned}$$

and the algorithm converges in only three iterations.

3.4.3 Model Checking of Properties for Stochastic Games

For stochastic games, the value of the property depends not only on the strategy of the controller player, but also on the strategy of the environment player. The type of stochastic games that we consider in this thesis are called *zero-sum* games. This is a type of a game where the controller player is trying to maximise (or minimise) the value of the property, while the environment player tries to minimise (or maximise). We will be interested in computing the optimal strategies, i.e. strategies that maximise (or minimise) the value of the property.

From [41, 55] we get that memoryless deterministic strategies are sufficient for obtaining the optimal strategies for both probabilistic reachability as well as expected total reward properties in zero-sum stochastic games. The best known complexity bound is NP $\cap$ co-NP [41], but, in practice, methods such as value iteration can be used efficiently.

We present techniques for computing the strategy that maximises the value of the property for the controller player; computing the minimal strategy follows similarly. We fix a stochastic game $\mathsf{G} = \langle S_\Diamond, S_\Box, S, \overline{s}, A, \delta, \mathcal{L} \rangle$ and target set $T \subseteq S$, and let x_s denote the value of the optimal strategy in state s.

Maximal Probabilistic Reachability

For stochastic games and value iteration, we do not apply precomputation to identify all states that have probability zero or one to reach the target set. We work with $S^{yes} = T$, where $S = S^{yes} \cup S^?$, and initialise $x^0_s = 0$ for all $s \in (S \setminus S^{yes})$. The value of the

probabilistic reachability property is defined as the limit of the series as $k \to \infty$:

$$x_s^k := \begin{cases} 1 & s \in S^{yes} \\ \max\limits_{a \in A(s)} \sum\limits_{s' \in S} \delta(s,a)(s') \cdot x_{s'}^{k-1} & s \in S^? \text{ and } s \in S_\Diamond \\ \min\limits_{a \in A(s)} \sum\limits_{s' \in S} \delta(s,a)(s') \cdot x_{s'}^{k-1} & s \in S^? \text{ and } s \in S_\Box. \end{cases}$$

We use the same convergence criterion as in the case of MDPs.

Example 12 Consider the model from Example 3, with the property $\mathtt{P}_{max=?}\,[\,\mathtt{F}\ succ\,]$. Below, we list the value of x_s for each state, for every iteration of value iteration.

$$\begin{array}{lllllll} k=0: & x_{\overline{s}}^0 = 0, & x_{s_1}^0 = 0, & x_{s_2}^0 = 0, & x_{s_3}^0 = 0, & x_{s_4}^0 = 1 \\ k=1: & x_{\overline{s}}^1 = 0, & x_{s_1}^1 = 0, & x_{s_2}^1 = 0.5, & x_{s_3}^1 = 0, & x_{s_4}^1 = 1 \\ k=2: & x_{\overline{s}}^2 = 0.45, & x_{s_1}^2 = 0.25, & x_{s_2}^2 = 0.5, & x_{s_3}^2 = 0, & x_{s_4}^2 = 1 \\ k=3: & x_{\overline{s}}^3 = 0.45, & x_{s_1}^3 = 0.25, & x_{s_2}^3 = 0.5, & x_{s_3}^3 = 0, & x_{s_4}^3 = 1. \end{array}$$

Values between the iteration for $k = 2$ and $k = 3$ do not change, causing the convergence criterion (both absolute and relative) to be satisfied, and subsequently terminating the computation.

Expected Total Reward

For computing the expected total reward, we do not need to consider target states; this simplifies the definition of the value x_s^k for each state:

$$x_s^k := \begin{cases} r(s) + \max\limits_{a \in A(s)} \sum\limits_{s' \in S} \delta(s,a)(s') \cdot x_{s'}^{k-1} + r(s,a) & s \in S_\Diamond \\ r(s) + \min\limits_{a \in A(s)} \sum\limits_{s' \in S} \delta(s,a)(s') \cdot x_{s'}^{k-1} + r(s,a) & s \in S_\Box. \end{cases}$$

We initialise $x_s^0 = 0$ for all $s \in S$ and use the same convergence criterion as before to compute the limit x_s.

Example 13 We use the model from Example 3 with the reward structure: $r_3(\overline{s}, east_1) = r_3(\overline{s}, south_1) = r_3(s_2, west_1) = 1$. The property is $\mathtt{R}^{r_3}_{max=?}\,[\,\mathtt{C}\,]$.

$$
\begin{array}{lllllll}
k=0: & x_{\bar{s}}^0=0, & x_{s_1}^0=0, & x_{s_2}^0=0, & x_{s_3}^0=0, & x_{s_4}^0=0 \\
k=1: & x_{\bar{s}}^1=1, & x_{s_1}^1=0, & x_{s_2}^1=1, & x_{s_3}^1=0, & x_{s_4}^1=0 \\
k=2: & x_{\bar{s}}^2=1.9, & x_{s_1}^2=0, & x_{s_2}^2=1, & x_{s_3}^2=0, & x_{s_4}^2=0 \\
k=3: & x_{\bar{s}}^3=1.9, & x_{s_1}^3=0, & x_{s_2}^3=1, & x_{s_3}^3=0, & x_{s_4}^3=0.
\end{array}
$$

Observe that value iteration converges after the third step.

3.5 The PRISM Modelling Language

In this section, we describe the native modelling language of the probabilistic model checker PRISM [88] used and extended in this thesis. We start by defining the syntax of the language, then describe the semantics, and conclude by presenting a set of examples for each of the model types we previously introduced.

A PRISM model description consists of a set of *constants* $c_1, \ldots, c_k$, a set of *modules*, a set of labels, and a set of players $p_1, \ldots, p_\ell$, where each player is represented by a set of actions it controls. Please note that each action in the model is controlled by exactly one player. In this thesis we focus on stochastic *two-player* games, and therefore we assume that $\ell = 2$. Every module M comprises a set of actions A^M, a set of guards G^M, a set of finite-domain variables $x_1, \ldots, x_n$ and a list of guarded commands of the form:

$$[a]\ g \to \lambda_1 : u_1 + \cdots + \lambda_m : u_m$$

where: $a \in A^M$ is an *action*; $g \in G^M$ is a *guard*, which is a Boolean expression over variables and constants, e.g. $(x_1 \leqslant c_2) \wedge (x_2{=}x_3 + 1)$; λ_i for $1 \leqslant i \leqslant m$ are probabilities, given as expressions over the set of constants, and summing to 1 for any valuation of the constants; and $u_1, \ldots, u_m$ are *variable updates*. A variable update is a conjunction of assignments of the form $(x_j{}'{=}expr_j)$, where x_j is a variable and $expr_j$ is an expression over variables and constants giving a new value for x_j based on the current state, e.g. $(x_1{}'{=}x_1{+}c_1){\wedge}(x_2{}'{=}x_3)$ is a variable update that increments x_1 by the value of the constant c_1 and assigns the value of x_3 to x_2. We re-use the definition of a guard and define a label l as a tuple (g_l, ap) where g_l is a guard and $ap \in AP$ is an atomic proposition.

Intuitively, a guarded command of the form given above indicates that, if the current values of the module's variables satisfy the guard g, then action a can be taken, after which update u_i is applied with probability λ_i.

Usually, a PRISM model comprises multiple modules, which are composed in parallel. In this section, for simplicity, we assume that a model always contains just a single module. The full semantics for the PRISM modelling language [104] describes how multiple modules can be syntactically composed into a single one.

Assuming a single module, the semantics of a PRISM model M is given by a stochastic game $\mathsf{G} = \langle S_\Diamond, S_\Box, S, \overline{s}, A, \delta, \mathcal{L}\rangle$ defined as follows. Let $x_1, \ldots, x_n$ be the variables of M and $V_1, \ldots, V_n$ their possible domains. For the controller player states we put $S_\Diamond = \{(v_1, \ldots, v_n) \in V_1 \times \cdots \times V_n \,|\, \exists\, g[v_1/x_1, \ldots, v_n/x_n] \wedge a \in p_1\}$. Conversely, for the environment player $S_\Box = \{(v_1, \ldots, v_n) \in V_1 \times \cdots \times V_n \,|\, \exists\, g[v_1/x_1, \ldots, v_n/x_n] \wedge a \in p_2\}$. We put $\overline{s} = (\overline{v}_1, \ldots, \overline{v}_n)$, where $\overline{v}_i \in V_i$ is the initial value of the variable x_i, and A to be equal to the set of all actions of all commands in M. The transition function δ is defined in terms of M's commands. To simplify the presentation we will make the assumption that, for each state $(v_1, \ldots, v_n)$ and action a, there is at most one a-labelled command of M whose guard g is satisfied in $(v_1, \ldots, v_n)$, i.e. where $g[v_1/x_1, \ldots, v_n/x_n]$ evaluates to true. Then, $\delta((v_1, \ldots, v_n), a)$ is defined for a state $(v_1, \ldots, v_n)$ if and only if such a command exists. When it does, let this (unique) command be:

$$[a]\ g \to \lambda_1 : u_1 + \cdots + \lambda_m : u_m$$

We define $\delta((v_1, \ldots, v_n), a)$ to be equal to the distribution which, for each state $(v'_1, \ldots, v'_n)$, assigns the probability equal to the sum of values λ_i, for $1 \leqslant i \leqslant m$, such that applying update u_i to variable values $(v_1, \ldots, v_n)$ gives $(v'_1, \ldots, v'_n)$. We set $\mathcal{L}((v_1, \ldots, v_n)) = \{ap\}$ if there is a label $l = (g_l, ap)$ such that $g_l[v_1/x_1, \ldots, v_n/x_n]$ evaluates to true. The semantics for DTMCs and MDPs follow similarly. Below, we give a simple example of a PRISM model description for each of the models considered in the thesis.

3.5.1 Discrete-Time Markov Chains

Example 14 We consider the PRISM model in Figure 3.5 of the DTMC from Figure 3.1. The PRISM model starts with a keyword that specifies the type of the model, in this case `dtmc`. The model consists of a single module *main* that contains one variable s and five commands. The declaration of the variable specifies its name, range and the initial value. The variable s will vary between zero and four, with initial value being zero. The first command in the module *main* is enabled in a state where $s = 0$. From that state, we can either go to the state where $s = 1$ with probability 0.2 or to the state $s = 2$ with probability 0.8. Labels mark the states with atomic propositions. In the last line of the PRISM model, the label marks the state where $s = 4$ with atomic proposition *succ*.

```
dtmc

module main

s : [0..4] init 0;

[] s=0 → 0.2 : (s'=1) + 0.8 : (s'=2);
[] s=1 → 0.5 : (s'=2) + 0.5 : (s'=3);
[] s=2 → 0.5 : (s'=3) + 0.5 : (s'=4);
[] s=3 → (s'=3);
[] s=4 → (s'=4);

endmodule

label "succ" = (s=4);
```

Figure 3.5: PRISM model description of the DTMC from Figure 3.1.

```
mdp

module main

s : [0..4] init 0;

[east_1] s=0 → (s'=1);
[south_2] s=0 → 0.1 : (s'=3) + 0.9 : (s'=2);
[south_1] s=1 → (s'=2);
[west_1] s=1 → (s'=3);
[north_1] s=2 → (s'=1);
[west_2] s=2 → 0.5 : (s'=3) + 0.5 : (s'=4);
[done_1] s=3 → (s'=3);
[done_2] s=4 → (s'=4);

endmodule

label "succ" = (s=4);
```

Figure 3.6: PRISM model description of the MDP from Figure 3.2.

3.5.2 Markov Decision Processes

Example 15 We analyse the PRISM model in Figure 3.6 of the MDP from Figure 3.2. In comparison to the DTMC example, we now have multiple commands defined for one state. In the state where $s = 0$ we have two commands labelled with $east_1$ and $south_2$ actions. The explanation of the rest of the model is the same as for the DTMC in Example 14.

3.5.3 Stochastic Games

Example 16 In our last example, we consider the PRISM model in Figure 3.7 of the stochastic game from Figure 3.3. At the beginning of a PRISM model describing a stochastic game, we need to define a mapping between actions (or whole modules) and players. In our case, we define two players; *controller* and *environment*, where the first player controls actions such as $east_1$ or $south_1$, while the second player controls $block_1$ or $pass_1$.

```
smg

player controller

[east_1], [south_1], [west_1], [done_1]

endplayer

player environment

[block_1], [pass_1], [block_2], [pass_2]

endplayer

module main

s : [0..4] init 0;

[east_1] s=0 → (s'=1);
[south_1] s=0 → 0.1 : (s'=3) + 0.9 : (s'=2);
[pass_1] s=1 → (s'=4);
[block_1] s=1 → 0.5 : (s'=2) + 0.5 : (s'=3);
[west_1] s=2 → 0.5 : (s'=3) + 0.5 : (s'=4);
[pass_2] s=3 → (s'=4);
[block_2] s=3 → (s'=3);
[done_1] s=4 → (s'=4);

endmodule

label "succ" = (s=4);
```

Figure 3.7: PRISM model description of the stochastic game from Figure 3.3.

The syntax of the PRISM model requires that each action is assigned to a player and it is never the case that two players own the same action. The rest of the model follows the description of DTMC and MDP models.

3.6 Difference Bound Matrices

One of the key factors behind the successful implementation of the methods presented in Chapter 5 is an efficient data structure called Difference Bound Matrices (DBMs) for representing conjunctions of linear constraints. This section is based on [116], which employed DBMs for real-time verification.

A DBM is a data structure that symbolically stores a conjunction of linear constraints of the form $x_i - x_j \prec k$, where $1 \leqslant i, j \leqslant n$ and $n \in \mathbb{N}$ is the number of variables present in the constraints, the operator $\prec \in \{\leqslant, \geqslant, <, >\}$ and $k \in \mathbb{R}_{\geqslant 0}$. There exist efficient algorithms [116] for performing various operations on DBMs, including operations that are relevant to this thesis: conjunction of two DBMs and computing a complement of a DBM.

Formally, a DBM [116] is square matrix of size $(n+1) \times (n+1)$. Each column and row of the matrix is assigned a variable. Row zero and column zero are assigned variables

with value zero and are used to represent constraints of the form $x_i \prec k$. Every element of the matrix is a tuple $(k, \lhd)$, where $\lhd \in \{<, \leqslant\}$. There are four cases that need to be considered when representing a linear constraint $x_i - x_j \prec k$ using a DBM $\mathcal{D}$:

- constraint $x_i - x_j \lhd k$, where $\lhd \in \{<, \leqslant\}$, for which $\mathcal{D}_{i,j} = (k, \lhd)$
- constraint $x_i - x_j \rhd k$, where $\rhd \in \{>, \geqslant\}$, for which $\mathcal{D}_{j,i} = (-k, <)$ if $\rhd \in \{>\}$, otherwise $\mathcal{D}_{j,i} = (-k, \leqslant)$
- constraint $x_i \lhd k$, for which $\mathcal{D}_{i,0} = (k, \lhd)$
- constraint $x_i \rhd k$, for which $\mathcal{D}_{0,i} = (-k, <)$ if $\rhd \in \{>\}$, otherwise $\mathcal{D}_{0,i} = (-k, \leqslant)$

All other non-diagonal elements of the DBM $\mathcal{D}$ are assigned the $(\infty, <)$ tuple. For the elements on the diagonal we use $(0, <)$. An example of a DBM for conjunction of constraints $(x_1 \leqslant 0 \wedge x_2 > 1)$ can be seen as $\mathcal{D}^1$ in Figure 3.8.

$\mathcal{D}^1$

	$\mathbf{0}$	x_1	x_2
$\mathbf{0}$	$(0, <)$	$(\infty, <)$	$(-1, <)$
x_1	$(0, \leqslant)$	$(0, <)$	$(\infty, <)$
x_2	$(\infty, <)$	$(\infty, <)$	$(0, <)$

$\mathcal{D}^2$

	$\mathbf{0}$	x_1	x_2
$\mathbf{0}$	$(0, <)$	$(\infty, <)$	$(-1, <)$
x_1	$(0, \leqslant)$	$(0, <)$	$(-1, <)$
x_2	$(\infty, <)$	$(\infty, <)$	$(0, <)$

Figure 3.8: DBM representation of a constraint $(x_1 \leqslant 0 \wedge x_2 > 1)$.

The DBM $\mathcal{D}^2$ from the Figure 3.8 not only represents $(x_1 \leqslant 0 \wedge x_2 > 1)$, but also adds a constraint $(x_1 - x_2 < -1)$. Given that many DBMs can represent the same conjunction of constraints, we define a partial order between DBMs and a normal form. For equal size DBMs $\mathcal{D}^1$ and $\mathcal{D}^2$, we define:

$$\mathcal{D}^1 \leqslant \mathcal{D}^2 \text{ iff } \forall\ 0 \leqslant i, j \leqslant n \ .\ \mathcal{D}^1_{i,j} \leqslant \mathcal{D}^2_{i,j}.$$

We say that DBM $\mathcal{D}$ is in normal form iff for every DBM $\mathcal{D}'$ that represents the same conjunction of constraints it holds that $\mathcal{D} \leqslant \mathcal{D}'$. The algorithm for computing the normal form of a DBM can be found in [116]. We now present how conjunction and complement of linear constraints can be done using DBM operations.

The conjunction of two DBMs $\mathcal{D}^1$ and $\mathcal{D}^2$ of the same size can be performed by creating a new DBM $\mathcal{D}^3$ of equal size, where each element:

$$\mathcal{D}^3_{i,j} = \min(\mathcal{D}^1_{i,j}, \mathcal{D}^2_{i,j}), \text{ where } 0 \leqslant i, j \leqslant n.$$

The resulting DBM $\mathcal{D}^3$ may need to be normalised. An example of a conjunction of two DBMs can be seen in Figure 3.9. The DBM $\mathcal{D}^1$ on the left-hand side of Figure 3.9 represents a conjunction of constraints $(x_1 \leqslant 0 \wedge x_2 > 1)$, DBM $\mathcal{D}^2$ represents the linear constraint $(x_1 \leqslant -2)$, and DBM $\mathcal{D}^3 = \mathcal{D}^1 \cap \mathcal{D}^2$ is now a conjunction of $(x_1 \leqslant -2) \wedge (x_2 > 1)$. For reasons of clarity $\mathcal{D}^3$ has not been normalised.

$\mathcal{D}^1$

	$\mathbf{0}$	x_1	x_2
$\mathbf{0}$	$(0, <)$	$(\infty, <)$	$(-1, <)$
x_1	$(0, \leqslant)$	$(0, <)$	$(\infty, <)$
x_2	$(\infty, <)$	$(\infty, <)$	$(0, <)$

$\mathcal{D}^2$

	$\mathbf{0}$	x_1	x_2
$\mathbf{0}$	$(0, <)$	$(\infty, <)$	$(\infty, <)$
x_1	$(-2, \leqslant)$	$(0, <)$	$(\infty, <)$
x_2	$(\infty, <)$	$(\infty, <)$	$(0, <)$

$\mathcal{D}^3$

	$\mathbf{0}$	x_1	x_2
$\mathbf{0}$	$(0, <)$	$(\infty, <)$	$(-1, <)$
x_1	$(-2, \leqslant)$	$(0, <)$	$(\infty, <)$
x_2	$(\infty, <)$	$(\infty, <)$	$(0, <)$

Figure 3.9: Conjunction of DBMs $\mathcal{D}^1$ and $\mathcal{D}^2$ (top), and the result DBM $\mathcal{D}^3$ (bottom).

To compute the complement $\overline{\mathcal{D}}$ of a DBM $\mathcal{D}$, we assume that $\mathcal{D}$ is represented as a conjunction of DBMs $\mathcal{D}^1, ..., \mathcal{D}^t$ where $1 \leqslant t \leqslant (n \times (n+1))$. Each element of the sequence $\mathcal{D}^1, ..., \mathcal{D}^t$ represents a single constraint that is expressed by DBM $\mathcal{D}$. An example can be found in Figure 3.10, where DBM $\mathcal{D}$ represents constraints $(x_1 \leqslant 0 \wedge x_2 > 1)$ and can be expressed as a conjunction of two DBMs $\mathcal{D}^1$ and $\mathcal{D}^2$. The DBM $\mathcal{D}^1$ represents the single constraint $(x_1 \leqslant 0)$, and DBM $\mathcal{D}^2$ represents $(x_2 > 1)$.

Complement $\overline{\mathcal{D}}$ can now be expressed as a disjunction of DBMs $\overline{\mathcal{D}^1}, ..., \overline{\mathcal{D}^t}$; this disjunction can be simply implemented as a list of DBMs. We assume that $(k_1, \lhd_1), ..., (k_t, \lhd_t)$ is the set of non-trivial bounds of $\mathcal{D}^1, ..., \mathcal{D}^t$, where a non-trivial bound is a non-diagonal element of a DBM that is different from $(\infty, <)$. For every $1 \leqslant l \leqslant t$, we have an element in the DBM $\mathcal{D}^l$ such that $\mathcal{D}^l_{i,j} = (k_l, \lhd_l)$, where $0 \leqslant i, j \leqslant n$. For every l, we create the complement $(-k_l, \lhd'_l)$ of $(k_l, \lhd_l)$, where $\lhd'_l$ is $<$ if $\lhd_l$ is $\leqslant$, and $\leqslant$ otherwise. We set $\overline{\mathcal{D}^l}_{j,i} = (k'_l, \lhd'_l)$ and $\overline{\mathcal{D}^l}_{i,j} = (0, <)$ when $i = j$ and $(\infty, <)$ for every other cell. Now the list of DBMs $\overline{\mathcal{D}^1}, ..., \overline{\mathcal{D}^t}$ represents a complement of DBM $\mathcal{D}$. The complement of DBMs $\mathcal{D}^1$ and $\mathcal{D}^2$ can be found in the bottom row of Figure 3.10.

$\mathcal{D}$

	$\mathbf{0}$	x_1	x_2
$\mathbf{0}$	$(0, <)$	$(\infty, <)$	$(-1, <)$
x_1	$(0, \leqslant)$	$(0, <)$	$(\infty, <)$
x_2	$(\infty, <)$	$(\infty, <)$	$(0, <)$

$\mathcal{D}^1$

	$\mathbf{0}$	x_1	x_2
$\mathbf{0}$	$(0, <)$	$(\infty, <)$	$(\infty, <)$
x_1	$(0, \leqslant)$	$(0, <)$	$(\infty, <)$
x_2	$(\infty, <)$	$(\infty, <)$	$(0, <)$

$\mathcal{D}^2$

	$\mathbf{0}$	x_1	x_2
$\mathbf{0}$	$(0, <)$	$(\infty, <)$	$(-1, <)$
x_1	$(\infty, <)$	$(0, <)$	$(\infty, <)$
x_2	$(\infty, <)$	$(\infty, <)$	$(0, <)$

$\overline{\mathcal{D}^1}$

	$\mathbf{0}$	x_1	x_2
$\mathbf{0}$	$(0, <)$	$(0, <)$	$(\infty, <)$
x_1	$(\infty, <)$	$(0, <)$	$(\infty, <)$
x_2	$(\infty, <)$	$(\infty, <)$	$(0, <)$

$\overline{\mathcal{D}^2}$

	$\mathbf{0}$	x_1	x_2
$\mathbf{0}$	$(0, <)$	$(\infty, <)$	$(\infty, <)$
x_1	$(\infty, <)$	$(0, <)$	$(\infty, <)$
x_2	$(1, \leqslant)$	$(\infty, <)$	$(0, <)$

Figure 3.10: DBM $\mathcal{D}$ and its complement $\overline{\mathcal{D}^1}$ and $\overline{\mathcal{D}^2}$.

3.7 Mixed-Integer Linear Programming

Mixed-Integer Linear Programming (MILP) is an optimisation problem, where we look for an assignment to a set of variables that satisfies a set of linear constraints and optimises the value of an objective function. Unlike linear programming (LP), in the case of MILP some of the variables can be required to be integers.

The MILP problem is known to be NP-hard [79], but there exist solvers [132, 131] that can solve even large instances of the problem efficiently. We refer the reader to [103] for an extensive review of combinatorial optimisation, including MILP. Below, we present an example of an MILP encoding as well as a run of the CPLEX solver [131].

Example 17 We use a simple Knapsack-based [79] combinatorial problem for which we provide an MILP encoding. Given a knapsack that can carry a fixed weight, we try to fit a number of items, each item having a weight and a value. The value of items in the knapsack should be maximised. In our case, we try to fit 4 items with the weights of $0.5, 1.0, 1.5$, and 2.0 and the corresponding values $1, 2, 3$, and 4. The knapsack can carry items whose combined weight is at most 3. The MILP encoding can be found below.

Maximise: $1.0 \cdot item_1 + 2.0 \cdot item_2 + 3.0 \cdot item_3 + 4.0 \cdot item_4$ subject to:

$$0.5 \cdot item_1 + 1.0 \cdot item_2 + 1.5 \cdot item_3 + 2.0 \cdot item_4 \leqslant 3.$$

The integer variables $item_i \in \{0, 1\}$, for $i = 1 \ldots 4$, indicate if the given item is in the knapsack.

The abbreviated output of the CPLEX solver can be found in Figure 3.11. We can see that the solver generates several solutions while trying to obtain the optimal one. We start with the solution that does not put any elements in the knapsack; this is indicated by column *Best Integer* and value 0. Often, obtaining the optimal solution may take

```
        Nodes                                         Cuts/
   Node  Left     Objective  IInf  Best Integer    Best Bound    ItCnt     Gap

*     0+    0                           0.0000       10.0000        0     ---
*     0+    0                           6.0000       10.0000        0   66.67%
      0     0        cutoff             6.0000        6.0000        0    0.00%

Solution pool: 2 solutions saved.

MIP - Integer optimal solution:  Objective =  6.0000000000e+00
```

Figure 3.11: CPLEX output for the MILP encoding from Example 17.

prohibitively long time. In those cases, we can stop the solver early and use an intermediate solution, which is guaranteed to satisfy the constraints but might be sub-optimal. Our example is a very simple problem, so we are able to find a solution within milliseconds. There are two equivalent optimal solutions. The first one picks the first three items, summing values to 6, while the second picks the second and last item, similarly summing to 6.

3.8 Real-Time Dynamic Programming

Finding an optimal strategy for an MDP is a widely studied problem in the field of planning [99], reinforcement learning [115], and optimal control [12]. Coming from these fields are two algorithms that are especially relevant in the context of this thesis, the Real-Time Dynamic Programming (RTDP) [9] and the Bounded Real-Time Dynamic Programming (BRTDP) [101].

We fix an MDP M, a reward structure r, a set of target states T and want to compute the minimal expected total reward in M. In contrast to Section 3.2.2, in this section we follow the definition of the expected total reward that includes target states. The difference is that, in the latter definition, instead of considering all infinite paths we only consider paths that reach the target state. We use such a definition to present RTDP and BRTDP as it was originally introduced in the literature.

Both RTDP and BRTDP work on a sub-class of MDPs that satisfy three assumptions [12]:

1. for all target states $s \in T$, $r(s) = 0$ and $\forall_{a \in A(s)} r(s, a) = 0$
2. M has at least one *proper* strategy
3. for all improper strategies there exists a state with infinite expected reward.

A *proper* strategy is a strategy that guarantees reaching the goal state with probability one for every state of M.

An outline of the RTDP algorithm can be found in Algorithm 1. We use x_s to denote the value of the expected reward in a state. The algorithm runs a potentially large number of random paths through the MDP, where each path starts in some predefined initial state and finishes in the goal state. While following each of the paths, the algorithm updates the current value of the expected reward for each visited state on the path. The choices between actions are resolved using a greedy strategy by choosing an action that gives the minimum expected reward. The successor of the action is picked randomly according to the distribution associated with the action.

Algorithm 1 Real-Time Dynamic Programming [9, 14]

Inputs: MDP M, state $s \in S$
$x_s \leftarrow 0$, for all $s \in S$
repeat
    RTDPTrial(s)
until *true*

function RTDPTrial(s)
    while $s \notin T$ **do**
        $a \leftarrow \arg\min_{a \in A(s)} r(s) + \sum_{s' \in S} \delta(s,a)(s')x_{s'} + r(s,a)$
        $x_s \leftarrow r(s) + \sum_{s' \in S} \delta(s,a)(s')x_{s'} + r(s,a)$
        $s \leftarrow$ sampled from $\delta(s,a)$
    end while
end function

The authors in [9] do not provide a termination criterion, but rather run the algorithm for a fixed amount of time; similarly, Algorithm 1 does not include a termination criterion. In [14] one possible termination criterion is presented. Bonet et al. [14] propose a termination criterion that is similar to the one presented earlier for value iteration. The computation of the RTDP algorithm is terminated when the difference between the currently stored expected reward and the value given by the greedily chosen action is smaller or equal than some predefined ε. This is checked for the initial state and all states reachable from it. The key reason behind the performance gains offered by RTDP comes from the fact that, while generating random paths through the MDP, we may not need to visit all the states before the computation converges. In contrast, in each iteration of value iteration we have to visit every state of the model.

The BRTDP algorithm [101] expands on the ideas presented in the RTDP algorithm. The key differences include the way states are visited and updated, as well as the termination criterion. In the case of BRTDP, the algorithm keeps both an upper and lower bound on the value of the expected reward. The algorithm is terminated when the difference between those bounds for the initial state is smaller than some ε. Apart from the termination criterion, the value of the difference between the upper and lower bound in a state can be used in the path generation mechanism. The algorithm favours paths that explore states with a larger difference, subsequently providing a significant speed-up. The BRTDP improves performance over the RTDP algorithm. A detailed analysis can be found in [101]. Below, we present an example for the RTDP algorithm. For a detailed presentation of the BRTDP algorithm and an example run we refer the reader to [101].

Example 18 The MDP from Figure 3.2 does not contain a proper strategy, and therefore cannot be used as an example for the RTDP algorithm. We slightly modified the MDP

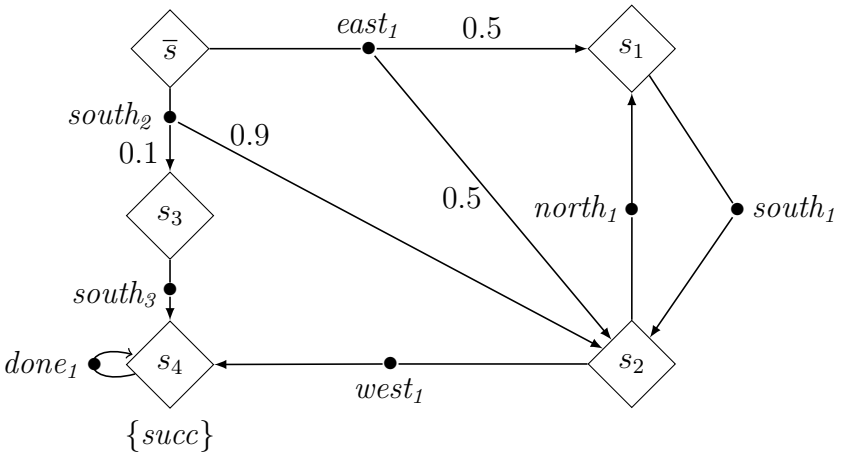


Figure 3.12: Example of an MDP M satisfying the assumptions (1), (2), and (3).

from Figure 3.2 and in our example we use MDP M from Figure 3.12 with reward structure $r(\overline{s}, east_1) = r(\overline{s}, south_2) = r(s_1, south_1) = r(s_2, north_1) = r(s_2, west_1) = r(s_3, south_3) = 1$ and s_4 as the only target state. The modified MDP satisfies the assumptions needed for Algorithm 1.

We consider three runs of the algorithm that generate three paths through the MDP ω_1, ω_2 and ω_3; all runs start in the initial state $\overline{s}$. The first run is $\omega_1 = \overline{s} east_1 s_1 south_1 s_2 north_1 s_1 south_1 s_2 west_1 s_4$. This run becomes trapped between s_1 and s_2 states, until the action $west_1$ is able to provide a lower value of the expected reward. The run is terminated after visiting the target state s_4. The runs $\omega_2 = \omega_3 = \overline{s} south_2 s_3 south_3 s_4$ never visit the s_1 state, as it no longer provides the minimal reward value. After the three runs the value in $x_{\overline{s}}$ contains the minimal expected reward of 2.0, but the algorithm itself does not terminate. The exact values of x_s variables after each run can be found below.

$$
\begin{array}{lllllll}
\omega_1: & x_{\overline{s}} = 0, & x_{s_1} = 2, & x_{s_2} = 1, & x_{s_3} = 0, & x_{s_4} = 0 \\
\omega_2: & x_{\overline{s}} = 1.9, & x_{s_1} = 2, & x_{s_2} = 1, & x_{s_3} = 1, & x_{s_4} = 0 \\
\omega_3: & x_{\overline{s}} = 2, & x_{s_1} = 2, & x_{s_2} = 1, & x_{s_3} = 1, & x_{s_4} = 0.
\end{array}
$$

Chapter 4

Verification and Controller Synthesis at Runtime

The purpose of this chapter is to set the context for this thesis by introducing a runtime framework. Every method presented in this thesis addresses a specific aspect of the framework, and in Chapter 8 we describe a case study implementing it.

In this thesis, we are interested in runtime control of systems in order to respond to dynamic changes in a system or its environment. The controller synthesis procedure is based on the analysis of a parameterised probabilistic model of the system. The model is updated over time using information from intermittent monitoring. Monitoring tracks changes happening to the system and its environment, which are then quantified and forwarded to the synthesis method. The generated controller is employed by the running system to decide which system actions should be executed to ensure that the property of interest is satisfied. The synthesis process is run continuously as new data is provided by the monitoring process, and new controllers are generated and used by the system.

In Chapter 8, we instantiate the framework in the context of an open-source stock monitoring application. *StockPriceViewer* is an application that provides stock price information for a portfolio of stocks. Stock price information can be retrieved using one of many available providers. Each provider offers a different quality of service. We develop a model of the application, where the control strategy is used to decide which stock information provider should be used at a given point in time. The controller should minimise the overall time needed for fetching the pricing information for a portfolio of stocks. Because each provider offers different quality of service, we use monitoring to gain information on the current performance of the providers. After parameterising the model with the data from the monitoring, we generate a controller which is then used by the *StockPriceViewer* at runtime. As time progresses, new controllers are generated and the process continues.

4.1 Components of the Framework

The framework consists of four components: a *computer system*, an *environment*, a *monitoring module*, and a *verification module*. In Figure 4.1 we show how the elements of the framework interact with each other and, in the following section, we describe each component in detail.

4.1.1 Computer System

We are interested in computer systems that contain both controllable and uncontrollable actions and exhibit probabilistic behaviour. Controllable actions represent *system actions*, such as choosing a path that a remotely controlled robot should follow. Uncontrollable actions represent choices that are outside system control. For example, while the robot travels on a path chosen by the system, it might be a subject of an enemy attack. Often such systems contain stochastic components, where probabilities are used to capture unreliability of the hardware components or uncertainty in the information received from the external sensors.

Probabilistic systems containing both controllable and uncontrollable behaviour can be modelled using *stochastic two-player games*, where one player represents the controllable actions of the system and the other represents the uncontrollable actions of the environment. We are interested in generating a controller that wins against all possible behaviours of the environment player and satisfies a property of interest. A range of properties of probabilistic systems can be described using probabilistic reachability and expected total reward, with examples of such properties being "the probability of a robot successfully fulfilling its mission is above 0.999" or "the overall expected time to fetch a portfolio of stocks is always below 500ms".

We are not always able to build the model of the system offline. The reason is that not all properties of the system can be specified in advance, or it is infeasible to consider all possible values of such properties. The mentioned properties include response times of Internet webservices that tend to change over time and need to be constantly updated or, in case of a robot, a battery level, where it is impractical to try to analyse all possible battery levels offline.

Therefore, we consider a family of models that are parameterised with values computed at runtime. We refer to system characteristics that are unknown until runtime as the *system parameters*.

For *StockPriceViewer*, we build a stochastic game model using the PRISM modelling language. The system actions represent the choice between different stock information

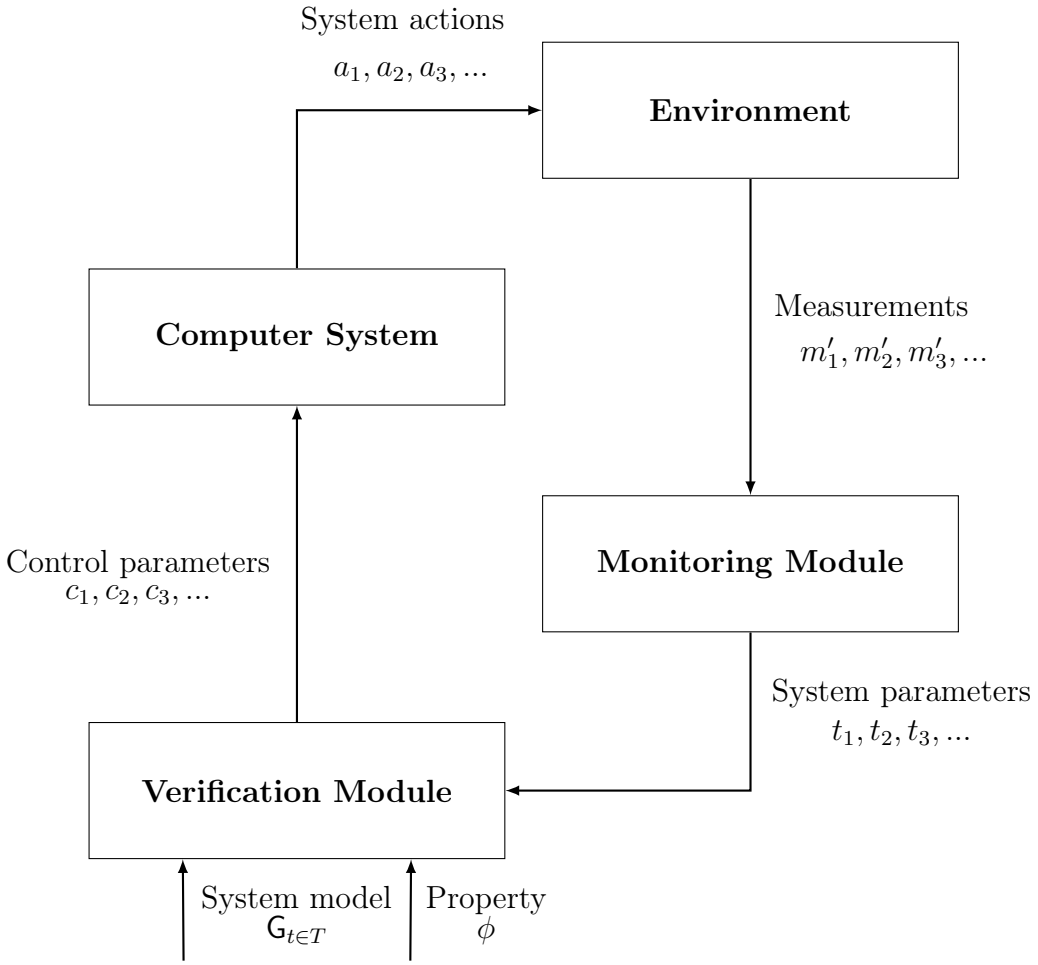


Figure 4.1: Verification and Controller Synthesis at Runtime.

providers, while the environment actions the adversarial behaviour of the environment. The system parameters represent the response time and probability of failure of each of the providers. The property of interest is to minimise the overall time that is needed for fetching stock pricing information for a portfolio of stocks. The controller in our model will decide which provider should be chosen and will be used by the application at runtime.

4.1.2 Environment

The *environment* component represents the outside world that the computer system interacts with. We assume that the environment offers a set of *system actions* that can be used to interact with it. Subsequently, we assume that the environment is not under our control but its behaviour can be fully observed and its parameters can be measured.

For *StockPriceViewer*, the environment represents multiple stock information providers. The computer system uses system actions to query the providers and retrieve information

about the stock price. Each provider is implemented as a webservice. While we cannot change how each of the providers work, we can measure the quality of service that each offers.

4.1.3 The Monitoring Module

To obtain the value of the *system parameters* at runtime, we assume the existence of a *monitoring module*. The monitoring module measures some of the parameters of the environment and, at regular time intervals, outputs the value of system parameters. We assume that the environment provides an interface that allows for measuring the current value of system parameters. Depending on the instantiation of the framework, the monitoring module may perform some computation on the data, for example to mitigate the influence of outliers. After the computation is done, the monitoring module will forward the computed values to the verification module.

In the case of *StockPriceViewer*, the monitoring module takes the form of a simple script. We query each of the stock information providers every second and obtain a sequence of measurements. Each measurement reports if the query was successful and includes a response time in milliseconds. To avoid being vulnerable to temporary spikes in response time, we average the results over a 5 minute period and forward the computed values to the verification module.

4.1.4 The Verification Module

The last component of the framework is the *verification module*, which alters the behaviour of the system based on the output of the model checker. The model checker analyses the system model, which is parameterised using data from the monitoring module, against a property ϕ, and outputs a number of *control parameters*. Each control parameter takes the form of a controller strategy, which defines which controllable actions the system should choose at any decision point.

We use the PRISM model checker as an implementation of the verification module. The input of PRISM is the stochastic game model of *StockPriceViewer*, with the response time and failure probability of each stock information provider being model parameters. These parameters are set according to the current values supplied by the monitoring module. We use a total expected reward property to specify a query that minimises the overall expected time needed to fetch the stock prices for a portfolio of stocks. We run PRISM for the mentioned model and property. The output of PRISM is a strategy which identifies the provider that should be chosen such that the property holds. The strategy is then stored in a textual form and used by the application.

4.2 Key Characteristics of the Verification Module

In this thesis we solely focus on the verification module and thus address three key requirements that must be met to make the work practical:

1. *Performance* - as the controlled system is running at the same time as controller synthesis, it is crucial for the synthesis method to provide results quickly so that they are still relevant for the running system. Before the controller can be synthesised, a probabilistic model of the system must be constructed from a high-level textual representation, e.g., the input language of a model checker. In many cases, the model construction process can account for a significant proportion of the overall synthesis time.

   In Chapter 5, we exploit the fact that the synthesis method is rerun continuously for the same model with possibly different parameters, and provide an *incremental model construction* method that significantly reduces the model construction time.

   In Chapter 8, we run our method on the probabilistic model of the *StockPriceViewer* application. The incremental model construction method is relevant here because, over time, we tend to vary the number of stocks in our portfolio. Ideally, we would want to avoid rebuilding the whole model when adding only a small number of stocks.

2. *Robustness* - due to transient failure of a system component, some of the controllable actions may become temporary unavailable. This could render the system inoperable in cases when the generated controller picks such an action.

   In Chapter 6, we propose a new *permissive controller synthesis* method that generates controllers that can overcome situations when some system actions become temporarily unavailable.

   In Chapter 8, we show how permissive controller synthesis can be used when some of the stock information providers become temporarily unavailable. It often happens that the currently used provider stops providing the content for a couple of seconds, in which case our method can be employed to pick a different provider that still satisfies the property.

3. *Scalability* - systems that we focus on in this thesis tend to grow over time and may be already large to begin with.

   In Chapter 7, we propose a new scalable *learning-based controller synthesis* method. The proposed method is based on machine learning techniques and allows us to

synthesise controllers for significantly larger models, outperforming current state-of-the-art methods.

In Chapter 8, we run our learning-based controller synthesis method on the model of the *StockPriceViewer*. By applying our technique, we were able to decrease the controller synthesis time by three orders of magnitude. This subsequently decreased the amount of time that the application has to wait until it could start fetching stock prices.

4.3 Summary

The aim of this chapter was to introduce a framework that is used throughout the thesis. We defined the four elements of the framework and described how they communicate with each other. We used the *StockPriceViewer* application as an example and explained how the framework works in such a setting. In Chapter 8, we provide more information on the framework implementation in the context of *StockPriceViewer*. To make the framework useful in practice, we identified three key requirements. In the coming chapters, we will present methods that address each of the presented requirements, thus fulfilling our aim of practicality of this work.

Chapter 5

Incremental Model Construction

5.1 Introduction

In this chapter, we present techniques to address the performance of the controller synthesis method working within the runtime framework described in Chapter 4. The controller synthesis method is run repeatedly and, in many cases, the analysed models only differ in the value of parameters. We assume that the models are described using PRISM's modelling language and that the changes in the models are caused by updating the values of model parameters based on the input from the monitoring module. We further assume that the model for one set of parameter values has been constructed, and we aim to build the model for a different value of some parameter.

To exploit the fact that we already have a model built for one set of parameters, we propose an *incremental model construction* technique. Our method avoids constructing the updated model from scratch by analysing the high-level PRISM model description and mapping the changes from the high-level model to the underlying probabilistic model. The mapping defines which states are subject to change and, by re-visiting only those, we are able to build the model for the new set of parameters while only visiting a small subset of states.

The key to performance improvements of our method are two-fold. The first is an effective way of mapping each command from the PRISM model to the set of states that the given command satisfies. We use a symbolic data structure called *difference bound matrices (DBM)* (see Section 3.6) that employs a matrix-based representation to succinctly describe a set of states that enable it. The second source of performance improvements is a method for storing and retrieving states based on their variable values. For this we use *red-black trees* [43], which outperform PRISM's native data structures.

The chapter begins with definitions, and specifically the notion of parameters and

parameter change in PRISM models. Next, we describe the non-incremental model construction as currently implemented in the PRISM model checker. After establishing the necessary background for our method, we present incremental model construction. The method is first presented without specifying the underlying data structures, which are introduced later. To conclude the chapter, we evaluate the method on several PRISM case studies, where we compare our approach with the non-incremental approach. Subsequently, we show that our method applies not only to stochastic games but also to other probabilistic models such as DTMCs and MDPs. To provide further statistics of performance improvements of our method, we include profiling of the model construction process, as well as experiments that consider several consecutive changes of the model parameters.

5.2 Parameters in PRISM Models

We start by formalising the notion of a parameter change. We assume that the stochastic game is given as a PRISM model description and the changes are made by altering *parameters* of the PRISM model. These are constants in the model description (see Section 3.5) whose values are not determined until runtime. We use *parameterised* PRISM model descriptions to capture the notion of change of a stochastic game G to G' by stipulating that the stochastic games G and G' are obtained from the same PRISM model by choosing different values of parameters.

There are two prominent kinds of changes that might be imposed. Firstly, *non-structural* changes preserve the initial state and the existence of non-zero probability transitions, i.e. they only change δ, while preserving the support of each $\delta(s, a)$ for any $s \in S$ and $a \in A$. Any other changes are *structural* and involve the modification of the stochastic game's underlying graph.

For non-structural changes, model construction is straightforward and has been considered in parametric model checking [32, 68]: we build a parametric probabilistic model, in which transition probabilities are stored as expressions over model parameters. To construct a probabilistic model for a particular set of parameter values, we evaluate the expression associated with each transition of the model using those values. For structural changes, we propose a new *incremental model construction* algorithm, which we describe in Section 5.4.

5.3 Non-Incremental Model Construction

Before we describe the incremental model construction algorithm, we outline the standard (non-incremental) model construction as implemented in the PRISM model checker. This will simplify the presentation of the incremental algorithm.

PRISM supports two distinct styles of implementation, *explicit-state* and *symbolic*. Explicit-state methods use data structures such as sparse matrices and arrays, and manipulate each state of a model separately. In comparison, symbolic methods use data structures such as binary decision diagrams and simultaneously manipulate *sets* of states. In this work, we focus on scenarios where small changes are made to individual states of a model, for which symbolic approaches are known to be less efficient. Thus, we work with explicit-state methods (although we later make performance comparison with both symbolic and explicit PRISM implementations).

The explicit-state model construction algorithm can be found in Algorithm 2. The input of the algorithm is the model M defined in the PRISM modelling language. The variable R stores the states for which we run our exploration; initially R contains the initial state. Between lines 4 and 18, we run a breadth-first search based exploration for the states found in R. For each state we explore, in lines 8 and 10 we identify its possible successors. This is done based on high-level description of the model. After adding the explored state to either $S_\Diamond$ or $S_\Box$ in lines 12 and 14, we move to analyse states that are reachable from the explored state. In line 16, we assign to $\hat{S}$ all successors of the currently explored state. In line 17, those successors are checked if they have been previously visited. If not, they are added to R and will be explored in the future iterations of the algorithm. The search is terminated when all reachable states have been visited.

One of the main optimisations of the model construction process is converting the created model G into a more optimised representation that keeps all the states sorted by the value. This is done outside of Algorithm 2 and in practice can significantly decrease the time needed for the subsequent analysis of the model.

Example 19 Figure 5.1 includes the PRISM model M of a simple stochastic game G where *good_robot* tries to move between locations, while *bad_robot* can impede this process. The *good_robot* states are marked as diamonds, whereas we use boxes to indicate *bad_robot* states. The model contains one parameter, constant *s_max*.

The PRISM model from Figure 5.1 produces a stochastic game with non-unique actions for values *s_max* higher than 3. This is possible as multiple states satisfy the command marked with the *south* action. Such scenarios are a common occurrence in PRISM case studies, as it is not always practical to enforce action uniqueness at the PRISM model level. Recall that, in this thesis, we assume that all actions are unique. Therefore, the

Algorithm 2 *ConstructModel*

Inputs: PRISM model description M
Outputs: Stochastic game G
$S_\Diamond := \emptyset$,$S_\Box := \emptyset$; $A = \emptyset$; $\delta := \emptyset$
Compute the initial state $\overline{s}$ in M
$R := \{\overline{s}\}$
while $R \neq \emptyset$ **do**
    $s := dequeue(R)$
    Compute $A(s)$ according to M and add it to A
    Compute $\mathcal{L}(s)$ according to M
    Set $\delta(s, a)$ for all $a \in A$ according to M
    if s belongs to $\Diamond$ player according to M **then**
        $S_\Diamond := S_\Diamond \cup \{s\}$
    else
        $S_\Box := S_\Box \cup \{s\}$
    end if
    $\hat{S} := \{s' \mid \exists a \in A(s) : \delta(s, a)(s') > 0\}$
    $R := R \cup (\hat{S} \setminus S)$
end while
$S = S_\Diamond \cup S_\Box$
$\mathsf{G} = \langle S_\Diamond, S_\Box, S, \overline{s}, A, \delta, \mathcal{L} \rangle$
return G

```
smg

const int s_max;

player good_robot

[east], [south], [done], [restart]

endplayer

player bad_robot

[pass], [block]

endplayer

module game

s : [0..s_max];

[east] s=0 → (s'=1);
[pass] s=1 → (s'=2);
[block] s=1 → 0.8 : (s'=2) + 0.2 : (s'=1);
[south] s > 1 & s < s_max → 1 : (s'=min(s + 1, s_max));
[done] s=s_max → 1 : true;
[restart] s=s_max → 1 : (s'=0);

endmodule
```

Figure 5.1: PRISM model M with s_max as the undefined constant.

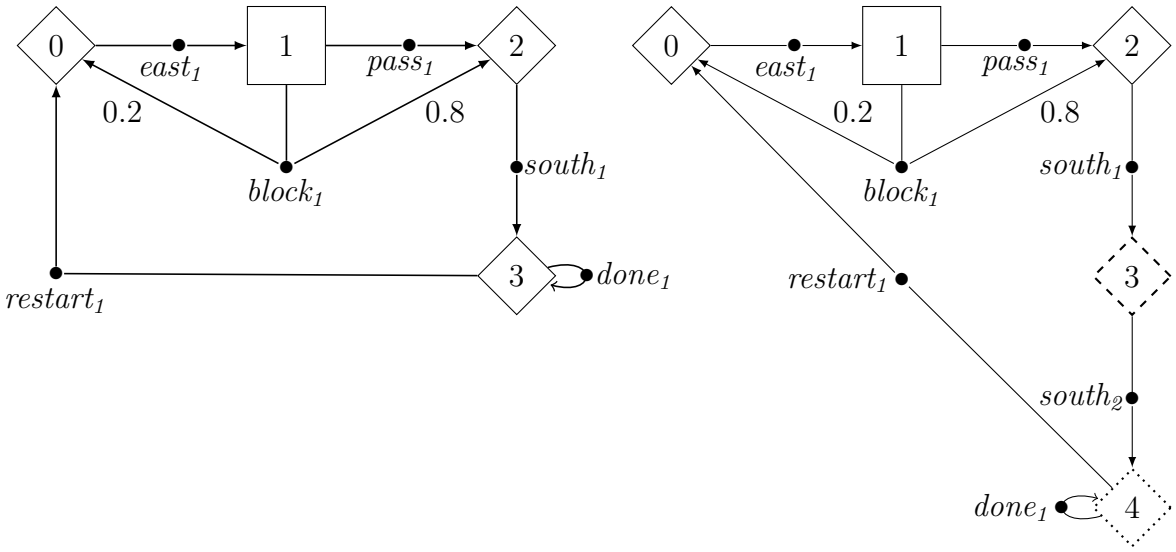


Figure 5.2: Stochastic games G^1(left) and G^2(right) for PRISM model from Figure 5.1.

model construction algorithm adds a unique suffix to each action so that the built model will only contain unique actions.

In Figure 5.2 we can see stochastic games G^1 (left) and G^2 (right) built for $s\_max = 3$ and $s\_max = 4$. Both models share the same states for $s = 0, 1, 2$ and 3, and intuitively the model construction for $s\_max = 4$ could be optimised by re-using the state space of G^1 and only re-visiting state 3.

In the last step before introducing the incremental model construction, we present results from profiling the non-incremental model construction algorithm. We used VisualVM [125] as the profiler running on a PC with a 1.7GHz i7 Core processor and 4GB RAM. We run PRISM-games [31], an extension of PRISM adapted to run stochastic games. As examples we use *mer* (for $N = 1..100$) and *firewire* (for $deadline = 1000..1050$). More information about the examples can be found in Appendix A.

The results are summarised in Figure 5.3, where we report the percentage of the total model construction time spent in each of the four categories of operations. The total model construction time is the sum of model construction time for each value of the parameter. The first category is the *evaluation* that refers to the process of evaluating the information about the given state; examples are lines 8, 10 and 11 in Algorithm 2. By *storage* we indicate operations pertaining to storing the states during the model construction process, present in lines 12, 14 and 17. Conversion of the model into a more optimised representation has been indicated as *transformation*. Finally, any other operation has been classified as *other*.

Most of the total model construction time is spent on evaluating states. This is a costly operation as it requires access to a high-level representation of the model. The

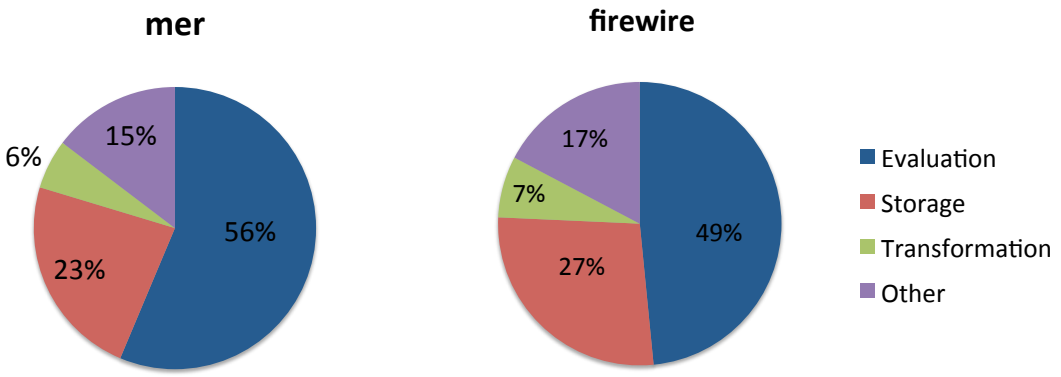


Figure 5.3: Profiling non-incremental model construction.

storage operations take between 23% to 27% of the model construction time, while the transformation and other operations add up to another $21\% - 24\%$. A successful implementation of incremental model construction will have to avoid as many evaluation and storage operations as possible.

5.4 Incremental Model Construction

For a PRISM model description, exploration of each state requires evaluating the guard for every command in the model to see if it is enabled in that state. However, when the number of states is large, checking every guard for every state becomes costly. Our approach to incremental model construction aims to reduce the work required by using a previously constructed model as a basis: re-exploration is only carried out from a subset of the states in the existing model, and only a subset of the guards of the PRISM model are evaluated. By re-using the previously constructed model, we will also perform fewer storage operations, further improving the performance.

More precisely, let us consider the scenario where we have a model description M, containing one or more parameters, and have already built a stochastic game $\mathsf{G}^1 = \langle S^1_\Diamond, S^1_\Box, S^1, \overline{s}^1, A^1, \delta^1, \mathcal{L}^1 \rangle$ by instantiating M with parameter values P_1. We now want to build a new stochastic game $\mathsf{G}^2 = \langle S^2_\Diamond, S^2_\Box, S^2, \overline{s}^2, A^2, \delta^2, \mathcal{L}^2 \rangle$ for parameter values P_2, using the existing model G^1 as a starting point.

Pseudocode for the incremental model construction algorithm *ConstructModelIncr* can be found in Algorithm 3. The inputs are G^1, M, P_1 and P_2, and the algorithm returns the new stochastic game G^2. We use $g[P_i]$ to denote the guard g instantiated with parameter values P_i, and $M[P_i]$ for the whole model description M instantiated with values P_i. The function $\mathit{Satisfy}(g)$ returns all (possibly unreachable) states that satisfy a guard g.

Algorithm 3 *ConstructModelIncr*

Inputs: Stochastic game G^1, PRISM model description M, parameters P_1, P_2
Outputs: Stochastic game G^2
$R^+ := \emptyset$
for all guards $g \in M$ containing parameters **do**
    $R^+ := R^+ \cup \big(\mathit{Satisfy}(g[P_1]) \setminus \mathit{Satisfy}(g[P_2])\big)$
    $R^+ := R^+ \cup \big(\mathit{Satisfy}(g[P_2]) \setminus \mathit{Satisfy}(g[P_1])\big)$
end for
$R := R^+ \cap S^1$
$\overline{s}_2 :=$ the initial state in $M[P_2]$
if $\overline{s}_2 \neq \overline{s}_1$ **then** $R := R \cup \{s_2\}$ **end if**
$S^2_\Diamond := S^1_\Diamond \backslash R$
$S^2_\Box := S^1_\Box \backslash R$
$S^2 := S^2_\Box \cup S^2_\Diamond$
$A^2 := \emptyset$
for all $s \in S^2$ **do**
    $\delta^2(s, \cdot) := \delta^1(s, \cdot)$
    $A^2 := A^2 \cup A^1(s)$
    $\mathcal{L}^2(s) := \mathcal{L}^1(s)$
end for
while $R \neq \emptyset$ **do**
    $s := \mathit{dequeue}(R)$
    Compute $A^2(s)$ according to $M[P_2]$ and add it to A^2
    Compute $\mathcal{L}^2(s)$ according to $M[P_2]$
    Set $\delta^2(s, a)$ for all $a \in A^2$ according to $M[P_2]$
    if s belongs to $\Diamond$ player according to $M[P_2]$ **then**
        $S^2_\Diamond := S^2_\Diamond \cup \{s\}$
    else
        $S^2_\Box := S^2_\Box \cup \{s\}$
    end if
    $S^2 := S^2 \cup \{s\}$
    $\hat{S} := \{s' \mid \exists a : \delta^2(s, a)(s') > 0\}$
    $R := R \cup (\hat{S} \setminus S^2)$
end while
$\mathsf{G}_2 := \langle S^2_\Diamond, S^2_\Box, S^2, \overline{s}^2, A^2, \delta^2, \mathcal{L}^2 \rangle$
$\mathsf{G}_2 := \mathit{RestrictToReachable}(\mathsf{G}^2, \overline{s}^2)$
return G^2

Lines 3–10 determine the set $R \subseteq S^1$ of states from G^1 that need to be re-explored to build G^2. This is done by analysing each guard g of a command in M that features a parameter and identifying two classes of states: those that satisfied g for P_1 but no longer do for P_2; and those that didn't satisfy g for P_1 but now do for P_2. We first identify the set R^+ of all possible such states, and then restrict this to those that appear in G^1 (see Section 5.5 for further details). If the initial state of G^2 differs from G^1, we also add that to R. The remainder of the algorithm constructs G^2, starting from G^1 as a basis (lines 11–19) and then re-explores states in R (lines 20–33). For states in S^1 but not in R, stochastic game G^2 is identical to G^1. States in R are recursively re-explored and added to G^2, except for states already contained in G^2 which are ignored. During this process, some transitions originally contained in G^1 may be removed, making some states of G^2 unreachable. To remove such states, we run the algorithm *RestrictToReachable*$(\mathsf{G}^2, \overline{s}_2)$, which returns the part of G^2 that is reachable from the initial state $\overline{s}_2$.

The most costly part of the algorithm is the *Satisfy* function since it has to traverse the state space to find states that satisfy a given guard. In the next section, we focus on improving the performance by representing the state space using a more effective data structure.

Algorithm 3 only considers cases when the change of the parameter is contained within the guard part of a command, since this seems to be the most common case in practice. The algorithm can be easily extended to cases when the change appears in the update part or to the probability values that cause a structural change in the model. To support such cases all states satisfying the guard associated with an affected element need to be added to the set R.

Example 20 We now illustrate the incremental model construction on the earlier example from Section 5.3 (see Figures 5.1 and 5.2). We keep s_max as a parameter and consider two parameter values, $P_1 = \{s\_max = 3\}$ and $P_2 = \{s\_max = 4\}$.

The stochastic game G^1 for parameter values P_1 is shown in Figure 5.2 (left). We construct G^2, for parameter values P_2, by following Algorithm 3. There are two distinct guards in the model, $g_1 = (s > 1 \wedge s < s\_max)$ and $g_2 = (s = s\_max)$, that contain the parameter s_max. The *Satisfy*$(g_1[P_1])$ function returns $\{2\}$, for *Satisfy*$(g_1[P_2])$ we have $\{2, 3\}$, *Satisfy*$(g_2[P_1]) = \{3\}$ and *Satisfy*$(g_2[P_2])$ returns $\emptyset$.

R^+ is the set of states satisfying $s \geqslant 3 \wedge s < 4$ and $R = \{3\}$. So, 3 is the only state from S^1 whose transitions change. From line 20, we start to re-explore states from R. Re-exploring state 3 yields the new state 4 (state 0 is also rediscovered but not re-explored since it is already in S^2). There are no unreachable states in the new model, and so *RestrictToReachable* leaves it unchanged. The resulting stochastic game G^2 is shown

in Figure 5.2 (right), where dashed lines denote the state whose outgoing transitions were modified, and dotted lines indicate newly added states.

Correctness

To conclude this section, we provide a formal proof of the correctness of Algorithm 3.

Theorem 1. *The stochastic game constructed by Algorithm 3 is the stochastic game corresponding to M with parameter values P_2.*

The key idea behind the proof is to first prove that the set R contains all the states from S^1 that may have different transitions in G^2 (compared to G^1), i.e., for every $s \in S^1$, whenever $\delta^1(s,a) \neq \delta^2(s,a)$ for some $a \in A^2$, then $s \in R$. Because δ_2 is re-computed for such states, we get that for any $s \in S^1 \cap S^2$ and $a \in A^2$ the distribution $\delta^2(s,a)$ is identified correctly. Subsequently, we have to prove that all new states in S^2, i.e. $s \in (S^2 \setminus S^1)$, are reachable on a path going through one of the states in R or on a path that starts in a new initial state and consists only of states that belong to $S^2 \setminus S^1$. The exploration, starting in line 20 (see Algorithm 3), ensures that such states will be added, along with the corresponding transitions. Finally, we need to prove that S^2 does not contain any unreachable states. The last claim follows trivially from the definition of the *RestrictToReachable* algorithm, and hence we only prove the first of the two above claims.

Lemma 1. *For every $s \in S^1$, whenever $\delta^1(s,a) \neq \delta^2(s,a)$ for some $a \in A^2$, then $s \in R$.*

Proof. We give a proof by contradiction for the case when we add a transition; removing a transition can be handled in a similar way. Let us assume there exists a state $s \in S^1$ satisfying $\delta^1(s,a) \neq \delta^2(s,a)$ for some $a \in A^2$, but not contained in R. When a new transition is added in a state $s \in S^1 \cap S^2$ there must exist a guard $g_i[P_2]$ satisfied in state s and a syntactically equivalent guard $g_i[P_1]$ that is not satisfied in state s. In such case in line 6, we have s being added to R^+, and hence it is subsequently added to R. □

Lemma 2. *All new states in S^2, i.e. $s \in (S^2 \setminus S^1)$, are reachable on a path going through one of the states in R, or on a path that starts in a new initial state and consists only of states that belong to $S^2 \setminus S^1$.*

Proof. Paths that start in a new initial state ($\overline{s}_2 \neq \overline{s}_1$) and consist only of states from $S^2 \setminus S^1$ are handled by the exploration that happens between lines 20-33 and by the fact that the new initial state is contained in R.

For new states reachable on paths going through R, we will prove the existence of such paths by contradiction. Let us assume that a new state $s \in (S^2 \setminus S^1)$ is reachable

on a path that does not include any state from R. As this path cannot contain states in $S^2 \setminus S^1$ only (as this case is covered by the first part of the proof), we move backwards on this path and find state $s \in S^1$ and its successor $s' \in (S^2 \setminus S^1)$ on a path. For such a state s there exists an action $a \in A^2$ such that $\delta^2(s,a)(s') > 0$. As $s' \notin S^1$ we have that $\delta^1(s,a)(s') = 0$, and from Lemma 1 we know that state s has to belong to R, which contradicts our assumption. □

5.5 Symbolic Representation

As mentioned in the previous section, the most costly part of the incremental model construction procedure described above is the computation of the set of states R whose transitions are to be re-generated. A simple implementation of the *Satisfy* function would iterate through the list of states in S^1, checking which ones satisfy each guard g for parameter values P_1 and P_2. However, by imposing some restrictions on the syntax of the PRISM modelling language, we can use a more efficient solution.

Typically, guards in a PRISM model are Boolean combinations of simple arithmetic constraints over variables and constants. To simplify presentation, we can assume that the guards are in disjunctive normal form (in fact, in practice, guards are often just a single conjunction of constraints). Let us further assume that we only have constraints of the form $x_i \sim c$ or $x_i - x_j \sim c$, where x_i and x_j are variables, $\sim \in \{\leqslant, \geqslant, <, >\}$ and $c \in \mathbb{Q}$. We will later describe how to generalise this further.

With this assumption in place, we can represent a guard from a PRISM model symbolically, using difference-bounded matrices (DBMs), see Section 3.6 for more information. We can now return to the implementation of Algorithm 3. Recall that we present the computation of the set R as being done in two steps: first computing the set R^+, and then computing $R := R^+ \cap S^1$. Assuming the restrictions on guards in a PRISM model outlined above, we can generate R^+ symbolically, using DBMs, and then intersect it with S^1.

For the first step, we construct a list of DBMs for each guard g, as required; this is done using standard operations such as disjunction, conjunction and negation on DBMs to build R^+. The second step can be done efficiently by storing S^1 as a set of red-black trees [43]. More precisely, we keep a red-black tree containing states of S^1, sorted by each variable of the model. Since a DBM representing a guard can give us the lower and upper bound on each variable, we use those values to obtain all states satisfying a given guard. Such range queries can be efficiently implemented using red-black trees.

As an optimisation, to avoid searching red-black trees containing all states of the model, we implement the following technique. If the guards are of the form $x \sim N \wedge g'$ where x

$$\text{Line 5:}\quad \begin{array}{c|cc} \textit{Satisfy}(g_1[P_1]) & \mathbf{0} & \mathbf{s} \\ \hline \mathbf{0} & (0,<) & (-2,\leqslant) \\ \mathbf{s} & (2,\leqslant) & (0,<) \end{array} \setminus \begin{array}{c|cc} \textit{Satisfy}(g_1[P_2]) & \mathbf{0} & \mathbf{s} \\ \hline \mathbf{0} & (0,<) & (-2,\leqslant) \\ \mathbf{s} & (3,\leqslant) & (0,<) \end{array} = \begin{array}{c} R^+ \\ \hline \emptyset \end{array}$$

$$\text{Line 6:}\quad \begin{array}{c|cc} \textit{Satisfy}(g_1[P_2]) & \mathbf{0} & \mathbf{s} \\ \hline \mathbf{0} & (0,<) & (-2,\leqslant) \\ \mathbf{s} & (3,\leqslant) & (0,<) \end{array} \setminus \begin{array}{c|cc} \textit{Satisfy}(g_1[P_1]) & \mathbf{0} & \mathbf{s} \\ \hline \mathbf{0} & (0,<) & (-2,\leqslant) \\ \mathbf{s} & (2,\leqslant) & (0,<) \end{array} = \begin{array}{c|cc} R^+ & \mathbf{0} & \mathbf{s} \\ \hline \mathbf{0} & (0,<) & (-3,\leqslant) \\ \mathbf{s} & (3,\leqslant) & (0,<) \end{array}$$

Figure 5.4: DBM representation of the guard g_1 from Example 21.

is a variable, $\sim \in \{\leqslant, \geqslant, <, >\}$ is a comparison operator, N is the parameter subject to change, and g' is an expression described by a DBM, we keep a red-black tree containing all states of the model satisfying g', ordered by the value of x. When performing model construction, we then identify trees linked to DBMs contained in R, and only search through these trees. Although in the worst case this can mean searching several large trees, in practice these trees tend to be small, and hence the method yields a significant speedup over searching one tree containing all states.

Example 21 We now go back to the PRISM model M and stochastic game G_1 from Examples 19 and 20. After introducing DBMs, the *Satisfy* function from Algorithm 3 does not return states but a DBM describing which states satisfy the given guard. In Figure 5.4, we can see a matrix representation of the DBM describing states that satisfy guard $g_1 = (s > 1 \wedge s < s\_max)$ for $P_1 = \{s\_max = 3\}$ and $P_2 = \{s\_max = 4\}$. The operations from lines 5 and 6 of Algorithm 3 now operate on DBMs and the set R^+ is also described as a DBM. The DBM storing R^+ can be seen on the right-hand side of Figure 5.4. DBMs provide upper and lower bound on the stored variable and for our example we obtain $s = 3$.

The conjunction operation from line 8 of Algorithm 3 is implemented as a range query on the red-black tree and in our case returns $R = \{3\}$. Now, from line 9, we follow the non-DBM version of the incremental model construction algorithm. For an example of a run of the non-DBM incremental model construction see Example 20.

Restrictions of DBMs

We previously mentioned that DBMs are an efficient way to store constraints of the form $(x_i - x_j) \leqslant c$. Without further insights, such restrictions on the structure of the constraints significantly reduce the number of models defined in the PRISM modelling language that we can handle. We propose a solution that handles additional types of constraints, including $(x_i + x_j) \leqslant c$, $(a \cdot x_i + b \cdot x_j) \leqslant c$, where $a, b \in \mathbb{Q}_{\geqslant 0}$, as well as functions, if the input of these functions are variables or defined constants.

To extend the class of models that we can handle we introduce a *renaming scheme*. Let us assume that one of the guards contains a constraint of the form $(x_i + x_j) \leqslant C$, where C is one of the unresolved parameters. We replace $x_i + x_j$ by an auxiliary variable $x_{i,j}$ and in every update of the model that changes the value of x_i or x_j we add $x_{i,j} := x_i + x_j$. We execute this step before obtaining DBMs from the guards. As such, the obtained DBM includes constraints of the variable $x_{i,j}$, which will be used during the search process. The other type of constraint can be supported using exactly the same scheme.

5.6 Experimental Results

We have implemented our incremental model construction algorithm in PRISM-games [31], building on its "explicit" model checking engine and using its built-in DBM library. Our experiments were performed on an Intel Core i7-2600 CPU 3.40GHz Quad-Core Processor with 8GB memory running Fedora 15 x86_64 Linux.

We present run times for both of PRISM's implementations of model construction, using symbolic and explicit-state data structures. Symbolic implementation is available in the MTBDD engine, explicit-state implementation in the explicit engine. The former is often more efficient, especially for large models, but the latter is also relevant since for some models it can outperform the MTBDD engine and it is the basis for our implementation. At the time of writing this thesis, the explicit engine was the only available engine for stochastic games.

We investigated the performance of Algorithm 3 on six PRISM models. Although this thesis focuses on stochastic games, the incremental model construction technique also applies to other types of probabilistic models supported by PRISM, namely Markov decision processes (MDPs) and discrete-time Markov chains (DTMCs). Of the six models we use, the first two are stochastic games: *android_3* and *mdsm*; *zeroconf*, *firewire*, and *mer* are MDPs and the last one is a DTMC (*crowds*). The *crowds* example is used to show how our method performs in a case when one parameter affects several variables. We include full details of these models, and of their parameters, online at [126], and

in Appendix A.

Table 5.1 and 5.2 show the running times for our incremental model construction approach, compared to the standard (non-incremental) methods implemented in PRISM. We present average times for constructing two models using successive values of a model parameter (see column 2). For the incremental case, this entails one normal model construction, followed by one incremental step. In our experiments, parameter values are incremented by one but the algorithm works in the same fashion when values are decremented or changed by a bigger value.

In Table 5.1, we see the comparison results for MDPs for both the explicit and MTBDD engine of PRISM. The first observation that we make is that in some cases the explicit engine can outperform the MTBDD engine. This can be seen for almost all parameter values in *zeroconf* case study and for all values in *firewire*. Previously [104], it has been shown that the MTBDD engine performs well for probabilistic models that contain a certain degree of regularity. We believe this is not the case for *zeroconf* and *firewire* models. Typically, regularity is present in models that originate from a model written in PRISM modelling language that contains multiple modules that are similar. This is not the case in the two examples that we just mentioned.

Performance results

We now move to compare results of the incremental method with non-incremental model construction. For *zeroconf* and *firewire* the incremental method shows very good improvements and the second step takes only a small fraction of time, bringing the average time close to the half of the explicit engine time.

This is possible because in many cases the explicit engine that is the basis of our implementation outperforms the MTBDD engine. Additionally, in the incremental step, the algorithm has to create a smaller set of states than a non-incremental model construction algorithm. For example, in the case of *zeroconf* and $K = 9 - 10$, the incremental model has to add $389,532$ states, whereas the non-incremental has to build a full state space of $3,001,911$ states. The difference is considerably larger in the case of *firewire* and $deadline = 6000 - 6001$, where the incremental algorithm adds 601 states compared to $3,374,468$ added by the non-incremental algorithm.

For *zeroconf* and $K = 9 - 10$, we observe counter-intuitive behaviour where the incremental method takes significantly less than half of the time of the non-incremental method. This is surprising because, for $K = 9$, we build the model using non-incremental method and only for $K = 10$ apply the incremental method. Subsequently, the best achievable result for the incremental method should be around half of the non-incremental method, but

Model [parameters]	Parameter values	States	Model construction time (s)		
			PRISM (MTBDD)	PRISM (explicit)	Incr. (Alg. 3)
zeroconf ($N = 6, K$)	$3 - 4$	179,774 – 307,768	6.1	2.7	2.3
	$4 - 5$	307,768 – 496,291	9.9	4.3	3.7
	$5 - 6$	496,291 – 798,471	16.2	7.2	6.1
	$6 - 7$	798,471 – 1,248,568	25.4	11.8	9.5
	$7 - 8$	1,248,568 – 1,870,338	38.9	18.8	14.6
	$8 - 9$	1,870,338 – 2,612,379	55.5	55.6	20.9
	$9 - 10$	2,612,379 – 3,001,911	69.8	95.6	27.6
firewire (*deadline*)	$3000 - 3001$	1,570,867 – 1,571,468	32.6	5.3	3.9
	$3500 - 3501$	1,871,367 – 1,871,968	43.1	6.4	4.4
	$4000 - 4001$	2,171,867 – 2,172,468	55.5	7.8	5.2
	$4500 - 4501$	2,472,367 – 2,472,968	71.7	8.7	6.0
	$5000 - 5001$	2,772,867 – 2,773,468	86.2	9.9	6.8
	$5500 - 5501$	3,073,367 – 3,073,968	109.5	10.7	7.3
	$6000 - 6001$	3,373,867 – 3,374,468	122.1	15.4	8.4
mer (N)	$25 - 26$	149,239 – 155,146	0.8	2.0	1.4
	$50 - 51$	296,914 – 302,821	1.6	3.8	2.5
	$75 - 76$	444,589 – 450,496	2.3	5.7	3.6
	$100 - 101$	592,264 – 598,171	3.0	7.9	4.7
	$125 - 126$	739,939 – 745,846	3.8	9.8	6.2
	$150 - 151$	887,614 – 893,521	4.6	11.6	7.0
crowds (*TotalRuns*)	$50 - 51$	357,477 – 378,847	1.2	1.9	1.6
	$60 - 61$	610,672 – 641,112	1.8	3.7	2.6
	$70 - 71$	961,767 – 1,002,877	2.8	5.5	4.2
	$80 - 81$	1,426,762 – 1,480,142	3.7	8.5	6.3
	$90 - 91$	2,021,657 – 2,088,907	4.8	13.4	9.0
	$100 - 101$	2,762,452 – 2,845,172	6.2	24.5	15.7

Table 5.1: Experimental results for incremental model construction (Algorithm 3) for MDPs and DTMCs.

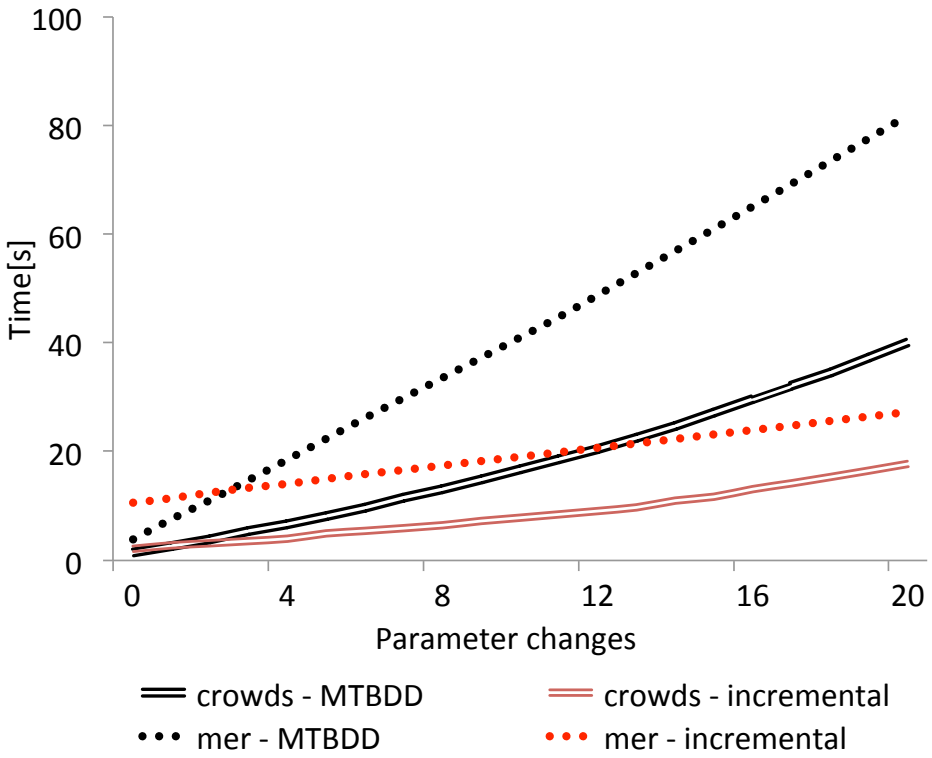


Figure 5.5: Multiple runs of incremental model construction.

not significantly below that number. We believe this behaviour is caused by poor garbage collection of the non-incremental model construction algorithm. The non-incremental algorithm keeps the first model in the memory, while the model for the second experiment is being built. The incremental method keeps only one model in memory, and therefore can avoid such situations.

Outperforming MTBDD engine

For both *crowds* and *mer* case study, the explicit engine is several times slower than the MTBDD one. Therefore, the incremental algorithm will not be able to outperform the MTBDD engine when considering just a single parameter change. Instead, in those cases we perform a set of experiments for a large range of the parameter values, and show the results as a graph in Figure 5.5.

The results demonstrate that the incremental method quickly improves over the non-incremental model construction that uses the MTBDD engine. In the case of the *crowds* case study we improved after only 2 iterations, and for *mer* we improved after 3. In subsequent iterations, the time required by the incremental method grows more slowly than the time required by the non-incremental algorithm. This is possible because, for both *crowds* and *mer*, we observed only small increases of the state space size when

Model [parameters]	Parameter values	States	Model construction time (s)	
			PRISM (explicit)	Incr. (Alg. 3)
android_3 $[r, s]$	$3, 500 - 501$	240,097−240,577	2.9	2.4
	$3, 1000 - 1001$	480,097−480,577	8.2	4.9
	$3, 1500 - 1501$	720,097−720,577	13.4	7.8
	$3, 2000 - 2001$	960,097−960,577	21.7	11.3
	$3, 2500 - 2501$	1,200,097−1,200,577	28.0	13.6
mdsm $[N, D]$	$3, 50 - 51$	574,968−586,488	7.6	6.1
	$3, 75 - 76$	862,968−874,488	10.7	7.1
	$3, 100 - 101$	1,150,968−1,162,488	16.1	9.4
	$3, 125 - 126$	1,438,968−1,450,488	20.9	12.8
	$3, 150 - 151$	1,726,968−1,738,488	24.8	18.8
	$4, 10 - 11$	613,728−675,936	10.8	7.7
	$4, 20 - 21$	1,235,808−1,298,016	21.5	13.0
	$4, 30 - 31$	1,857,888−1,920,096	34.6	22.9

Table 5.2: Experimental results for incremental model construction (Algorithm 3) for stochastic games.

incrementing the parameter.

Table 5.2 presents our results for stochastic games. The incremental model construction outperforms the non-incremental method in every case. Overall, when compared to results obtained for MDPs and DTMC, our algorithm exhibits similar performance for stochastic games. This is expected because the model construction process does not significantly differ for the three types of probabilistic models. Similarly to MDPs and DTMC examples, we attribute the performance of the algorithm to a small number of states that needs to be re-visited and subsequently added.

Profiling incremental method

We conclude the experimental section with a profiling experiment that we previously considered in Section 5.3, but now performed for the incremental model construction. We refer the reader to Section 5.3 for more information about the setup of the experiment. We present the results in Figure 5.6.

In contrast to the non-incremental algorithm, the incremental method spends significantly less time on the *evaluation* and *storage* operations. At the end of Section 5.3, we mentioned that reducing the time requirement of these operations will be a key factor for obtaining performance improvements. The incremental algorithm spends the majority of its time on *other* and *transformation* operations, which always need to be executed,

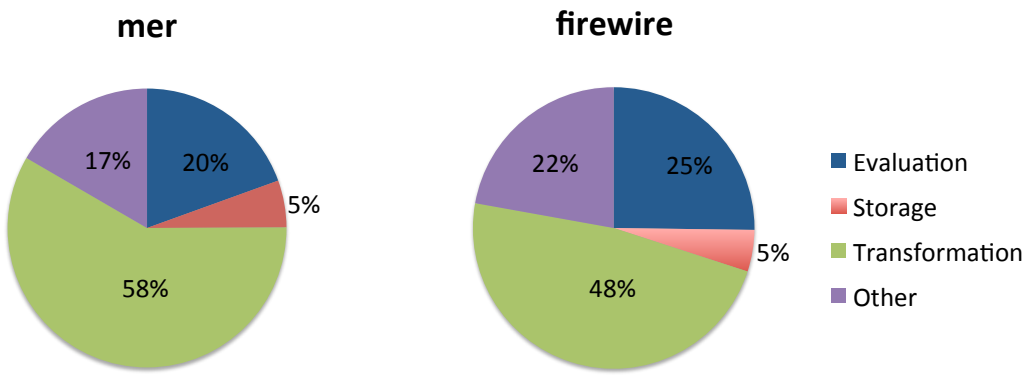


Figure 5.6: Profiling incremental model construction.

regardless of the model construction method used. This justifies our intuition that our implementation of the incremental model construction is efficient and reduces the number of expensive state re-evaluations to the minimum.

The experimental results in this section are very encouraging and show that the incremental model construction performs well in every case we considered. The method almost always outperforms the explicit engine and, for cases where the non-incremental model construction gives better results, we improve when considering several parameter changes.

5.7 Summary

In this chapter, we presented a novel incremental model construction technique. We first showed how changes to PRISM model parameters can cause either structural or non-structural changes to the probabilistic model. We then presented the non-incremental model construction algorithm, which forms the basis of our incremental method.

By performing profiling of the non-incremental method, we learned that, in order to achieve performance gains in our new algorithm, we had to reduce the number of state evaluations and use a more efficient data structure for storing states.

A DBM proved to be an effective way for mapping between PRISM commands and states satisfying guards of those commands. This data structure allowed us to significantly reduce the number of state evaluations, which was the first source of the performance improvements. To efficiently store states, we used red-black trees, which was the second source of speed-up provided by our method.

Lastly, we discussed the implementation of our techniques and showed experimental results performed using several PRISM case studies. Our method turned out to be very efficient when compared to non-incremental model construction and, even when considering

a single change of parameters, we were able to demonstrate a significant speed-up.

Chapter 6

Permissive Controller Synthesis

6.1 Introduction

In this chapter, we tackle the problem of synthesising *robust* and *flexible* controllers, which are resilient to unexpected changes in the system at runtime. For example, one or more of the actions that the controller can choose at runtime might unexpectedly become unavailable, or additional constraints may be imposed on the system that make some actions preferable to others. We cast this problem as a means to improve the robustness of the controllers used in the framework from Chapter 4.

To formalise the notion of robust and flexible controllers we, develop novel, *permissive* controller synthesis techniques for systems modelled as stochastic two-player games. Rather than generating *strategies*, which specify a single action to take at each time-step, we synthesise *multi-strategies*, which specify multiple possible actions. As in classical controller synthesis, generation of a multi-strategy is driven by a formally specified quantitative property: we focus on probabilistic reachability and expected total reward properties. The property must be guaranteed to hold, whichever of the specified actions are taken and regardless of the behaviour of the environment.

Simultaneously, we aim to synthesise multi-strategies that are as *permissive* as possible, which we quantify by assigning *penalties* to actions. These are incurred when a multi-strategy blocks (does not make available) a given action. Actions can be assigned different penalty values to indicate the relative importance of allowing them. Permissive controller synthesis amounts to finding a multi-strategy whose total incurred penalty is minimal, or below some given threshold.

Next, we propose practical methods for synthesising multi-strategies using mixed-integer linear programming (MILP) (see Section 3.7). We give an exact encoding for deterministic multi-strategies and an approximation scheme (with adaptable precision)

for the randomised case. Finally, we implement our techniques and evaluate their effectiveness on a range of case studies.

The chapter is structured as follows. We start by formalising the notions of multi-strategies and penalties, and define the permissive controller synthesis problem. In the following sections, we discuss the expressiveness of different types of multi-strategies and prove that synthesising a multi-strategy is an NP-hard problem. After presenting the theoretical properties of multi-strategies, we show a practical MILP encoding along with several optimisations that allow for multi-strategies to be used in practice. The practicality of the results is described in the experimental section, where we analyse both the performance of the method, as well as explain how multi-strategies can provide new insights when analysing PRISM case studies.

6.2 Permissive Controller Synthesis

We now define the *permissive controller synthesis* problem, which generalises classical controller synthesis by producing *multi-strategies* that offer the controller flexibility about which actions to take in each state.

6.2.1 Multi-Strategies

Multi-strategies generalise the notion of strategies, as defined in Section 3.1.

Definition 1 (Multi-strategy)**.** *A (memoryless)* multi-strategy *for a game* G *is a function* $\theta : S_\Diamond \to Dist(2^A)$ *with* $\theta(s)(\emptyset) = 0$ *for all* $s \in S_\Diamond$*.*

As for strategies, a multi-strategy θ is deterministic if θ always returns a Dirac distribution, and randomised otherwise. We write $\Theta_{\mathsf{G}}^{det}$ and $\Theta_{\mathsf{G}}^{rand}$ for the sets of all deterministic and randomised multi-strategies in G, respectively.

A deterministic multi-strategy θ chooses a set of *allowed actions* in each state $s \in S_\Diamond$, i.e., those in the unique set $B \subseteq A$ for which $\theta(s)(B) = 1$. The remaining actions $A(s) \setminus B$ are said to be *blocked* in s. In contrast to classical controller synthesis, where a strategy σ can be seen as providing instructions about precisely which action to take in each state, in permissive controller synthesis a multi-strategy provides multiple actions, any of which can be taken. A randomised multi-strategy generalises this by selecting a set of allowed actions in state s randomly, according to distribution $\theta(s)$.

We say that a controller strategy σ *complies* with multi-strategy θ if it picks actions that are allowed by θ.

Definition 2 (Compliance)**.** *Let θ be a multi-strategy and σ a strategy for a game G. We say that σ is* compliant *(or that it* complies*) with θ, written $\sigma \triangleleft \theta$, if, for any state s and non-empty subset $B \subseteq A(s)$, there is a distribution $d_{s,B} \in Dist(B)$ such that, for all $a \in A(s)$, $\sigma(s)(a) = \sum_{B \ni a} \theta(s)(B) \cdot d_{s,B}(a)$.*

Now, we can define the notion of a *sound* multi-strategy, i.e., one that is guaranteed to satisfy a property ϕ when complied with.

Definition 3 (Sound multi-strategy)**.** *A multi-strategy θ for game G is* sound *for a property ϕ if any strategy σ that complies with θ is sound for ϕ.*

Example 22 Figure 6.1 shows a stochastic game G, with controller and environment player states drawn as diamonds and squares, respectively. It models the control of a robot moving between 4 locations ($\overline{s}, s_2, s_3, s_5$). When moving east ($\overline{s} \rightarrow s_2$ or $s_3 \rightarrow s_5$), it may be blocked by a second robot, depending on the position of the latter. If it is blocked, there is a chance that it does not successfully move to the next location. We use a reward structure *moves*, which assigns 1 to the controller actions $north_1$, $east_1$, $east_2$, $south_1$, $south_2$, and define property $\phi = \mathtt{R}^{moves}_{\leqslant 5}[\mathtt{C}]$, meaning that the expected number of moves to reach s_5 is at most 5.

The strategy that picks $south_1$ in $\overline{s}$ and $east_2$ in s_3 results in an expected reward of 3.5 (i.e., 3.5 moves on average to reach s_5). The strategy that picks $east_1$ in $\overline{s}$ and $south_2$ in s_2 yields expected reward 5. Finally, the strategy that picks $south_1$ in $\overline{s}$ and $north_1$ in s_3 has an expected reward of 15.16, which does not satisfy the bound.

Thus, if we combine the first two strategies, we obtain a (deterministic) *multi-strategy* θ that picks $\{south_1, east_1\}$ in $\overline{s}$, $\{south_2\}$ in s_2 and $\{east_2\}$ in s_3 and is sound for ϕ since the expected reward is always at most 5.

6.2.2 Penalties and Permissivity

The motivation for multi-strategies is to offer flexibility in the actions to be taken, while still satisfying a particular property ϕ. Generally, we want a multi-strategy θ to be as *permissive* as possible, i.e. to impose as few restrictions as possible on actions to be taken. We formalise the notion of permissivity by assigning *penalties* to actions in the model, which we then use to quantify the extent to which actions are blocked by θ. Penalties provide expressivity in the way that we quantify permissivity: if it is more preferable that certain actions are allowed than others, then these can be assigned higher penalty values. To formalise the notion of penalties we define the *penalty scheme*.

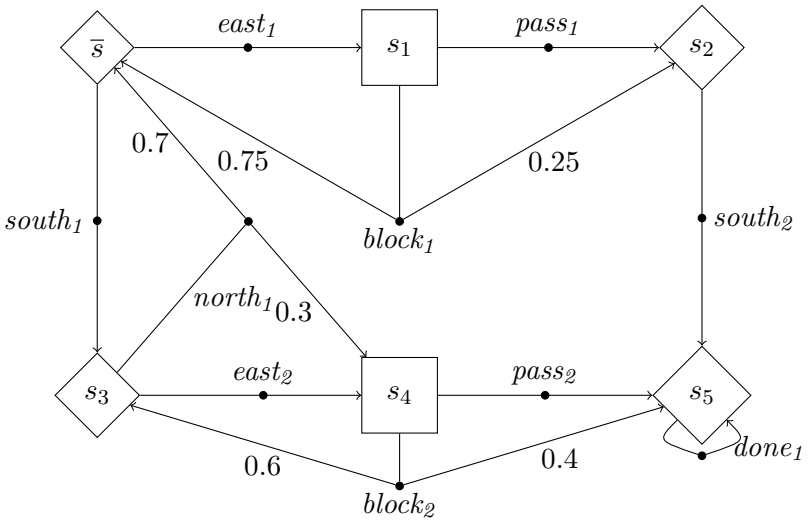


Figure 6.1: A stochastic game G for Example 22.

Definition 4 (Penalty scheme)**.** *A penalty scheme is a pair (ψ, t), comprising a penalty function $\psi : S_\Diamond \times A \to \mathbb{R}_{\geqslant 0}$ and a penalty type $t \in \{sta, dyn\}$.*

The function ψ represents the impact of blocking each action in each controller state of the game. The type t dictates how penalties for individual actions are combined to quantify the permissiveness of a specific multi-strategy. For *static penalties* ($t = sta$), we simply sum penalties across all states of the model. For *dynamic penalties* ($t = dyn$), we take into account the likelihood that blocked actions would actually have been available, by using the *expected sum* of penalty values. Before we define the notion of static and dynamic penalties, we first introduce the *local* penalty.

Definition 5 (Local penalty)**.** *The local penalty for θ and ψ at the state s is:*

$$pen_{loc}(\psi, \theta, s) = \sum_{B \subseteq A(s) a \notin B} \sum \theta(s, B)\psi(s, a).$$

If θ is deterministic, $pen_{loc}(\psi, \theta, s)$ is simply the sum of the penalties of actions that are blocked by θ in s. If θ is randomised, $pen_{loc}(\psi, \theta, s)$ gives the expected penalty value in s, i.e. the sum of penalties weighted by the probability with which θ blocks them in s.

Definition 6 (Static penalty)**.** *The static penalty of multi-strategy θ and penalty function ψ is given by:*

$$pen_{sta}(\psi, \theta) = \sum_{s \in S_\Diamond} pen_{loc}(\psi, \theta, s).$$

For the dynamic case, we use the (worst-case) expected sum of local penalties. We define a reward structure ψ^θ_{rew} given by the local penalties: $\psi^\theta_{rew}(s, a) = pen_{loc}(\psi, \theta, s)$ for all $a \in A(s)$.

Definition 7 (Dynamic penalty)**.** *Dynamic penalty of multi-strategy θ and a penalty function ψ is given by:*

$$pen_{dyn}(\psi, \theta, s) = \sup\{E^{\sigma,\pi}_{\mathsf{G},s}(\psi^{\theta}_{rew}) \mid \sigma \in \Sigma^{\Diamond}_{\mathsf{G}}, \pi \in \Sigma^{\Box}_{\mathsf{G}} \text{ and } \sigma \text{ complies with } \theta\}.$$

We use $pen_{dyn}(\psi, \theta) = pen_{dyn}(\psi, \theta, \overline{s})$ to reference the dynamic penalty in the initial state.

6.2.3 Permissive Controller Synthesis

We can now formally define the central problem studied in this chapter.

Definition 8 (Permissive controller synthesis)**.** *Consider a game G, a class of multi-strategies $\star \in \{det, rand\}$, a property ϕ, a penalty scheme (ψ, t) and a threshold $c \in \mathbb{Q}_{\geqslant 0}$. The* permissive controller synthesis *problem asks: does there exist a multi-strategy $\theta \in \Theta^{\star}_{\mathsf{G}}$ that is sound for ϕ and satisfies $pen_t(\psi, \theta) \leqslant c$?*

Alternatively, in a more quantitative fashion, we can aim to synthesise (if it exists) an *optimally permissive* sound multi-strategy.

Definition 9 (Optimally permissive)**.** *Let G, $\star$, ϕ and (ψ, t) be as in Definition 8. A sound multi-strategy $\hat{\theta} \in \Theta^{\star}_{\mathsf{G}}$ is* optimally permissive *if its penalty $pen_t(\psi, \hat{\theta})$ equals $\inf\{pen_t(\psi, \theta) \mid \theta \in \Theta^{\star}_{\mathsf{G}}$ and θ is sound for $\phi\}$.*

Example 23 We return to Example 22 and consider a static penalty scheme (ψ, sta) assigning 1 to the actions $north_1$, $east_1, east_2$, $south_1, south_2$. The deterministic multi-strategy θ from Example 22 is optimally permissive for $\phi = \mathtt{R}^{moves}_{\leqslant 5}[\mathtt{C}]$, with penalty 1 (just $north_1$ in s_3 is blocked). If we instead use $\phi' = \mathtt{R}^{moves}_{\leqslant 16}[\mathtt{C}]$, the multi-strategy θ' that extends θ by also allowing $north_1$ is now sound and optimally permissive, with penalty 0. Alternatively, the randomised multi-strategy θ'' that picks $0.7{:}\{east_2\}{+}0.3{:}\{north_1, east_2\}$ in s_3 is sound for ϕ with penalty just 0.7.

Next, we establish several fundamental results about the permissive controller synthesis problem. Technical proofs can be found in Appendix B.

Optimality

Recall that two key parameters of the problem are the type of multi-strategy sought (deterministic or randomised) and the type of penalty scheme used (static or dynamic). We first note that *randomised* multi-strategies are strictly more powerful than deterministic ones, i.e. they can be more permissive (yield a lower penalty) for the same property ϕ.

Theorem 2. *The answer to a permissive controller synthesis problem (for either a* static *or* dynamic *penalty scheme) can be "no" for* deterministic *multi-strategies, but "yes" for* randomised *ones.*

Proof. Consider an MDP with states s, t_1 and t_2, and actions a_1 and a_2, where $\delta(s, a_i)(t_i) = 1$ for $i \in \{1, 2\}$, and t_1, t_2 have self-loops only. Let r be a reward structure assigning 1 to (s, a_1) and 0 to all other state-action pairs, and ψ be a penalty function assigning 1 to (s, a_2) and 0 elsewhere. We then ask whether there is a multi-strategy satisfying $\phi = \mathtt{R}^r_{\geqslant 0.5}[\mathtt{C}]$ and with penalty of at most 0.5.

Considering either static or dynamic penalties, the randomised multi-strategy θ that chooses distribution $0.5{:}\{a_1\} + 0.5{:}\{a_2\}$ in s is sound and yields penalty of 0.5. However, there is no such deterministic multi-strategy.

□

This is why we explicitly distinguish between classes of multi-strategies when defining permissive controller synthesis. This situation contrasts with classical controller synthesis, where deterministic strategies are optimal for the same classes of properties ϕ. Intuitively, randomisation is more powerful in this case because of the trade-off between rewards and penalties. Similar results exist in, for example, multi-objective controller synthesis for MDPs [53].

Second, we observe that, for the case of static penalties, the optimal penalty value for a given property (the infimum of achievable values) may not actually be achievable by any randomised multi-strategy.

Theorem 3. *For permissive controller synthesis using a* static *penalty scheme, an optimally permissive* randomised *multi-strategy does not always exist.*

Proof. Consider a stochastic game with states s and t, and actions a and b, where we define $\delta(s, a)(s) = 1$ and $\delta(s, b)(t) = 1$, and t has just a self-loop. The reward structure r assigns 1 to (s, b) and 0 to all other state-action pairs. The penalty function ψ assigns 1 to (s, a) and 0 elsewhere.

Now observe that any multi-strategy which blocks the action a with probability $\varepsilon > 0$ and does not block any other actions incurs penalty ε and is sound for $\mathtt{R}^r_{\geqslant 1}[\mathtt{C}]$ since any strategy which complies with the multi-strategy satisfies that the action b is taken eventually. Thus, the infimum of achievable penalties is 0. However, the multi-strategy that incurs penalty 0, i.e. does not block any actions, is not sound for $\mathtt{R}^r_{\geqslant 1}[\mathtt{C}]$.

□

If, on the other hand, we restrict our attention to deterministic strategies, then an optimally permissive multi-strategy *does* always exist (since the set of deterministic, mem-

oryless multi-strategies is finite). For randomised multi-strategies with dynamic penalties, the question remains open.

Complexity

Next, we present complexity results for the different variants of the permissive controller synthesis problem. The complete proofs can be found in Appendix B. We begin with lower bounds.

Theorem 4. *The permissive controller synthesis problem is NP-hard, for either* static *or* dynamic *penalties, and* deterministic *or* randomised *multi-strategies.*

We prove NP-hardness by reduction from the Knapsack problem, where weights of items are represented by penalties, and their values are expressed in terms of rewards to be achieved. The most delicate part is the proof for randomised strategies, where we need to ensure that the multi-strategy cannot benefit from picking certain actions (corresponding to items being put to the knapsack) with probability other than 0 or 1. For upper bounds, we have the following.

Theorem 5. *The permissive controller synthesis problem for* deterministic *(resp.* randomised*) strategies is in NP (resp. PSPACE) for* dynamic/static *penalties.*

For deterministic multi-strategies, it is straightforward to show NP membership in both the dynamic and static penalty case, since we can guess a multi-strategy satisfying the required conditions and check its correctness in polynomial time. For randomised multi-strategies, with some technical effort we can encode existence of the required multi-strategy as a formula of the existential fragment of the theory of real arithmetic, and solvable with polynomial space [27].

A question is whether the PSPACE upper bound for randomised multi-strategies can be improved. We show that this is likely to be difficult, by giving a reduction from the square-root-sum problem. We use a variant of the problem that asks, for positive rationals $x_1,\ldots,x_n$ and y, whether $\sum_{i=1}^{n} \sqrt{x_i} \leqslant y$. This problem is known to be in PSPACE, but establishing a better complexity bound is a long-standing open problem in computational geometry [63].

Theorem 6. *There is a reduction from the square-root-sum problem to the permissive controller synthesis problem with* randomised *multi-strategies, for both* static *and* dynamic *penalties.*

6.3 MILP-Based Synthesis of Multi-Strategies

We now consider practical methods for synthesising multi-strategies that are sound for a property ϕ and optimally permissive for some penalty scheme. Our methods use mixed-integer linear programming (MILP) (see Section 3.7), which optimises an objective function subject to linear constraints that mix both real and integer variables. A variety of efficient, off-the-shelf MILP solvers exists.

An important feature of the MILP solvers is that they work incrementally, producing a sequence of increasingly good solutions. Here, that means generating a series of sound multi-strategies that are increasingly permissive. In practice, when resources are constrained, it may be acceptable to stop early and accept a multi-strategy that is sound but not necessarily optimally permissive.

It is possible to devise a similar encoding to the one that we present but for an SMT solver. From our experience SMT solvers are less efficient than MILP solvers when challenged with a task of minimising an objective function. Therefore, our decision was to purely focus on MILP-based solution.

We begin by discussing the synthesis of deterministic multi-strategies, first for static penalties and then for dynamic penalties. Subsequently, we present an approach to synthesising approximations to optimal randomised multi-strategies. In each case, we describe encodings into MILP problems and prove their correctness. We conclude this section with a brief discussion of ways to optimise the MILP encodings. Then, in Section 6.4, we investigate the practical applicability of our techniques.

Here, and in the rest of this section, we assume that the property ϕ is of the form $\mathtt{R}^{r}_{\geqslant b}[\,\mathtt{C}\,]$. For the upper bounds on expected rewards ($\phi = \mathtt{R}^{r}_{\leqslant b}[\,\mathtt{C}\,]$) a similar encoding follows. For the purposes of encoding into MILP, we rescale r and b such that $\sup_{\sigma,\pi} E^{\sigma,\pi}_{\mathsf{G},s}(r) < 1$ for all s, and rescale every (non-zero) penalty such that $\psi(s,a) \geqslant 1$ for all s and $a \in A(s)$.

6.3.1 Deterministic Multi-Strategies with Static Penalties

Figure 6.2 shows an encoding into MILP of the problem of finding an optimally permissive *deterministic* multi-strategy for property $\phi = \mathtt{R}^{r}_{\geqslant b}[\,\mathtt{C}\,]$ and a *static* penalty scheme (ψ, sta). The encoding uses 5 types of variables: $y_{s,a} \in \{0,1\}$, $x_s \in [0,1]$, $\alpha_s \in \{0,1\}$, $\beta_{s,a,t} \in \{0,1\}$ and $\gamma_t \in [0,1]$, where $s,t \in S$, and $a \in A$. The worst-case size of the MILP problem is $\mathcal{O}(|A|{\cdot}|S|^2{\cdot}\kappa)$, where κ stands for the longest encoding of a number used.

Variables $y_{s,a}$ encode a multi-strategy θ as follows: $y_{s,a}$ has value 1 iff θ allows action a in s (constraint (6.2) enforces at least one allowed action per state). Variables x_s represent

Minimise: $-x_{\overline{s}} + \sum_{s \in S_\Diamond} \sum_{a \in A(s)} (1 - y_{s,a}) \cdot \psi(s,a)$ subject to:

$$x_{\overline{s}} \geqslant b \tag{6.1}$$

$$1 \leqslant \sum_{a \in A(s)} y_{s,a} \qquad \text{for all } s \in S_\Diamond \tag{6.2}$$

$$x_s \leqslant \sum_{t \in S} \delta(s,a)(t) \cdot x_t + r(s,a) + (1 - y_{s,a}) \qquad \text{for all } s \in S_\Diamond,\ a \in A(s) \tag{6.3}$$

$$x_s \leqslant \sum_{t \in S} \delta(s,a)(t) \cdot x_t \qquad \text{for all } s \in S_\Box,\ a \in A(s) \tag{6.4}$$

$$x_s \leqslant \alpha_s \qquad \text{for all } s \in S \tag{6.5}$$

$$y_{s,a} = (1 - \alpha_s) + \sum_{t \in supp(\delta(s,a))} \beta_{s,a,t} \qquad \text{for all } s \in S, a \in A(s) \tag{6.6}$$

$$y_{s,a} = 1 \qquad \text{for all } s \in S_\Box, a \in A(s) \tag{6.7}$$

$$\gamma_t < \gamma_s + (1 - \beta_{s,a,t}) + r(s,a) \qquad \text{for all } (s,a,t) \in supp(\delta) \tag{6.8}$$

Figure 6.2: MILP encoding for deterministic multi-strategies with static penalties.

Minimise: $z_{\overline{s}}$ subject to (6.1),...,(6.8) and:

$$\ell_s = \sum_{a \in A(s)} \psi(s,a) \cdot (1 - y_{s,a}) \qquad \text{for all } s \in S_\Diamond \tag{6.9}$$

$$z_s \geqslant \sum_{t \in S} \delta(s,a)(t) \cdot z_t + \ell_s - c \cdot (1 - y_{s,a}) \qquad \text{for all } s \in S_\Diamond,\ a \in A(s) \tag{6.10}$$

$$z_s \geqslant \sum_{t \in S} \delta(s,a)(t) \cdot z_t \qquad \text{for all } s \in S_\Box,\ a \in A(s) \tag{6.11}$$

Figure 6.3: MILP encoding for deterministic multi-strategies with dynamic penalties.

the worst-case expected total reward (for r) from state s, under any controller strategy complying with θ and under any environment strategy. This is captured by constraints (6.3)–(6.4) (which amounts to minimising the reward in an MDP). Constraint (6.1) imposes the required bound of b on the reward from $\overline{s}$.

The objective function minimises the static penalty (the sum of all local penalties) minus the expected reward in the initial state. The latter acts as a tie-breaker between solutions with equal penalties (but, thanks to rescaling, is always dominated by the penalties and therefore does not affect optimality).

As an additional technicality, we need to ensure the values of x_s are the *least* solution of the defining inequalities, to deal with the possibility of zero reward loops. To achieve this, we use an approach similar to the one taken in [121]. It is sufficient to ensure that $x_s = 0$ whenever the minimum expected reward from s under θ is 0, which is true if and only if, starting from s, it is possible to avoid ever taking an action with positive reward.

In our encoding, $\alpha_s = 1$ if x_s is positive (constraint (6.5)). The binary variables $\beta_{s,a,t} = 1$ represent, for each such s and each action a allowed in s, a choice of successor $t = t(s,a) \in \mathit{supp}(\delta(s,a))$ (constraint (6.6)). The variables γ_s then represent a ranking function: if $r(s,a) = 0$, then $\gamma_s > \gamma_{t(s,a)}$ (constraint (6.8)). If a positive reward could be avoided starting from s, there would in particular be an infinite sequence $s_0, a_1, s_1, \ldots$ with $s_0 = s$ and, for all i, either (i) $x_{s_i} > x_{s_{i+1}}$, or (ii) $x_{s_i} = x_{s_{i+1}}$, $s_{i+1} = t(s_i, a_i)$ and $r(s_i, a_i) = 0$, and therefore $\gamma_{s_i} > \gamma_{s_{i+1}}$. Since the set of states S is finite, this sequence would have to enter a loop, leading to a contradiction.

Example 24 In Figure 6.4, we show the MILP encoding for deterministic multi-strategies with static penalties for the model from Figure 6.1. For readability reasons, we will only show constraints handling zero reward loops (i.e. constraints from lines 6.5 – 6.8) for states that have zero reward loops, that is, state s_5.

In our example, we scaled the rewards by dividing them by 100, such that the expected reward for any state is always smaller than one. We consider a new property, $\phi = \mathtt{R}^{\mathit{moves}}_{\geqslant 0.05}[\,\mathtt{C}\,]$, that guarantees that the robot takes at *least* 0.05 moves to reach s_5. An optimally permissive multi-strategy satisfying ϕ picks $\{\mathit{east}_1, \mathit{south}_1\}$ in $\overline{s}$, south_2 in s_2, and north_1 in s_3.

An equivalent solution to the MILP problem would assign $y_{\overline{s},\mathit{east}_1} = 1$, $y_{\overline{s},\mathit{south}_1} = 1$, $y_{s_2,\mathit{south}_2} = 1$, $y_{s_3,\mathit{north}_1} = 1$, $y_{s_3,\mathit{east}_2} = 0$, and $y_{s_5,\mathit{done}_1} = 1$. For the state s_5, we have $\alpha_{s_5} = 0$, $\gamma_{s_5} = 0$, and $\beta_{s_5,\mathit{done}_1,s_5} = 0$, which forces the expected total reward in that state to be equal to zero.

Correctness

Below, we present the correctness proof for the encoding from Figure 6.2.

Theorem 7. *Let* G *be a game,* $\phi = \mathtt{R}^{r}_{\geqslant b}[\,\mathtt{C}\,]$ *a property and* (ψ, sta) *a* static *penalty scheme. There is a sound multi-strategy for* ϕ *with penalty* p *if and only if any optimal assignment to the MILP instance from Figure 6.2 satisfies* $p = \sum_{s \in S_\Diamond} \sum_{a \in A(s)} (1 - y_{s,a}) \cdot \psi(s,a)$.

The correctness proof is divided into two parts. First, we show how a sound multi-strategy yields an assignment to variables that satisfy the constraints from Figure 6.2. In the other direction, given a satisfying assignment from the encoding, we show how to build a corresponding multi-strategy.

The key observation that we make in the proof is that, by restricting a stochastic game to choices that are allowed by the multi-strategy, we end up with solving an MDP. This allows us to use some of the standard results [107] for MDPs and LP. A full proof can be found in Appendix B.

$$
\begin{aligned}
\text{Minimise: } & -x_{\overline{s}} + \\
& (1 - y_{\overline{s},east_1}) + (1 - y_{\overline{s},south_1}) + (1 - y_{s_2,south_2}) + \\
& (1 - y_{s_3,north_1}) + (1 - y_{s_3,east_2}) \quad \text{subject to:}
\end{aligned}
$$

$$
\begin{aligned}
x_{\overline{s}} &\geqslant 0.05 \\
1 &\leqslant y_{\overline{s},east_1} + y_{\overline{s},south_1} \\
1 &\leqslant y_{s_2,south_2} \\
1 &\leqslant y_{s_3,north_1} + y_{s_3,east_2} \\
1 &\leqslant y_{s_5,done_1}
\end{aligned}
$$

$$
\begin{aligned}
x_{\overline{s}} &\leqslant x_{s_1} + 0.01 + (1 - y_{\overline{s},east_1}) \\
x_{\overline{s}} &\leqslant x_{s_3} + 0.01 + (1 - y_{\overline{s},south_1}) \\
x_{s_2} &\leqslant x_{s_5} + 0.01 + (1 - y_{s_2,south_2}) \\
x_{s_3} &\leqslant 0.7 \cdot x_{\overline{s}} + 0.3 \cdot x_{s_4} + 0.01 + (1 - y_{s_3,north_1}) \\
x_{s_3} &\leqslant x_{s_4} + 0.01 + (1 - y_{s_3,east_2})
\end{aligned}
$$

$$
\begin{aligned}
x_{s_1} &\leqslant x_{s_2} \\
x_{s_1} &\leqslant 0.75 \cdot x_{\overline{s}} + 0.25 \cdot x_{s_2} \\
x_{s_4} &\leqslant x_{s_5} \\
x_{s_4} &\leqslant 0.6 \cdot x_{s_3} + 0.4 \cdot x_{s_5}
\end{aligned}
$$

$$
\begin{aligned}
x_{s_5} &\leqslant \alpha_{s_5} \\
y_{s_5,done_1} &= (1 - \alpha_{s_5}) + \beta_{s_5,done_1,s_5} \\
\gamma_{s_5} &< \gamma_{s_5} + (1 - \beta_{s_5,done_1,s_5})
\end{aligned}
$$

Figure 6.4: MILP encoding for the model from Figure 6.1.

6.3.2 Deterministic Multi-Strategies with Dynamic Penalties

Next, we show how to compute a sound and optimally permissive *deterministic* multi-strategy for a *dynamic* penalty scheme (ψ, *dyn*). This case is more subtle since the optimal penalty can be infinite. Hence, our solution proceeds in two steps as follows.

Initially, we determine if there is *some* sound multi-strategy. For this, we just need to check for the existence of a sound strategy using standard algorithms for solution of stochastic games [42, 55]. If there is no sound multi-strategy, we are done. Otherwise, we use the MILP problem in Figure 6.3 to determine the penalty for an optimally permissive sound multi-strategy. This MILP encoding extends the one in Figure 6.2 for static penal-

ties, adding variables ℓ_s and z_s, representing the local and the expected penalty in state s, and three extra sets of constraints. First, (6.9) and (6.10) define the expected penalty in controller states, which is the sum of penalties for all disabled actions and those in the successor states, multiplied by their transition probabilities. The behaviour of the environment states is then captured by constraint (6.11), where we only maximise the penalty, without incurring any penalty locally.

The constant c in (6.10) is chosen to be no lower than any *finite* penalty achievable by a deterministic multi-strategy, a possible value being:

$$\sum_{i=0}^{\infty}(1-p^{|S|})^i \cdot p^{|S|} \cdot i \cdot |S| \cdot pen_{\max} \tag{6.12}$$

where p is the smallest non-zero probability assigned by δ, and $pen_{\max}$ is the maximal local penalty over all states. To see that (6.12) gives a safe bound on c, observe that, for the penalty to be finite under a deterministic multi-strategy, for every state s there must be a path of length at most $|S|$ to a state from which no penalty will be incurred.

If the MILP problem has a solution, this is the optimal dynamic penalty over all sound multi-strategies. If not, no deterministic sound multi-strategy has finite penalty and the optimal penalty is ∞ (recall that we already established there is *some* sound multi-strategy). In practice, we might choose a lower value of c than the one above, resulting in a multi-strategy that is sound, but possibly not optimally permissive.

Correctness

Formally, correctness of the MILP encoding for the case of dynamic penalties is captured by the following theorem.

Theorem 8. *Let* G *be a game,* $\phi = \mathtt{R}^{r}_{\geqslant b}[\,\mathtt{C}\,]$ *a property and* (ψ, dyn) *a* dynamic *penalty scheme. Assume there is a sound multi-strategy for* ϕ*. The MILP formulation from Figure 6.3 satisfies: (a) there is no solution if and only if the optimally permissive deterministic multi-strategy yields infinite penalty; and (b) there is a solution* $\bar{z}_{\bar{s}}$ *if and only if an optimally permissive deterministic multi-strategy yields penalty* $\bar{z}_{\bar{s}}$*.*

The proof is an extension of the proof for Theorem 7 and can be found Appendix B.

6.3.3 Approximating Randomised Multi-Strategies

In Section 6.2.3, we showed that randomised multi-strategies can outperform deterministic ones. The MILP encodings in Figures 6.2 and 6.3, though, cannot be adapted to the randomised case, since this would need non-linear constraints.

Instead, in this section, we propose an *approximation* which finds the optimal randomised multi-strategy θ in which each probability $\theta(s, B)$ is a multiple of $\frac{1}{M}$ for a given *granularity* M. Any such multi-strategy can then be simulated by a deterministic one on a transformed game, allowing synthesis to be carried out using the MILP-based methods described in the previous section. Before giving the definition of the transformed game, we show that we can simplify our problem by restricting to multi-strategies which in any state select at most two actions with non-zero probability.

Theorem 9. *For a (static or dynamic) penalty scheme (ψ, t) and any sound multi-strategy θ we can construct another sound multi-strategy θ' such that $pen_t(\psi, \theta) \geqslant pen_t(\psi, \theta')$ and $|supp(\theta'(s))| \leqslant 2$ for any $s \in S_\Diamond$.*

In the proof we use a geometrical representation where the reward and penalty achieved by each set of actions are represented by a point in the two-dimensional plane. We show that for every multi-strategy a convex combination of two points yields a randomised multi-strategy with optimal penalty. The points specify the sets of actions on which we randomise, while the coefficients specify the probability for each set to be chosen. A more detailed presentation of the proof can be found in Appendix B.

The result in Theorem 9 allows us to simplify the game construction that we use to map between (discretised) randomised multi-strategies and deterministic ones. The transformation is illustrated in Figure 6.5. The left-hand side shows a controller state $s \in S_\Diamond$ in the original game (i.e., the game for which we are seeking randomised multi-strategies). For each such state, we add the two layers of states illustrated on the right-hand side of the figure: *gadgets* s'_1, s'_2 representing the two subsets $B \subseteq A(s)$ with $\theta(s, B) > 0$, and *selectors* s_i (for $1 \leqslant i \leqslant m$), which distribute probability among the two gadgets. The s_i are reached from s via a transition using fixed probabilities $p_1, \ldots, p_m$ which need to be chosen appropriately. For efficiency, we want to minimise the number of selectors m for each state s.

We let $m = \lfloor 1 + \log_2 M \rfloor$ and $p_i = \frac{l_i}{M}$, where $l_1 \ldots, l_m \in \mathbb{N}$ are defined recursively as follows: $l_1 = \lceil \frac{M}{2} \rceil$ and $l_i = \lceil \frac{M-(l_1+\cdots+l_{i-1})}{2} \rceil$ for $2 \leqslant i \leqslant m$. This allows us to encode any probability distribution $(\frac{l}{M}, \frac{M-l}{M})$ between two subsets B_1 and B_2.

Our next goal is to show that, by varying the granularity M, we can get arbitrarily close to the optimal penalty for a randomised multi-strategy and, for the case of static penalties, define a suitable choice of M. This will be formalised in Theorem 11. First, we need to establish the following intermediate result, stating that, in the static case, in addition to Theorem 9, we can require the action subsets allowed by a multi-strategy to be ordered with respect to the subset relation.

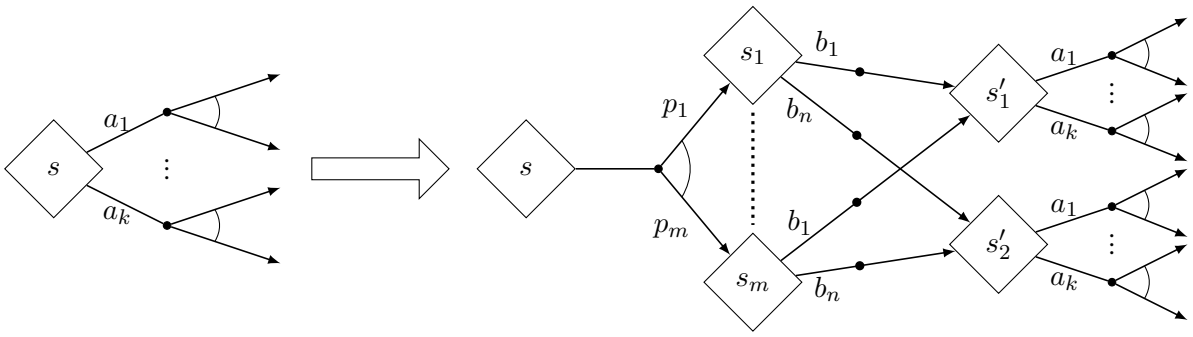


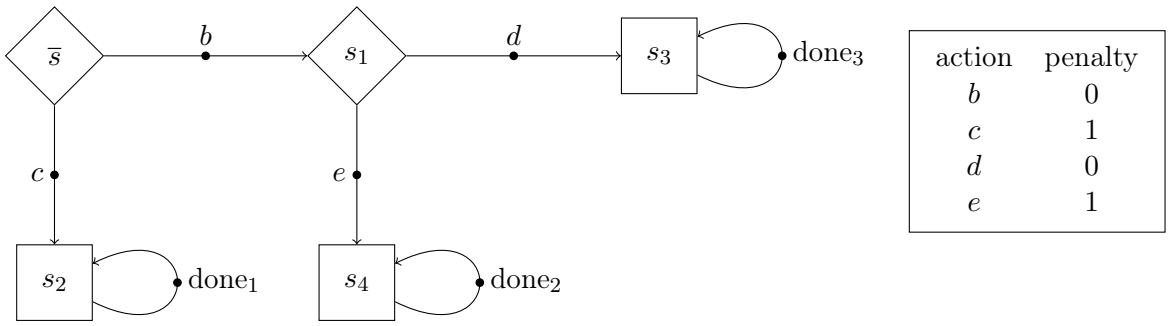
Figure 6.5: A node in the original game (left), and the corresponding nodes in the transformed game for approximating randomised multi-strategies (right).

Theorem 10. *For a static penalty scheme* (ψ, sta) *and any sound multi-strategy* θ*, we can construct another sound multi-strategy* θ' *such that, for each* $s \in S_{\Diamond}$*, if* $supp(\theta'(s))=\{B, C\}$*, then either* $B \subseteq C$ *or* $C \subseteq B$*.*

Theorem 9 states that it is enough for a multi-strategy to allow only two sets of actions. Each of those sets may achieve a different expected reward. In the proof for Theorem 10, we show that by adding actions from the set that achieves the higher expected reward to the set that achieves the lower expected reward we will not affect the reward nor we will increase the overall penalty. This operation makes one set a subset of the other set. More explanation of the proof details can be found in Appendix B.

For the sake of completeness, in Example 25 we show that Theorem 10 does *not* extend to dynamic penalties. The key observation is that increasing the probability of allowing an action can lead to an increased penalty if one of the successor states has a high expected penalty.

Example 25 Let us consider stochastic game from Figure 6.6, for which we want to reach the goal state s_3 with probability of at least 0.5.



action	penalty
b	0
c	1
d	0
e	1

Figure 6.6: Counterexample for Theorem 10 in case of dynamic penalties.

This implies $\theta(\overline{s},\{b\})\cdot\theta(s_1,\{d\})\geqslant 0.5$, and so $\theta(\overline{s},\{b\})>0, \theta(s_1,\{d\})>0$. If θ satisfies the condition of Theorem 10, then $\theta(\overline{s},\{c\}) = \theta(s_1,\{e\}) = 0$, so an opponent can always use b, forcing an expected penalty of $\theta(\overline{s},\{b\}) + \theta(s_1,\{d\})$, for a minimal value of $\sqrt{2}$. However, the sound multi-strategy θ with $\theta(\overline{s},\{b\})=\theta(\overline{s},\{c\})=0.5$ and $\theta(s_1,\{d\})=1$ achieves a dynamic penalty of just 1.

We can now return to the issue of how to vary the granularity M to get sufficiently close to the optimal penalty, we formalise this as follows.

Theorem 11. *Let θ be a sound multi-strategy. For any $\varepsilon > 0$, there is an M and a sound multi-strategy θ' of granularity M satisfying $pen_t(\psi,\theta') - pen_t(\psi,\theta) \leqslant \varepsilon$. Moreover, for static penalties it suffices to take $M = \lceil \sum_{s\in S, a\in A(s)} \frac{\psi(s,a)}{\varepsilon} \rceil$.*

The proof of Theorem 11 follows a simple intuition that by making small local changes to the multi-strategy we will not significantly change its overall reward or penalty. The cases for static and dynamic penalties are proved separately. The static case is trivial as the differences in the penalty value can be easily summed over all states. The dynamic case is slightly more complex but follows the same intuition. A full proof is available in Appendix B.

Example 26 We present the approximation for the model from the Figure 6.1. We choose an error of $\varepsilon = \frac{1}{10}$ and from Theorem 11 compute $M = 50$. For every controller state we have $m = 6$ selectors where $p_1 = \frac{25}{50}, p_2 = \frac{13}{50}, p_3 = \frac{6}{50}, p_4 = \frac{3}{50}, p_5 = \frac{2}{50}, p_6 = \frac{1}{50}$. The reachable state space of the model with gadgets now contains 22 states and we omitted building gadgets for states with only one action. The randomised multi-strategy θ'' from Example 23 can now be approximated with θ''' which in s_3 picks $(\frac{25}{50} + \frac{6}{50} + \frac{3}{50} + \frac{1}{50})$:$\{east_2\}$+$(\frac{13}{50} + \frac{2}{50})$:$\{north_1, east_2\}$ and incurs a penalty of 0.7. An example of an approximation gadget for the state $\overline{s}$ can be seen in Figure 6.7.

6.3.4 Optimisations

We conclude this section on MILP-based multi-strategy synthesis by presenting some optimisations that can be applied to our methods. The general idea is to add additional constraints to the MILP problems that will reduce the search space to be explored by a solver. We present two optimisations, targeting different aspects of our encodings: (i) possible variable values; and (ii) penalty structures.

Bounds on variable values. In our encodings, for the variables x_s, we only specified very general lower and upper bounds that would constrain its value. Narrowing the set of values that a variable may take can significantly reduce the search space and thus the

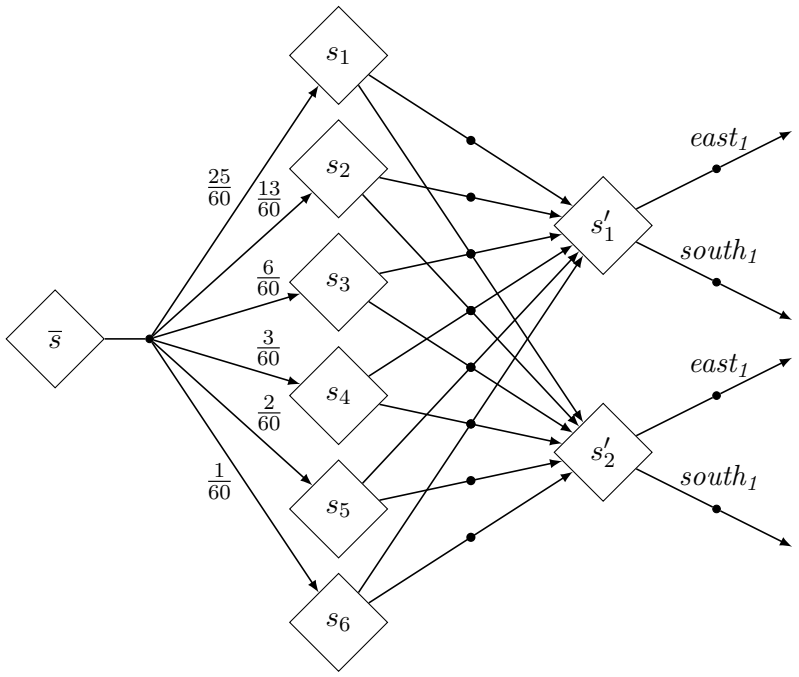


Figure 6.7: An approximation gadget for state $\overline{s}$ from Figure 6.1 and Example 26.

solution time required by an MILP solver. One possibility that works well in practice is to bound the values of the variables by the minimal and maximal expected reward achievable from the given state, i.e., add the constraints:

$$\inf_{\sigma\in\Sigma_{\mathsf{G}}^{\Diamond},\pi\in\Sigma_{\mathsf{G}}^{\Box}}\{E_{\mathsf{G},s}^{\sigma,\pi}(r)\} \leqslant x_s \leqslant \sup_{\sigma\in\Sigma_{\mathsf{G}}^{\Diamond},\pi\in\Sigma_{\mathsf{G}}^{\Box}}\{E_{\mathsf{G},s}^{\sigma,\pi}(r)\} \qquad \text{for all } s \in S$$

where both the infima and suprema above are constants obtained by applying standard probabilistic model checking algorithms.

Actions with zero penalties. Our second optimisation exploits the case where an action has zero penalty assigned to it. Intuitively, this action could always be disabled without harming the overall penalty of the multi-strategy. At the same time, enabling an action with zero penalty might be the only way to satisfy the property and therefore we cannot disable all such actions. Our optimisation exploits the fact that it is enough to allow at most one action that has zero penalty. For simplicity of the presentation, we assume $Z_s = \{a \in A(s) \,|\, \psi(s,a) = 0\}$; then formally we add the constraints:

$$\sum_{a\in Z_s} y_{s,a} \leqslant 1 \qquad \text{for all } s \in S_{\Diamond}.$$

6.4 Experimental Results

We have implemented our techniques within PRISM-games [31], an extension of the PRISM model checker for performing model checking and strategy synthesis on stochastic games. Prior to this thesis, PRISM-games could thus already be used for (classical) controller synthesis problems on stochastic games. We added to this the ability to synthesise multi-strategies using the MILP-based methods described in Section 6.3. Our implementation currently uses CPLEX [131] or Gurobi [132] to solve MILP problems. We run our experiments on a PC with a 2.8GHz Xeon processor and 32GB of RAM, running Fedora 14.

We investigated the applicability and performance of our approach on a variety of case studies, some of which are existing benchmark examples and some of which were developed for this thesis. These are described in Appendix A and the files used can be found online [127].

Deterministic multi-strategy synthesis

We first discuss the generation of optimal *deterministic* multi-strategies, the results of which are summarised in Table 6.1 and Table 6.2. Table 6.1 summarises the models and properties considered. The table gives: the case study name, any parameters used, the number of states ($|S|$) and of controller states ($|S_\Diamond|$), and property used (ϕ). The final column gives, for comparison purposes, the time required for performing classical (single) strategy synthesis on each model and property ϕ using value iteration. In Table 6.2 we show the case study name and the parameter value, and report the penalty value of the optimal multi-strategy and the time to generate it. We report the time needed for encoding without any optimisations, with only one optimisation used, and with both optimisations used. For the purpose of brevity of presentation, only the last result is reported for both CPLEX and Gurobi. Below, we give further details for each case study, illustrating the variety of ways that permissive controller synthesis can be used.

cloud: We adapt the PRISM model from [21] to synthesise deployment of services across virtual machines (VMs) in a cloud infrastructure. Our property ϕ specifies that, with high probability, services are deployed to a preferred subset of VMs, and we then assign unit (dynamic) penalties to all actions corresponding to deployment on this subset. The resulting multi-strategy has very low expected penalty (see Table 6.2), indicating that the goal ϕ can be achieved whilst the controller experiences reduced flexibility only on executions with low probability.

android: We apply permissive controller synthesis to a model created for runtime control of

Name [parameters]	Parameter values	States	Controller states	Property	Strat. synth. time (s)
cloud	5	8,841	2,177	$\mathtt{P}_{\geqslant 0.9999}[\mathtt{F}\ \mathit{deployed}]$	0.1
$[vm]$	6	34,953	8,705	$\mathtt{P}_{\geqslant 0.999}[\mathtt{F}\ \mathit{deployed}]$	0.1
android_3	1, 48	2,305	997	$\mathtt{R}^{\mathit{time}}_{\leqslant 10000}[\mathtt{C}]$	0.2
$[r, s]$	2, 48	9,100	3,718		0.6
	3, 48	23,137	9,025		0.9
mdsm	3	62,245	9,173	$\mathtt{P}_{\leqslant 0.1}[\mathtt{F}\ \mathit{deviated}]$	5.6
$[N]$	3	62,245	9,173	$\mathtt{P}_{\leqslant 0.01}[\mathtt{F}\ \mathit{deviated}]$	
investor	5,10	10,868	3,344	$\mathtt{R}^{\mathit{profit}}_{\geqslant 4.98}[\mathtt{C}]$	0.9
$[vinit, vmax]$	10, 15	21,593	6,644	$\mathtt{R}^{\mathit{profit}}_{\geqslant 8.99}[\mathtt{C}]$	2.2
team-form	3	12,476	2,023	$\mathtt{P}_{\geqslant 0.9999}[\mathtt{F}\ \mathit{done}_1]$	0.2
$[N]$	4	96,666	13,793		0.8
cdmsn $[N]$	3	1,240	604	$\mathtt{P}_{\geqslant 0.9999}[\mathtt{F}\ \mathit{prefer}_1]$	0.1

Table 6.1: Details of the models and properties used for experiments with deterministic multi-strategies, and execution times shown for classical (single) strategy synthesis.

Name [parameters]	Parameter values	Penalty	Multi-strategy synthesis time (s)				
			CPLEX				Gurobi
			No-opts	Bounds	Zero	Both	Both
cloud	5	0.001	2.5	3.4	13.0	10.4	1.5
$[vm]$	6	0.01	62.5	*	63.6	25.4	4.7
android_3	1, 48	0.0009	1.1	0.7	1.0	0.5	0.5
$[r, s]$	2, 48	0.0011	28.6	8.4	28.5	8.4	3.6
	3, 48	0.0013	*	13.4	*	13.3	47.6
mdsm	3	52	28.1	36.3	27.9	33.7	19.4
$[N]$	3	186	11.9	11.6	11.9	11.6	12.3
investor	5,10	1	68.6	131.4	68.9	131.4	12.0
$[vinit, vmax]$	10, 15	1	*	*	*	*	208.9
team-form	3	0.890	0.2	0.3	0.2	0.3	0.8
$[N]$	4	0.890	249.4	249.5	186.4	184.5	3.8
cdmsn $[N]$	3	2	0.6	0.6	0.6	0.6	1.7

* No optimal solution to MILP problem within 5 minute time-out.

Table 6.2: Experimental results for synthesis of optimal deterministic multi-strategies.

an Android application that provides real-time stock monitoring (see Chapter 8 for details). We extend the application to use multiple stock information providers and synthesise a multi-strategy which specifies an efficient runtime selection of providers (ϕ bounds the total expected response time). We use static penalties, assigning higher values to actions that select the two most efficient providers at each time point and synthesise a multi-strategy that always ensures a choice of at least two sources (in case one becomes unavailable), while preserving ϕ.

mdsm: Microgrid demand-side management (MDSM) is a randomised scheme for managing local energy usage. A stochastic game analysis [30] previously showed it is beneficial for users to selfishly deviate from the protocol. We synthesise a multi-strategy for a (potentially selfish) user, with the goal (ϕ) of bounding the probability of deviation (at either 0.1 or 0.01). The resulting multi-strategy could be used to modify the protocol, restricting the behaviour of this user to reduce selfish behaviour. To make the multi-strategy as permissive as possible, restrictions are only introduced where necessary to ensure ϕ. We also guide where restrictions are made by assigning (static) penalties at certain times of the day.

investor: This example [100] synthesises strategies for a futures market investor, who chooses when to reserve shares, operating in a (malicious) market which can periodically ban him from investing. We generate a multi-strategy that achieves 90% of the maximum expected profit (obtainable by a single strategy) and assign (static) unit penalties to all actions, showing that, after an immediate share purchase, the investor can choose his actions freely and still meet the 90% target.

team-form: This example [33] synthesises strategies for forming teams of agents in order to complete a set of collaborative tasks. Our goal (ϕ) is to guarantee that a particular task is completed with high probability (0.9999). We use (dynamic) unit penalties on all actions of the first agent and synthesise a multi-strategy representing several possibilities for this agent while still achieving the goal.

cdmsn: Lastly, we apply permissive controller synthesis to a model of a protocol for collective decision making in sensor networks (CDMSN) [30]. We synthesise strategies for nodes in the network such that consensus is achieved with high probability (0.9999). We use (static) penalties inversely proportional to the energy associated with each action a node can perform, to ensure that the multi-strategy favours more efficient solutions.

Performance and scalability

Unsurprisingly, permissive controller synthesis is more costly to execute than classical controller synthesis – this is clearly seen by comparing the times in the rightmost column

of Table 6.1 with the times in Table 6.2. The smallest slowdown has been observed for *mdsm* case study, where we run classical controller synthesis for 5.6 seconds and for 12.3 seconds for the permissive controller synthesis. In the worse case, we observe a two orders of magnitude slowdown for the *investor* case study.

Another difference is the scalability of our method. The computation time for classical controller synthesis scales better than the corresponding time for permissive controller synthesis. One example is the *investor* case study where we observe in Table 6.1 a three-fold increase in computation time when increasing the model size. When computing a permissive controller we observe almost a 20-fold increase in computation time.

The reasons for such differences can be attributed to the fact that computing a multi-strategy is a more complex problem than computing a single strategy. For example, for single strategies, by computing locally optimal value of the strategy, we can obtain a strategy that is also globally optimal. This is not possible for multi-strategies, where we might have to consider various trade-offs between states to obtain an optimally permissive strategy.

The performance and scalability of our method is affected (as usual) by the state space size. In particular, it is affected by the number of actions in controller states, and the number of target states. Each action in the model results in an integer variable, which is the most expensive part of the solution. On the other hand, each target state decreases the number of integer variables because for target states we simply assign value $x_s = 0$ without using any integer variables.

Apart from performance and scalability, there are other important differences between computing the classical and the permissive controllers. The results in Table 6.1 do not depend on the bound that is used in the property. This can be clearly seen for the *mdsm* example, where for different properties we observe the same computation time. For the $\mathtt{P}_{\leqslant 0.1}[\,\mathtt{F}\ \mathit{deviated}\,]$ and $\mathtt{P}_{\leqslant 0.01}[\,\mathtt{F}\ \mathit{deviated}\,]$ properties the classical controller synthesis involves performing the same computation, namely, computing the minimising controller. This is not the case for permissive controller synthesis, and for different properties we solve different MILP problems, resulting in different computation time.

The performance optimisations presented in Section 6.3.4 often allowed us to obtain an optimal multi-strategy more quickly. In many cases, it proved beneficial to apply both optimisations at the same time. In the best case ($\mathit{android\_3}, r{=}3, s{=}48$), an order of magnitude gain was observed. We reported a slowdown after applying optimisation in the case of the *investor* example. We attribute this to the fact that adding bounds on variable values can make finding the initial solution of the MILP problem harder, causing a slowdown of the overall solution process.

While the *bounds* optimisation can be applied automatically, the *zero* optimisation

depends on the domain knowledge of the person running the tool. Actions that are for some reason less important for the multi-strategy can be assigned penalty equal to zero. One example is the *mdsm* case study, where we set a zero penalty for actions that model accessing the power grid outside the specified time interval.

Both solvers were run using a single-threaded configuration and Gurobi proved to be a more efficient solver. Using multiple threads did not significantly improve the performance. In some cases it added randomness to the computation time when the solver would take longer time than usual. In the case of CPLEX, we observed worse numerical stability, causing it to return a sub-optimal multi-strategy as optimal. In the case of Gurobi, we did not see any such behaviour.

Randomised multi-strategy synthesis

We report the results for approximating optimal *randomised* multi-strategies by first summarising the considered models and properties in Table 6.3. In Table 6.4, we report the effects of adding approximation gadgets on the state space size of these case studies. We use a different set of case studies than for the deterministic multi-strategies. This is because for some models we were unable to generate randomised multi-strategies in reasonable time. We picked three different granularities, $M = 100, M = 200$ and $M = 300$; for higher values of M we did not observe improvements in the penalties of the generated multi-strategies.

Finally, in Table 6.5 we show penalties obtained by randomised multi-strategies. We compare the (static) penalty value of generated randomised strategies to the value obtained by optimal deterministic multi-strategies for the same models. Again, we impose a time-out of 5 minutes and use Gurobi as the MILP solver in every case, since it was shown above to perform better than CPLEX.

In Table 6.4, we can see that our approximation method added a considerable number of states to the model. The approximation is applied for every controller state and we observe a 3-fold increase in the state space size. An additional increase in the state space size occurs with increasing M. The size of the state space after the approximation is the major contributor to the worse performance of our method when compared to the encoding for deterministic strategies.

For the case studies we considered, we were able to generate a sound multi-strategy in every instance. In only one case our solver finished the computation before the 5 minute timeout. This was the case for the *android_3* case study ($r = 1, s = 1$), and the model consisted of only 49 states. In other cases our solver returned a possibly non-optimal multi-strategy (denoted by a * in Table 6.5). To obtain a sound but non-optimal multi-

Name [parameters]	Parameters values	States	Controller states	Property
android_3 $[r, s]$	1,1 1,10	49 481	10 112	$\mathtt{P}_{\geqslant 0.9999}[\mathtt{F}\ done]$ $\mathtt{P}_{\geqslant 0.999}[\mathtt{F}\ done]$
cloud $[vm]$	5	8,841	2,177	$\mathtt{P}_{\geqslant 0.9999}[\mathtt{F}\ deployed]$
investor $[vinit, vmax]$	5,10	10,868	3,344	$\mathtt{R}^{profit}_{\geqslant 4.98}[\mathtt{C}]$
team-form $[N]$	3	12,476	2,023	$\mathtt{P}_{\geqslant 0.9999}[\mathtt{F}\ done_1]$
cdmsn $[N]$	3	1,240	604	$\mathtt{P}_{\geqslant 0.9999}[\mathtt{F}\ prefer_1]$

Table 6.3: Details of models and properties for approximating optimal randomised multi-strategies.

Name [parameters]	Parameters values	States	Controller states	States		
				M=100	M=200	M=300
android_3 $[r, s]$	1,1 1,10	49 481	10 112	90 1,629	94 1,741	98 1,853
cloud $[vm]$	5	8,841	2,177	29,447	32,686	35,233
investor $[vinit, vmax]$	5,10	10,868	3,344	33,440	35,948	38,456
team-form $[N]$	3	12,476	2,023	31,928	33,716	35,504
cdmsn $[N]$	3	1,240	604	3,625	3,890	4,155

Table 6.4: State space growth for approximating optimal randomised multi-strategies.

Name [parameters]	Parameters values	Pen. (det.)	Penalty (randomised)		
			M=100	M=200	M=300
android_3 $[r, s]$	1,1	1.01	0.91	0.905	0.903
	1,10	19.13	12.27*	9.12*	17.18*
cloud $[vm]$	5	1	0.91*	0.905*	0.91*
investor $[vinit, vmax]$	5,10	1	1*	1*	1*
team-form $[N]$	3	264	263.96*	263.95*	263.95*
cdmsn $[N]$	3	2	0.38*	1.9*	0.5*

* Sound but possibly non-optimal multi-strategy obtained after 5 minute MILP time-out.

Table 6.5: Experimental results for deterministic and approximated optimal randomised multi-strategies.

strategy, we used a feature of MILP solvers that allows the computation to be interrupted at any point in time. In many cases, we found that the solver was able to come up with a sound multi-strategy quickly but spent a prohibitively long time trying to prove that it is the optimal multi-strategy.

As would be expected, in most cases we observe smaller penalties with increasing values of M. One example where this is not true is the *cdmsn* case study, for which we obtain the smallest penalty for $M = 100$. We attribute this behaviour to the size of the MILP problem, which grows with M. The largest relative difference between the penalty of deterministic and randomised multi-strategy has been obtained for the *cdmsn* case study. The optimal deterministic strategy has a penalty of 2, while the randomised strategy has the penalty of at most 0.38. A natural question to ask is whether the lower penalty compensates for the increased computation time. We believe this question is likely to the depend on the domain where the multi-strategy will be used.

6.5 Summary

We have presented a framework for permissive controller synthesis for stochastic two-player games, based on generation of multi-strategies that guarantee a specified objective and are optimally permissive with respect to a penalty function. We proved that the problem is NP-hard for every type of multi-strategy and penalty structure. We established that randomised multi-strategies are more expressive than deterministic ones and propose an approximation scheme where deterministic strategies can be used to emulate

the randomised one.

Since the key aim of this thesis is to develop methods that are useful in practice, we proposed an MILP encoding for every type of multi-strategy and penalty structure. To make the MILP encoding work on non-trivial examples, we implemented optimisations that in some cases significantly reduced the synthesis time. We showed that bounds on variable values can reduce the search space of the problem and enable the MILP solver to find the solution more quickly. As a result of our experiments with various PRISM case studies, we realised that not all actions allowed by the model are relevant. This idea led to a second optimisation, where actions with zero penalty assigned to them could be dealt with more efficiently.

We developed a full implementation of the proposed approach as an extension of PRISM-games and used two MILP solvers to better understand the performance characteristics of the proposed methods. While MILP solvers can produce an optimal solution of the analysed constraints, we noticed that in many cases the interim solution produced by a solver can still yield a useful multi-strategy. In addition to analysing the performance characteristics of our encoding, we studied how multi-strategies can be used in practice. In Chapter 8, we present how multi-strategies can improve the robustness of an open-source stock monitoring application.

Chapter 7

Learning-based Controller Synthesis

7.1 Introduction

The efficiency and scalability of controller synthesis are often limited by excessive time or space requirements, caused by the need to store a full model in memory. This is especially true for the framework from Chapter 4, where we continuously analyse models at runtime. It is often the case that those models are already large to begin with, and tend to grow over time. For example, a cloud infrastructure may serve more and more clients as time passes, or a robot may discover new terrain during its explorations.

In this chapter, we explore new opportunities offered by learning-based methods, as used in fields such as planning [99], reinforcement learning [115] or optimal control [12]. In particular, we focus on algorithms that explore a stochastic game by generating trajectories through it and, whilst doing so, produce increasingly precise approximations of desired properties. The approximate values, along with other information, are used as heuristics to guide the model exploration and to minimise the solution time, as well as the portion of the model that needs to be explored.

We propose a general framework for applying such methods to the verification of stochastic games. Our algorithm is based on *real-time dynamic programming* (RTDP) [9] (see Section 3.8), which has been proposed for a subclass of MDPs. We extend it for arbitrary MDPs and a subclass of stochastic games. In its basic form, RTDP generates trajectories through the model and computes approximations of the property value in the form of lower bounds. While this may suffice in some scenarios (e.g. planning), in the context of verification we typically require more precise guarantees. Therefore, we also consider bounded RTDP (BRTDP) [101], which supplements RTDP with an additional upper bound. We extend BRTDP in a similar way as RTDP and add support for a larger class of probabilistic models. We have implemented our framework within PRISM-games [31] and

demonstrate considerable speed-ups over the fastest methods in PRISM-games. Because PRISM-games is an extension of PRISM [88], for MDPs both tools use the same solution methods. This means our algorithms are also able to outperform the fastest methods available in PRISM.

This chapter is divided into several sections. We start by outlining the general framework for models without end components. We then extend the framework to stochastic games that contain end components whose all states belong to a particular player, and to arbitrary MDPs. In the experimental section, we describe three different heuristics that are used to guide the generation of trajectories through the model. Next, we compare our methods against PRISM-games using a number of case studies, and perform further experiments giving additional insight on the performance characteristics of the proposed algorithms.

7.2 Learning-based Algorithms for Stochastic Games

In this chapter, we study a class of algorithms based on machine learning that stochastically approximate the value function of a stochastic game. We focus on probabilistic reachability, leaving the total expected reward as future work. We fix $\mathsf{G} = \langle S_\Diamond, S_\Box, S, \overline{s}, A, \delta, \mathcal{L}\rangle$ and denote by $V : S \times A \to [0,1]$ the *value function* for state-action pairs of G, defined for all (s,a) where $s \in S$ and $a \in A(s)$ by:

$$V(s,a) := \sum\nolimits_{s' \in S} \delta(s,a)(s') \cdot V(s').$$

Intuitively, $V(s,a)$ is the value of the probabilistic reachability property in s assuming that the first action performed is a. In this chapter, we are interested in properties that require computing a maximising strategy for the controller player. Techniques for minimising the property value follow similarly. To make the definition of $V(s,a)$ complete, we define $V(s)$:

$$V(s) \;\; := \;\; \begin{cases} \max\limits_{a \in A(s)} \sum\limits_{s' \in S} \delta(s,a)(s') \cdot V(s,a) & \text{if } s \in S_\Diamond \\ \min\limits_{a \in A(s)} \sum\limits_{s' \in S} \delta(s,a)(s') \cdot V(s,a) & \text{if } s \in S_\Box. \end{cases}$$

A *learning algorithm* $\mathcal{A}$ takes as input a stochastic game G, set of target states T and precision ε. The algorithm produces an approximation of the probabilistic reachability value accompanied by a strategy achieving the value. This is done by running a possibly large number of simulated executions of G, and iteratively updating upper and lower approximations $U : S \times A \to [0,1]$ and $L : S \times A \to [0,1]$, respectively, of the true value function $V : S \times A \to [0,1]$.

U and L are initialised to 1 and 0 so that $L(s,a) \leqslant V(s,a) \leqslant U(s,a)$ for all $s \in S$ and $a \in A$. During the computation of $\mathcal{A}$, simulated executions start in the initial state $\overline{s}$ and move from state to state according to choices made by the algorithm. The values of $U(s,a)$ and $L(s,a)$ are updated for the states s visited by the simulated execution.

The learning algorithm $\mathcal{A}$ terminates when $\max_{a \in A(s)} U(\overline{s},a) - \max_{a \in A(s)} L(\overline{s},a) < \varepsilon$ for $\overline{s} \in S_\Diamond$ and $\min_{a \in A(s)} U(\overline{s},a) - \min_{a \in A(s)} L(\overline{s},a) < \varepsilon$ for $\overline{s} \in S_\Box$. Since $U(s,a)$ and $L(s,a)$ are updated with new values based on the simulations only for states that are visited, the computation of the learning algorithm may be randomised.

Definition 10. *Denote by $\mathcal{A}(\varepsilon)$ the instance of a learning algorithm $\mathcal{A}$ with precision ε. We say that $\mathcal{A}$* converges *if, for every $\varepsilon > 0$, the computation of $\mathcal{A}(\varepsilon)$ terminates with $L(\overline{s},a) \leqslant V(\overline{s},a) \leqslant U(\overline{s},a)$.*

The function U defines a memoryless strategy σ_U which in every state $s \in S_\Diamond$ chooses all actions a maximising the value $U(s,a)$ over $A(s)$. If there exist several actions achieving the same maximal value, we pick one uniformly at random. As the strategy σ_U maximises over all actions, it is the best strategy that we can choose and we use it as an output of the algorithm.

7.3 Stochastic Games without End Components

We first present an algorithm for stochastic games *without* end components (ECs), which considerably simplifies the adaptation of BRTDP for verification purposes. End components are parts of the model in which generated trajectories can get stuck, preventing termination of the algorithm. Later, in Section 7.4, we extend our methods to a subclass of stochastic games and to arbitrary MDPs. Formally, we assume the following.

Assumption-EC-free. Stochastic game G has no ECs, except for two trivial ones containing distinguished terminal states **1** and **0**, respectively. We will assume that all target states are replaced by the **1** state and all non-target terminal states are replaced by **0**. This is to improve the presentation of the algorithm. Subsequently, we set $U(\mathbf{1}) = L(\mathbf{1}) = V(\mathbf{1}) = 1$ and $U(\mathbf{0}) = L(\mathbf{0}) = V(\mathbf{0}) = 0$. Because all targets states have been replaced by **1** states, we will also assume that $\mathcal{L}(s) = \emptyset$ for every $s \in S$.

7.3.1 Unbounded Reachability with BRTDP

The algorithm is presented as Algorithm 4, and works as follows. Recall that functions U and L store the current upper and lower bounds on the value function V, respectively. Each iteration of the outer loop is divided into two phases: Explore and Update. In

Algorithm 4 Learning algorithm (for stochastic game with no ECs)

1: **Inputs:** An EC-free stochastic game G
2: $U(\cdot,\cdot) \leftarrow 1, L(\cdot,\cdot) \leftarrow 0$
3: $L(\mathbf{1},\cdot) \leftarrow 1, U(\mathbf{0},\cdot) \leftarrow 0$
4: **repeat**
5: $\quad \omega \leftarrow \overline{s}$ /* Explore phase */
6: $\quad$ **repeat**
7: $\quad\quad$ **if** $s \in S_\Diamond$ **then**
8: $\quad\quad\quad a \leftarrow$ sampled uniformly from $\arg\max_{a \in A(last(\omega))} U(last(\omega), a)$
9: $\quad\quad$ **else**
10: $\quad\quad\quad a \leftarrow$ sampled uniformly from $\arg\min_{a \in A(last(\omega))} L(last(\omega), a)$
11: $\quad\quad$ **end if**
12: $\quad\quad s \leftarrow \textsc{GetSucc}(\omega, a)$
13: $\quad\quad \omega \leftarrow \omega\ a\ s$
14: $\quad$ **until** $s \in \{\mathbf{1}, \mathbf{0}\}$
15: $\quad$ **repeat** /* Update phase */
16: $\quad\quad pop(\omega)$
17: $\quad\quad a \leftarrow pop(\omega)$
18: $\quad\quad s \leftarrow last(\omega)$
19: $\quad\quad U(s,a) := \sum_{s' \in S} \delta(s,a)(s') U(s')$
20: $\quad\quad L(s,a) := \sum_{s' \in S} \delta(s,a)(s') L(s')$
21: $\quad$ **until** $\omega = \overline{s}$
22: **until** $U(s) - L(s) < \varepsilon$

the Explore phase (lines 5 - 14), the algorithm samples a finite path ω in G from $\overline{s}$ to a state in $\{\mathbf{1}, \mathbf{0}\}$ by always randomly choosing one of the enabled actions that maximises the U value, and sampling the successor state using the probabilistic transition function. In the Update phase (lines 15 - 21), the algorithm updates U and L for the state-action pairs along the path in a backward manner. Here, the function *pop* pops and returns the last element of the given sequence.

New values of $U(s,a)$ and $L(s,a)$ are computed by taking the weighted average of the corresponding U and L values, respectively, over all successors of s via action a. Formally, denote $U(s) = \max_{a \in A(s)} U(s,a)$ and $L(s) = \max_{a \in A(s)} L(s,a)$ when $s \in S_\Diamond$, and in the case of $s \in S_\Box$ we minimise over both $U(s,a)$ and $L(s,a)$ values.

Note that, in the Explore phase, an action maximising the value of U is chosen and the successor is sampled based on the output of the GetSucc function. In the most basic scenario, GetSucc returns a successor according to the probabilistic transition function of G. However, we can consider various modifications. Successors may be chosen in different ways, for instance, uniformly at random, in a round-robin fashion, or assigning various probabilities (bounded from below by some fixed $p > 0$) to all possibilities in any biased way. In order to guarantee convergence, some conditions have to be satisfied. Intuitively,

we require that the state-action pairs used by ε-optimal strategies have to be chosen a sufficient number of times. If this condition is satisfied then the convergence is preserved and the practical running times may significantly improve. For details, see Section 7.5.

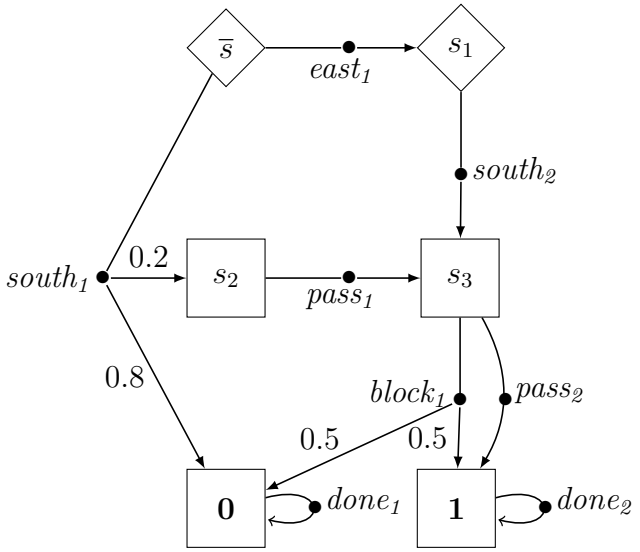


Figure 7.1: Stochastic game G for Example 27.

Theorem 12. *The algorithm BRTDP converges almost surely under Assumption-EC-free.*

The proof for Theorem 12 is divided into two parts. We first prove that for every iteration of Algorithm 4 it holds that $L(s,a) \leqslant V(s,a) \leqslant U(s,a)$ for any state and action. We then prove that the gap between the the upper and lower bound in the initial state, i.e. $U(\overline{s}) - L(\overline{s})$, eventually goes to zero. As a corollary of both of these claims, we obtain that the algorithm terminates and converges to the correct value. The full proof can be found in Appendix C.

Example 27 To show an example run of Algorithm 4, we first need a stochastic game that satisfies Assumption-EC-free. In Figure 7.1, we show a game G that satisfies the assumption. It models a robot that is under our control and moves between five locations, with a goal to reach the **1** state. We start in the $\overline{s}$ state and we can either move in the $east_1$ or $south_1$ direction. When moving in the $east_1$ direction our movement can be blocked by the second robot, but with probability 0.5 we still reach the goal state. In the case of moving in the $south_1$ direction, we cannot be blocked by the second robot but with high probability we go to the sink state **0**. To express our property we use the probabilistic reachability property $\phi = \mathsf{P}_{\geqslant 0.5}[\mathtt{F}\ \mathbf{1}]$ and the strategy satisfying the property would pick $east_1$ in $\overline{s}$.

Consider a run of Algorithm 4 given game G as input and $\varepsilon = 10^{-8}$. Assume that, the first path that we get from the EXPLORE phase is $\omega = \overline{s}south_1\mathbf{0}$. In the following

UPDATE phase, we compute: $U(\overline{s}, south_1) = 0.2, L(\overline{s}, south_1) = 0$, and the algorithm cannot terminate because at line 22 we have $U(\overline{s}) - L(\overline{s}) = 1$ due to the unexplored $east_1$ action. In the second iteration $\omega = \overline{s} east_1 s_1 south_2 s_3 block_1 \mathbf{0}$ and in the UPDATE phase we have $U(s_3, block_1) = L(s_3, block_1) = 0.5$, $U(s_1, south_2) = 0.5, L(s_1, south_2) = 0$ and $U(\overline{s}, east_1) = 0.5, L(\overline{s}, east_1) = 0$. Again, we cannot terminate because $U(\overline{s}) - L(\overline{s}) = 0.5$. In the next iteration $\omega = \overline{s} east_1 s_1 south_2 s_3 pass_2 \mathbf{0}$, and in the UPDATE phase we compute $U(s_3, pass_2) = L(s_3, pass_2) = 1$, $U(s_1, south_2) = L(s_1, south_2) = 0.5$ and $U(\overline{s}, east_1) = L(\overline{s}, east_1) = 0.5$. Subsequently, in line 22 we get $U(\overline{s}) - L(\overline{s}) = 0$, which terminates the algorithm.

The resulting strategy picks $east_1$ in $\overline{s}$ and achieves the value of 0.5 which satisfies the property ϕ. The algorithm was able to terminate without ever visiting state s_2. This was possible because, even in the best scenario, i.e. $U(s_2) = L(s_2) = 1$, the $south_1$ action in $\overline{s}$ would only achieve $U(\overline{s}, south_1) = L(\overline{s}, south_1) = 0.2$, whereas $U(\overline{s}, east_1) = L(\overline{s}, east_1) = 0.5$.

7.3.2 Step-bounded Reachability with BRTDP

Algorithm 4 can be trivially adapted to handle bounded reachability properties by preprocessing the input stochastic game. Every state is equipped with a bounded counter with values ranging from 0 to b where b is the step bound, the current value denoting the number of steps taken so far. All target states remain as target states for all counter values, and every non-target state with counter value b becomes rejecting. Then, to determine the b-step reachability in the original stochastic game one can compute the (unbounded) reachability in the new stochastic game. Although this means that the number of states is multiplied by $b + 1$, in practice the size of the explored part of the model can be small.

7.4 Stochastic Games with one-player ECs

We first illustrate with an example that the algorithm BRTDP as presented in Section 7.3 may not converge when there are ECs in the stochastic game.

Example 28 Consider the game G from Figure 7.2 with an EC $(\{\overline{s}, s_1\}, \{east_1, west_1\})$. We re-use the property from the previous example, namely $\phi = \mathtt{P}_{\geqslant 0.5}[\mathtt{F}\ \mathbf{1}]$ and a strategy satisfying the property would pick $east_1$ in $\overline{s}$ and $south_1$ in s_1.

We run Algorithm 4 with $\varepsilon = 10^{-8}$ as in Example 27 and assume that the first two explored paths are $\omega = \overline{s} east_1 s_1 south_1 s_2 block_1 \mathbf{0}$ and $\omega = \overline{s} east_1 s_1 south_1 s_2 pass_1 \mathbf{1}$. In the UPDATE phase we compute $U(s_1, south_1) = L(s_1, south_1) = 0.5$ and run the third iteration

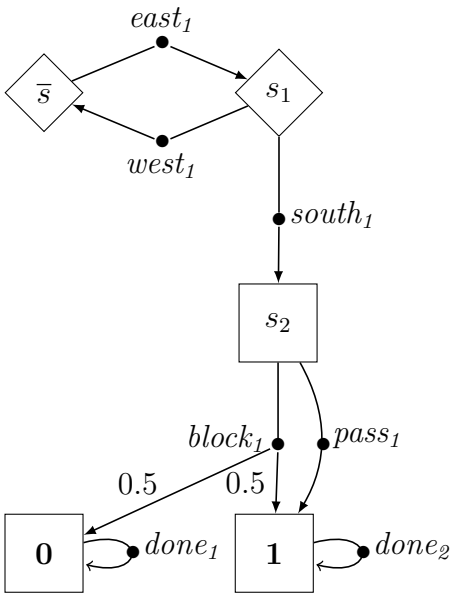


Figure 7.2: Stochastic game G with an EC.

of the algorithm. Because $U(s_1, west_1) = 1$ and $U(\overline{s}, east_1) = 1$, in line 8 the algorithm is going to greedily choose those actions, causing an infinite loop in the EXPLORE part.

In general, any state in an EC has an upper bound of 1 since, by definition, there are actions that guarantee the next state is in the EC, i.e., is a state with upper bound 1. This argument holds even for standard value iteration with values initialised to 1.

One way of dealing with this problem is to identify all MECs [28, 29] and "collapse" them into single states. This can be safely done for MDPs and for stochastic games where all ECs belong to a subclass that we call *one-player ECs* . Supporting arbitrary stochastic games remains an open problem which we plan to address in future work.

Definition 11 (One-player EC)**.** *An end component* (R, B) *of a stochastic game* G *is a* one-player EC *iff* $player(R) \in \{\Diamond, \Box\}$.

The stochastic game from Figure 7.2 contains a single one-player EC consisting of two states and two actions: $(\{\overline{s}, s_1\}, \{east_1, west_1\})$. Please note that in the case of an MDP all ECs are one-player ECs as the $player()$ function returns $\Diamond$ for every state of the MDP.

7.4.1 Identification and Processing of *one-player ECs*

Typically, algorithms for identification and collapsing of ECs require that the full underlying graph of the model is known, which proves difficult for large models. Hence, we propose a method that allows us to deal with ECs "on the fly". We first describe the collapsing of a set of states and then present how to identify and process ECs.

Collapsing states

In the following, we say that a stochastic game $\mathsf{G}' = \langle S_\Diamond', S_\Box', S', \overline{s}', A', \delta', \mathcal{L}' \rangle$ is obtained from $\mathsf{G} = \langle S_\Diamond, S_\Box, S, \overline{s}, A, \delta, \mathcal{L} \rangle$ by *collapsing* a tuple (R, B) into a single state $s_{(R,B)}$, where $R \subseteq S$ and $B \subseteq A$ with $B \subseteq \bigcup_{s \in R} A(s)$ if:

- $S' = (S \setminus R) \cup \{s_{(R,B)}\}$
- $\overline{s}'$ is either $s_{(R,B)}$ or $\overline{s}$, depending on whether $\overline{s} \in R$ or not
- $A' = A \setminus B$
- δ' is defined for all $s \in S'$ and $a \in A'$ by:
  - $\delta'(s,a)(s') = \delta(s,a)(s')$ for $s, s' \neq s_{(R,B)}$
  - $\delta'(s,a)(s_{(R,B)}) = \sum_{s' \in R} \delta(s,a)(s')$ for $s \neq s_{(R,B)}$
  - $\delta'(s_{(R,B)},a)(s') = \delta(s,a)(s')$ for $s' \neq s_{(R,B)}$ and s the state with $a \in A(s)$
  - $\delta'(s_{(R,B)},a)(s_{(R,B)}) = \sum_{s' \in R} \delta(s,a)(s')$ where s is the state with $a \in A(s)$
- $\mathcal{L}'(s) = \mathcal{L}(s)$ for $s \in S'$ and $\mathcal{L}'(s_{(R,B)}) = \emptyset$.

We denote the above transformation, which creates G' from G, as the COLLAPSE function, i.e., COLLAPSE(R, B). As a special case, given a state s and a terminal state $s' \in \{\mathbf{1}, \mathbf{0}\}$, we use MAKETERMINAL$(s, s')$ as a shorthand for COLLAPSE$(\{s, s'\}, A(s))$, where the new state is renamed to s'. Intuitively, after MAKETERMINAL(s, s'), every transition previously leading to state s will now lead to the terminal state s'.

For practical purposes, it is important to note that the collapsing does not need to be implemented explicitly, but can be done by keeping a separate data structure which stores information about the collapsed states.

Identification of one-player ECs

We identify ECs "on-the-fly" through simulations that get stuck in them. In Algorithm 4, the variable ω contains the currently explored path. Let (N, G) be the set of states and actions explored in ω. We use $N_\Diamond$ to refer to visited states that belong to the controller player and $N_\Box$ for the environment player states. To obtain the EC from the set N of explored states, we use Algorithm 5.

This computes an auxiliary stochastic game $\mathsf{G}^N = \langle N_\Diamond', N_\Box', N', \overline{s}, A', \delta', \mathcal{L}' \rangle$ that includes all visited states as well as their immediate successors. The game is defined as follows:

Algorithm 5 Identification of one-player ECs for BRTDP

```
1: function IdentifyOnePlayerECs(G, N)
2:     compute G^N
3:     G' ← MECs of G^N
4:     G ← {(R, B) ∈ G' | R ⊆ N ∧ player(R) ∈ {◇, □}}
5:     return G
6: end function
```

- $N'_\Diamond = N_\Diamond \cup \{t \mid \exists s \in N, a \in A(s) \text{ such that } \delta(s,a)(t) > 0 \text{ and } t \in S_\Diamond\}$
- $N'_\Box = N_\Box \cup \{t \mid \exists s \in N, a \in A(s) \text{ such that } \delta(s,a)(t) > 0 \text{ and } t \in S_\Box\}$
- $N' = N'_\Diamond \cup N'_\Box$
- $A' = \bigcup_{s \in N} A(s) \cup \{\bot\}$
- $\delta'(s,a) = \delta(s,a)$ if $s \in N$, and $\delta'(s,\bot)(s) = 1$ otherwise
- $\mathcal{L}'(s) = \mathcal{L}(s)$ for $s \in N'$.

The algorithm then computes all MECs of G^N that are contained in N and identifies those that are one-player ECs. The following lemma states that each of these is indeed one-player EC in the original stochastic game.

Lemma 3. *Let G, G^N be the stochastic games from the construction above and N be the set of explored states. Then every one-player MEC (R, B) in G^N such that $R \subseteq N$ is a one-player EC in G.*

Proof. Let $\mathsf{G} = \langle S_\Diamond, S_\Box, S, \overline{s}, A, \delta, \mathcal{L}\rangle$ and $\mathsf{G}^N = \langle N'_\Diamond, N'_\Box, N', \overline{s}, A', \delta', \mathcal{L}'\rangle$ be the two stochastic games, and let (R, B) be a MEC in G^N such that $R \subseteq N$. As $N \subseteq S$, we have that the states of R are present in the stochastic game G and from the definition of A' we have that $B \subseteq \bigcup_{s \in R} A(s)$. The two required properties of an EC:

1. if $\delta(s,a)(s') > 0$ for some $s \in R$ and $a \in B$, then $s' \in R$
2. for all $s, s' \in R$, there exists a path $\omega = \overline{s} a_0 s_1 a_1 \dots s_n$ such that $\overline{s} = s$, $s_n = s'$, and for all $0 \leqslant i < n$ we have that $a_i \in B$ and $\delta(s_i, a)(s_{i+1}) > 0$

follow easily from the fact that for all states $s \in R$ and actions $a \in B$ we have: $A(s) = A'(s)$; $\delta(s,a) = \delta'(s,a)$; and (R, B) is an EC in G^N. □

Processing of one-player ECs

After identifying all one-player ECs in the currently explored model, we can start processing them. Processing of one-player ECs can be found in Algorithm 6 and is divided into three steps. Firstly, in line 4 each one-player EC is collapsed into a single state. Next, the processing phase assigns the old values of actions to the new states $s_{(R,B)}$ created during the collapsing process; this happens in lines 6 and 7. Lastly, lines 9-21 of the processing algorithm contain optimisations when collapsing ECs that have certain structure. The correctness of the processing phase is established in Lemma 4.

Algorithm 6 Identification and processing of one-player ECs

```
1: function On-the-fly-EC(G, N)
2:     G ← IdentifyOnePlayerECs(G, N)
3:     for all (R, B) ∈ G do    /* Processing phase */
4:         Collapse(R, B)
5:         for all s ∈ R and a ∈ A(s) \ B do
6:             U(s_(R,B), a) ← U(s, a)
7:             L(s_(R,B), a) ← L(s, a)
8:         end for
9:         if player(R) = ◇ then
10:            if R ∩ T ≠ ∅ then
11:                MakeTerminal(s_(R,B), 1)
12:            else if no actions enabled in s_(R,B) then
13:                MakeTerminal(s_(R,B), 0)
14:            end if
15:        else if player(R) = □ then
16:            if R ∩ T ≠ ∅ and no actions enabled in s_(R,B) then
17:                MakeTerminal(s_(R,B), 1)
18:            else if R ∩ T = ∅ then
19:                MakeTerminal(s_(R,B), 0)
20:            end if
21:        end if
22:    end for
23: end function
```

Lemma 4. *Assume (R, B) is a one-player EC in a stochastic game G, V_{G} the value before* Processing *phase in Algorithm 6, and $V_{\mathsf{G}'}$ the value after (R, B) has been processed, then:*

1. *for $i \in \{\mathbf{1}, \mathbf{0}\}$, if* MakeTerminal$(s_{(R,B)}, i)$ *is called, then $\forall s \in R : \ V_{\mathsf{G}}(s) = i$*
2. *$\forall s \in S \setminus R : \ V_{\mathsf{G}}(s) = V_{\mathsf{G}'}(s)$*
3. *$\forall s \in R : \ V_{\mathsf{G}}(s) = V_{\mathsf{G}'}(s_{(R,B)})$.*

The proof for Lemma 4 can be found in Appendix C. For point (1), we prove each case when MAKETERMINAL() function is called. Points (2) and (3) are proven using techniques similar to [36].

7.4.2 BRTDP and *one-player ECs*

To obtain BRTDP that works with stochastic games containing one-player ECs, we modified Algorithm 4. The changes can be seen in Algorithm 7. In line 17, we insert a check such that, if the length of the path ω explored (i.e., the number of states) exceeds k_i (see below), then we invoke the ON-THE-FLY-EC function with a set of visited states N as an argument. The ON-THE-FLY-EC function possibly modifies the stochastic game by processing (collapsing) some ECs as described in Algorithm 6. After the ON-THE-FLY-EC function terminates, we interrupt the current EXPLORE phase, and jump to the beginning of the EXPLORE phase for the i+1-th iteration (i.e., generating a new path again, starting from $\overline{s}$ in the modified stochastic game). To complete the algorithm description we describe the choice of k_i.

Choice of k_i. Because computing one-player ECs can be expensive, we do not call ON-THE-FLY-EC every time a new state is explored, but only after every k_i steps of the repeat-until loop at lines 8–21 in iteration i. The specific value of k_i can be decided experimentally and changed as the computation progresses. In our experiments we use $k_i = 10^{5+\lfloor i/10 \rfloor}$. This guarantees that ON-THE-FLY-EC is executed early for models that converge quickly, and, for models that take a longer time to converge, they have time for the exploration and are not constantly running the ON-THE-FLY-EC function.

Finally, we establish that the modified algorithm, which we refer to as *on-the-fly BRTDP*, converges.

Theorem 13. *On-the-fly BRTDP converges almost surely for stochastic games with one-player ECs and for all MDPs.*

The proof is in Appendix C. An important observation that we make in the proof of Theorem 13 is that, while we start with a stochastic game containing one-player ECs after finitely many calls to ON-THE-FLY-EC we obtain a stochastic game with all end components collapsed. For such a game, we can use similar ideas to those in the proof for Theorem 12.

Example 29 Let us describe the execution of on-the-fly BRTDP on the stochastic game G from Figure 7.3 (left). Choose $k_i \geqslant 6$ for all i. The two iterations of the EXPLORE loop at lines 8 to 21 of Algorithm 7 generate paths ω and ω' that contain some (possibly zero) number of loops $\overline{s}\,east_1\, s_1\, west_1$ followed by $s_1\, south_1\, s_2\, block_1\, \mathbf{1}$ or $s_1\, south_1\, s_2\, pass_1\, \mathbf{0}$. In

Algorithm 7 Learning algorithm (for stochastic game with one-player ECs)

Inputs: Stochastic game G with only one-player ECs
$U(\cdot,\cdot) \leftarrow 1, L(\cdot,\cdot) \leftarrow 0$
$L(\mathbf{1},\cdot) \leftarrow 1, U(\mathbf{0},\cdot) \leftarrow 0$
$N \leftarrow \emptyset$, $i \leftarrow 0$
repeat
  `explore:`
  $\omega \leftarrow \overline{s}$ /* EXPLORE phase */
  repeat
    if $s \in S_\Diamond$ **then**
      $a \leftarrow$ sampled uniformly from $\arg\max_{a \in A(last(\omega))} U(last(\omega), a)$
    else
      $a \leftarrow$ sampled uniformly from $\arg\min_{a \in A(last(\omega))} L(last(\omega), a)$
    end if
    $s \leftarrow \text{GETSUCC}(\omega, a)$
    $\omega \leftarrow \omega\ a\ s$
    $N \leftarrow N \cup s$
    if $|\omega| \geqslant k_i$ **then**
      ON-THE-FLY-EC(G, N)
      `go to explore`
    end if
  until $s \in \{\mathbf{1}, \mathbf{0}\}$
  repeat /* UPDATE phase */
    $pop(\omega)$
    $a \leftarrow pop(\omega)$
    $s \leftarrow last(\omega)$
    $U(s,a) := \sum_{s' \in S} \delta(s,a)(s')U(s')$
    $L(s,a) := \sum_{s' \in S} \delta(s,a)(s')L(s')$
  until $\omega = \overline{s}$
  $i \leftarrow i + 1$
until $U(s) - L(s) < \varepsilon$

the subsequent UPDATE phase, we set $U(s_2, block_1) = L(s_2, block_1) = 0.5$, $U(s_2, pass_1) = L(s_2, pass_1) = 1$ and $U(s_2) = L(s_2) = 0.5$. In the third iteration of the EXPLORE phase, the path $\omega'' = \overline{s}\, east_1\, s_1\, west_1\, \overline{s}\, east_1\, s_1\, west_1 \ldots$ is generated, and the newly inserted check for ON-THE-FLY-EC will be triggered once ω'' achieves the length k_i.

The algorithm now aims to identify one-player ECs in the stochastic game based on the part of the stochastic game explored so far. To do so, the stochastic game G^N for the set $N = \{\overline{s}, s_1, s_2, \mathbf{0}, \mathbf{1}\}$ is constructed. We then run one-player EC detection on G^N, finding that $(\{\overline{s}, s_1\}, \{east_1, west_1\})$ is a one-player EC and so it gets collapsed according to the COLLAPSE procedure. This gives the stochastic game G' from Figure 7.3 (right).

The execution then continues with G'. A new path is generated in the EXPLORE phase

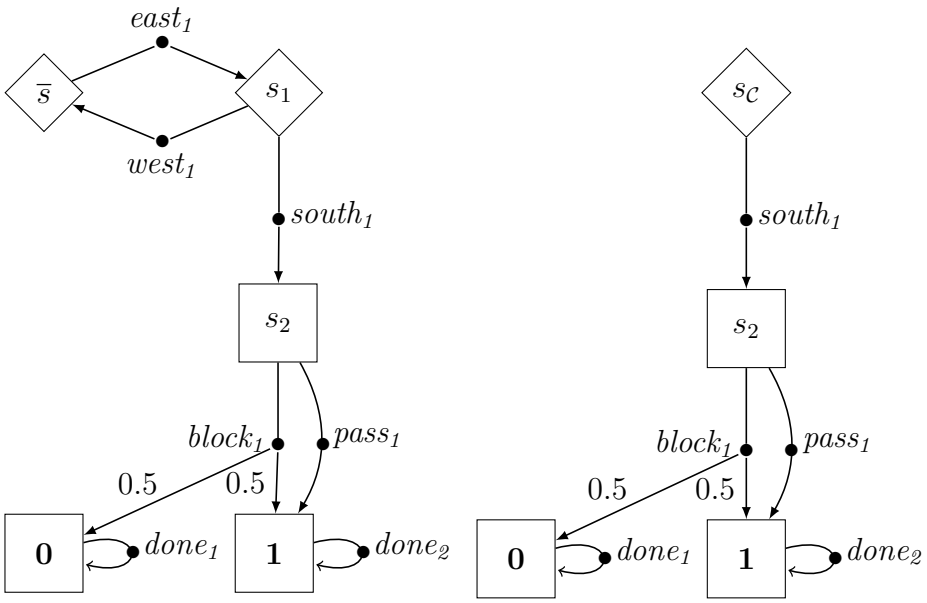


Figure 7.3: Executing COLLAPSE function on a stochastic game G.

of the Algorithm 7; suppose it is $\omega''' = s_{\mathcal{C}} south_1 s_2 pass_1 t$ where $t \in \{\mathbf{1}, \mathbf{0}\}$. In the UPDATE phase we then update the value $U(s_{\mathcal{C}}, south_1) = L(s_{\mathcal{C}}, south_1) = 0.5$, which makes the condition at the last line of Algorithm 7 satisfied, and the algorithm terminates, having computed the correct value.

7.5 Experimental Results

We developed an implementation of our learning-based framework as an extension of PRISM-games [31], building upon its simulation engine for generating trajectories.

We implement Algorithm 7, with the optimisation of taking N as the set of all states explored so far. We consider three distinct variants of the learning algorithm by modifying the GETSUCC function in Algorithm 7, which is the heuristic responsible for picking a successor state s' after choosing some action a in each state s of a trajectory. The first variant takes the unmodified GETSUCC, selecting s' at random according to the distribution $\delta(s, a)$. This behaviour follows that of the original RTDP algorithm [9]. The second uses the heuristic proposed for BRTDP in [101], selecting the successor $s' \in supp(\delta(s, a))$ that maximises the difference $U(s') - L(s')$ between bounds for those states (referred to as M-D). For the third, we propose an alternative approach that systematically chooses all successors s' in a round-robin (referred to as R-R) fashion, and guarantees sure termination.

We evaluated our implementation on six existing benchmark models, running the experiments on a machine with a 2.8GHz Xeon processor and 32GB of RAM, running Fe-

dora 14, and using a 30 minute timeout for each verification. We first consider four MDP models: three models from the PRISM benchmark suite [89], *zeroconf*, *wlan*, and *firewire_impl_dl*, and a fourth one from [54], *mer*. The stochastic game examples include *android_3* (see Chapter 8) and *mdsm* [30]. The first three MDPs and the first stochastic game example use (unbounded) probabilistic reachability properties; the fourth MDP example and the last stochastic game example uses a step-bounded probabilistic reachability property. The latter is employed to show differences between heuristics that were less visible in the unbounded case. Descriptions of the case studies used can be found in Appendix A and the models can be found at [128].

We run BRTDP from Algorithm 7 and compare its performance to PRISM-games. Please note that, for MDPs, PRISM-games and PRISM share the same solution methods so our results also outperform PRISM. We terminate PRISM-games when the bounds L and U differ by at most ε in the initial state. We use $\varepsilon = 10^{-6}$ in all cases except *zeroconf* and *android_3*, where $\varepsilon = 10^{-8}$ is used since the actual values are very small. For PRISM-games, in the case of MDPs, we use its fastest engine, which is the "sparse" engine, running value iteration. In the case of stochastic games, we also run value iteration but use the "explicit" engine as this was the only available engine at the time of writing this thesis.

Computation of our algorithm is terminated when the values for all states in successive iterations differ by at most ε. Strictly speaking, this is not guaranteed to produce an ε-optimal strategy (e.g. in the case of very slow numerical convergence), but on all these examples it does.

The experimental results are summarised in Tables 7.1 to 7.4. For each model, we give the name of the case study, the parameters used and the number of states in the full model. The summary of the properties used can be found in Table 7.1. Due to randomised behaviour of our algorithms the results have been averaged over 20 runs. We present a comparison of the running times of our algorithm against PRISM-games. In the case of PRISM-games, we report the total verification time, which includes model construction, precomputation of zero/one states and value iteration.

Performance and scalability

In Table 7.2, we see that, our method outperforms PRISM-games on all six benchmarks. On average, for the fastest heuristic, our learning-based algorithm works several orders of magnitude faster than PRISM-games. In the best case, PRISM-games did not finish within the 30 minute timeout for the *android* case study and we reported only 2.6 seconds for the R-R heuristic. In the worst case, the RTDP heuristic took 247.8 seconds to finish for the *android* case study. The other two heuristics for the same example terminated

Name [parameters]	Parameter values	States	Property
zeroconf $[N = 20, K]$	10 14 18	3,001,911 4,427,159 5,477,150	$\mathtt{P}_{min=?}[\,\mathtt{F}\ \mathit{configured}\,]$
wlan $[BOFF]$	4 5 6	345,000 1,295,218 5,007,548	$\mathtt{P}_{max=?}[\,\mathtt{F}\ \mathit{sent\_correctly}\,]$
firewire_impl_dl $[delay = 36,$ *deadline*$]$	220 240 260 280	10,490,495 13,366,666 15,255,584 19,213,802	$\mathtt{P}_{max=?}[\,\mathtt{F}\ \mathit{leader\_elected}\,]$
mer $[N, q]$	$3000, 0.0001$ $3000, 0.9999$ $4500, 0.0001$ $4500, 0.9999$	17,722,564 17,722,564 26,583,064 26,583,064	$\mathtt{P}_{max=?}[\,\mathtt{F}^{\leqslant 30}\ \mathit{no\_deadlock}\,]$
android_3 $[r, s = 10]$	15 20 25 30	364,033 832,168 1,589,953 2,707,138	$\mathtt{P}_{max=?}[\,\mathtt{F}\ \mathit{done}\,]$
mdsm $[k, N]$	$8, 6$ $8, 7$ $8, 8$ $10, 6$ $10, 7$ $10, 8$	2,384,369 2,384,369 2,384,369 6,241,312 6,241,312 6,241,312	$\mathtt{P}_{max=?}[\,\mathtt{F}^{\leqslant k}\mathit{first\_job\_arrived}\,]$

Table 7.1: Details of models and properties for running the BRTDP algorithm.

Name [parameters]	Parameter values	States	Time (s)			
			PRISM	RTDP	M-D	R-R
zeroconf $[N = 20, K]$	10	3,001,911	129.9	4.8	0.7	1.0
	14	4,427,159	218.2	8.9	1.2	1.3
	18	5,477,150	303.8	47.9	3.5	2.6
wlan $[BOFF]$	4	345,000	7.4	0.6	0.5	0.6
	5	1,295,218	22.3	0.6	0.5	0.5
	6	5,007,548	82.9	0.6	0.5	0.5
firewire_impl_dl $[delay = 36,$ $deadline]$	220	10,490,495	103.9	2.5	2.5	2.5
	240	13,366,666	145.4	13.9	14.7	18.1
	260	15,255,584	185.4	53.6	32.8	47.1
	280	19,213,802	245.4	12.7	10.3	10.7
mer $[N, q]$	$3000, 0.0001$	17,722,564	158.5	145.5	5.3	9.4
	$3000, 0.9999$	17,722,564	157.7	26.1	6.5	16.3
	$4500, 0.0001$	26,583,064	250.7	143.6	5.3	9.6
	$4500, 0.9999$	26,583,064	246.6	25.7	6.4	16.1
android_3 $[r, s = 10]$	15	364,033	108.6	236.5	1.4	13.9
	20	832,168	320.3	235.9	1.4	13.3
	25	1,589,953	741.4	234.6	1.4	13.0
	30	2,707,138	*	247.8	1.5	13.9
mdsm $[k, N]$	$8, 6$	2,384,369	68.8	3.5	0.7	0.8
	$8, 7$	2,384,369	73.9	6.3	0.9	1.2
	$8, 8$	2,384,369	69.9	10.1	1.3	1.7
	$10, 6$	6,241,312	245.4	41.7	1.2	2.1
	$10, 7$	6,241,312	251.5	84.9	1.9	3.3
	$10, 8$	6,241,312	259.2	146.2	3.1	5.4

* Algorithm did not terminate within 30 minute time-out.

Table 7.2: Verification times using BRTDP (three different heuristics) and PRISM.

within seconds.

The improvements in execution time on these benchmarks are possible due to multiple factors. The algorithm explores only a portion of the state space and a large number of suboptimal policies can be disregarded after only partial exploration. Moreover, in most cases, target states in our benchmarks are located close to the initial states, meaning that relatively short paths need to be explored in the EXPLORE part of the algorithm. Later in this section we will expand on each of the causes.

The RTDP heuristic is generally the slowest of the three, and tends to be sensitive to the probabilities in the model. In the *mer* example, changing the parameter q can mean that some states are no longer visited due to low probabilities on incoming transitions. This results in a considerable slowdown. This is a potential problem for models containing rare events, i.e., modelling failures that occur with very low probability. In most cases, the M-D and R-R heuristics perform very similarly despite being quite different (one is randomised, the other deterministic). Both perform consistently well on most examples.

An important feature of our method is the improved scalability of computation when compared to PRISM-games. The difference can be clearly seen in Table 7.2. In every case, we see that, with increasing model size, PRISM-games takes proportionally more time to compute the optimal strategy. A similar increase in model size has a much weaker effect on our algorithms. For example, for the *wlan* case study we report an almost 10-fold increase in PRISM-games computation time when increasing the model size from $345,000$ to $5,007,548$. For the same case study, our algorithms do not report almost any increase in the computation time. We believe this is caused by a certain structure of the model that our method can exploit. In the paragraph describing the size of the obtained strategies we will expand on this finding.

The number of distinct states visited by the algorithm can be found in Table 7.3. In every case the learning-based algorithm has to visit several orders of magnitude fewer states than PRISM-games. We attribute this to the fact our algorithms can quickly discard parts of the state space that are not crucial for convergence. This is only possible for models where the subset of states relevant to the convergence is small.

A surprising result has been obtained for the RTDP heuristic, which on average visits the smallest number of states. We observe that visiting fewer states than the other heuristics does not guarantee quicker termination time. This is caused by RTDP visiting only a subset of states that are reached with high probability. This subset is not necessarily the same as the subset relevant for convergence. One possible optimisation to the RTDP could be to modify its approach. After some initial time, the GETSUCC function could start picking the successors from the transitions with low probability.

In Table 7.4, we report the size of the optimal strategies that are obtained by running

Name [parameters]	Parameter values	States	Visited States		
			RTDP	M-D	R-R
zeroconf $[N = 20, K]$	10	3,001,911	723	2,089	2,554
	14	4,427,159	941	4,209	3,078
	18	5,477,150	1,335	5,886	3,848
wlan $[BOFF]$	4	345,000	1,613	1,024	1,230
	5	1,295,218	1,829	1,232	1,312
	6	5,007,548	1,663	1,210	1,279
firewire_impl_dl $[delay = 36,$ *deadline*$]$	220	10,490,495	21,125	23,487	21,098
	240	13,366,666	18,199	14,773	15,011
	260	15,255,584	19,314	15,317	14,906
	280	19,213,802	29,485	23,715	24,594
mer $[N, q]$	$3000, 0.0001$	17,722,564	2,152	3,080	3,641
	$3000, 0.9999$	17,722,564	3,294	4,826	4,807
	$4500, 0.0001$	26,583,064	2,146	3,084	3,642
	$4500, 0.9999$	26,583,064	3,286	4,830	4,804
android_3 $[r, s = 10]$	15	364,033	1,991	13,093	38,003
	20	832,168	1,991	13,389	38,350
	25	1,589,953	1,998	13,010	38,915
	30	2,707,138	1,997	13,403	39,669
mdsm $[k, N]$	$8, 6$	2,384,369	2,501	2,940	2,986
	$8, 7$	2,384,369	3,931	4,749	4,809
	$8, 8$	2,384,369	5,794	7,176	7,252
	$10, 6$	6,241,312	6,982	7,527	7,702
	$10, 7$	6,241,312	12,235	13,210	13,515
	$10, 8$	6,241,312	19,667	21,242	21,666

Table 7.3: Number of visited states using BRTDP (three different heuristics).

Name [parameters]	Parameter values	States	Strategy size		
			RTDP	M-D	R-R
zeroconf $[N = 20, K]$	10	3,001,911	271	1,307	1,665
	14	4,427,159	334	2,222	2,106
	18	5,477,150	465	2,209	1,470
wlan $[BOFF]$	4	345,000	68	68	85
	5	1,295,218	83	76	67
	6	5,007,548	67	65	86
firewire_impl_dl $[delay = 36,$ *deadline*$]$	220	10,490,495	340	443	341
	240	13,366,666	346	341	341
	260	15,255,584	341	342	341
	280	19,213,802	354	354	353
mer $[N, q]$	$3000, 0.0001$	17,722,564	43	43	42
	$3000, 0.9999$	17,722,564	63	61	62
	$4500, 0.0001$	26,583,064	42	43	42
	$4500, 0.9999$	26,583,064	62	62	61
android_3 $[r, s = 10]$	15	364,033	243	267	241
	20	832,168	250	259	235
	25	1,589,953	247	261	237
	30	2,707,138	239	267	241
mdsm $[k, N]$	$8, 6$	2,384,369	12	12	12
	$8, 7$	2,384,369	13	13	13
	$8, 8$	2,384,369	14	14	14
	$10, 6$	6,241,312	12	12	12
	$10, 7$	6,241,312	13	13	13
	$10, 8$	6,241,312	14	14	14

Table 7.4: Number of reachable states under ε-optimal strategy using BRTDP (three different heuristics).

the algorithm. The optimal strategies tend to be very small when compared to the size of the model. In most cases they contain from tens to thousands of states. The smallest strategies have been reported for the *mdsm* case study, while the largest for *zeroconf*. For every case study the size of the strategy stays fairly consistent despite the parameter change. A possible explanation is that the optimal strategy does not significantly change with increasing model size. We believe this behaviour is responsible for the scalability improvements we previously reported.

Limitations of our methods

While the results presented above are encouraging, they tend to depend on structural properties of the model. In particular, for models that require generating long trajectories we observed that our algorithms deteriorate quickly. In Figure 7.4, we show how the length

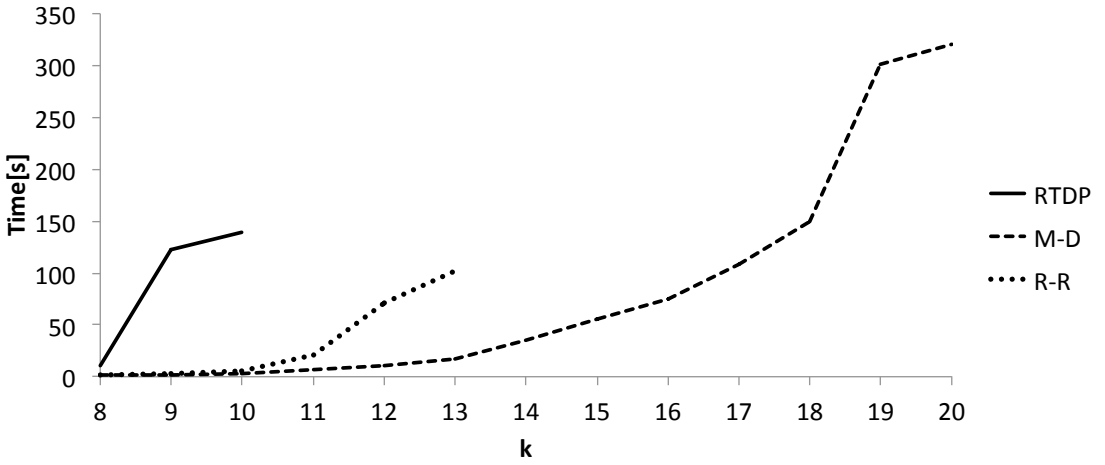


Figure 7.4: Influence of the length of the explored paths on the computation time

of the explored paths in the EXPLORE phase of the algorithm can have a significant impact on the runtime of the algorithm. We run the experiment with three heuristics using the *mdsm* example for $N = 8$. We vary the k parameter, which for a bounded property directly translates to the length of the explored paths. On the x-axis we report the value of the k parameter and on the y-axis the computation time.

We observe that increasing the length of the explored paths causes a steep increase in the runtime of the algorithm. For the RTDP heuristics, we are unable to solve models when exploring paths longer than 10 states. In the case of the M-D heuristic, we performed better, but our algorithm will not be able to solve, within reasonable time, models that explore paths longer than 20 states. One possible solution would be to run our path simulation in parallel. While using multiple threads should decrease the computation time, the steep increase that we observed may still prohibit obtaining gains on larger models.

Experimental results presented in this section are encouraging, but may not extend to all models. For example, for models that require generating long trajectories or include optimal strategies of significant size we may experience a significant slowdown. Nevertheless, for models with favourable structure, we outperform PRISM-games by several orders of magnitude. Given that our results for MDPs also outperform PRISM, a leading probabilistic model checker, we believe our algorithms can be considered as a preferred solution method when solving large probabilistic models with certain structure.

7.6 Summary

We have presented a framework for controller synthesis for stochastic games using learning algorithms. For reasons of clarity, we first presented our framework for models without end components. We proved that our method converges to the correct values for probabilistic reachability, while keeping the expected total reward as future work. We showed a simple example where our method will only visit a subset of the states of the model, and yet obtains the same results as standard methods for solving stochastic games such as value iteration.

After looking into several PRISM case studies, it became apparent that only a small subset of models do not contain end components. To support a wider range of models, we extended our framework to handle stochastic games with one-player ECs and arbitrary MDPs. We developed an on-the-fly EC detection method that is able to identify end components based on the generated trajectories. As our work aims at practical results, we proposed three different heuristics that can be used for models with different underlying structure.

A crucial element of any evaluation of a method based on heuristics are experiments on a range of case studies. We implemented our methods as an extension of PRISM-games and compared against the fastest available engine. Our method was able to outperform PRISM-games by several orders of a magnitude. This was possible due to the partial exploration of the state space that our algorithm relies on. It should be noted that not all models have a structure that is favourable to heuristics, and in some cases our methods performed worse than PRISM-games.

Chapter 8

Case Study and Implementation

8.1 Introduction

One of the key aims of this thesis is to develop methods that can be applied to real-world examples. In this chapter, we demonstrate the controller synthesis framework from Chapter 4, along with methods from Chapters 5, 6, and 7, to an open-source stock monitoring application called *StockPriceViewer*. Controller synthesis at runtime is used in the context of *StockPriceViewer* to allow the application to update stock information in a timely manner while using multiple stock information providers.

We run each method developed in this thesis and demonstrate how they improve the practical applicability of our framework. For *incremental model construction* from Chapter 5, we consider a scenario where the number of monitored stocks increases over time and show how incremental methods can shorten the time that *StockPriceViewer* has to wait for a new controller. We use *learning-based controller synthesis* from Chapter 7 to decrease the initial time that *StockPriceViewer* has to wait before it receives the first controller strategy. A more detailed run of the framework has been considered for *permissive controller synthesis* from Chapter 6. Permissive controllers are employed to improve the robustness of *StockPriceViewer* and handle a scenario when one of the stock information providers becomes temporarily unavailable.

We begin the chapter by describing the *StockPriceViewer* application in Section 8.2. In Section 8.3, we explain how each element of the framework has been implemented and describe the PRISM model of *StockPriceViewer*. The evaluation of the methods from Chapters 5, 6, and 7 on *StockPriceViewer* has been done in Section 8.4. In Section 8.5, we summarise the results obtained in this chapter.

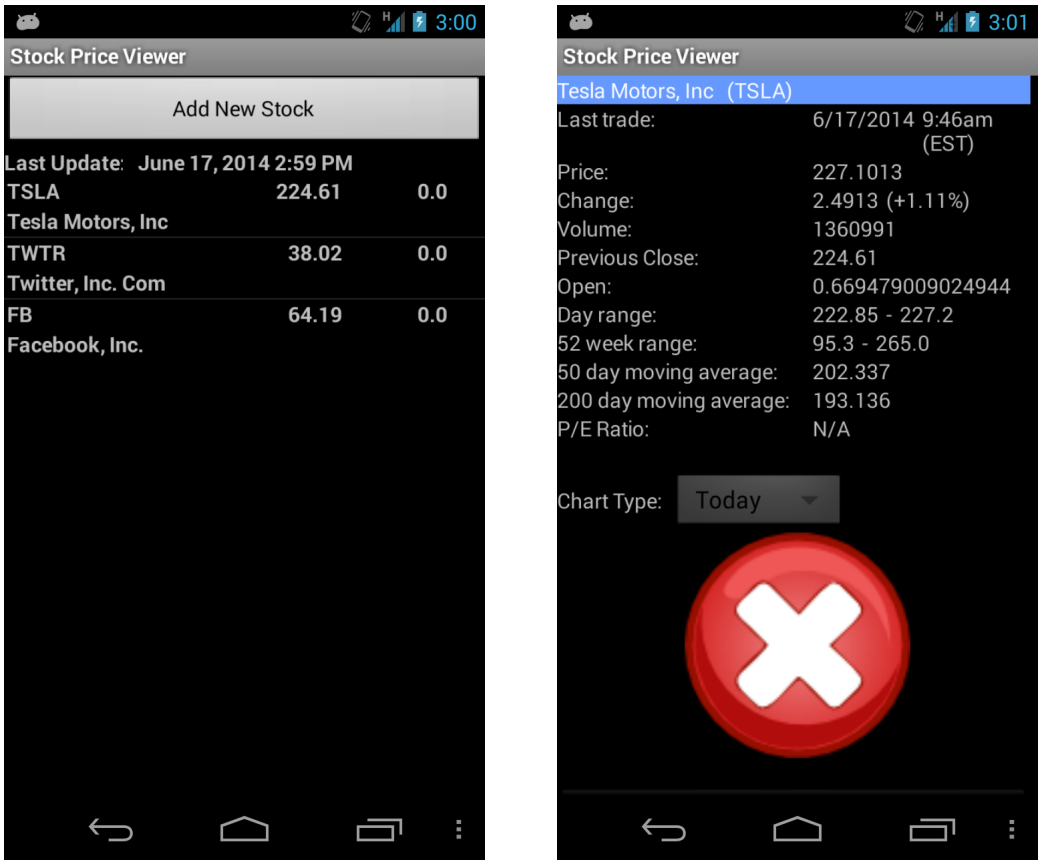


Figure 8.1: *StockPriceViewer* application.

8.2 *StockPriceViewer* Application

StockPriceViewer [130] is an open-source Android application that enables users to view real-time stock prices. In Figure 8.1, we show two screenshots from the application. The left-hand side shows the main screen that contains a stock portfolio of three companies; to the right of each company name we can find the stock price. On the right-hand side, we show a second screen that appears after clicking on the particular stock from the list and includes detailed pricing information.

The pricing information is updated every five seconds using an Internet-based stock information provider. Different providers are characterised by different response times and different failure probabilities. From the user perspective, the most important property of the application is to provide current stock prices without delays, so that he or she can quickly respond to market variations. To rapidly update information about the stock portfolio, the application needs to choose from a variety of providers, a challenging task given that providers with short response times may suffer from high failure probability and reliable providers may be slow.

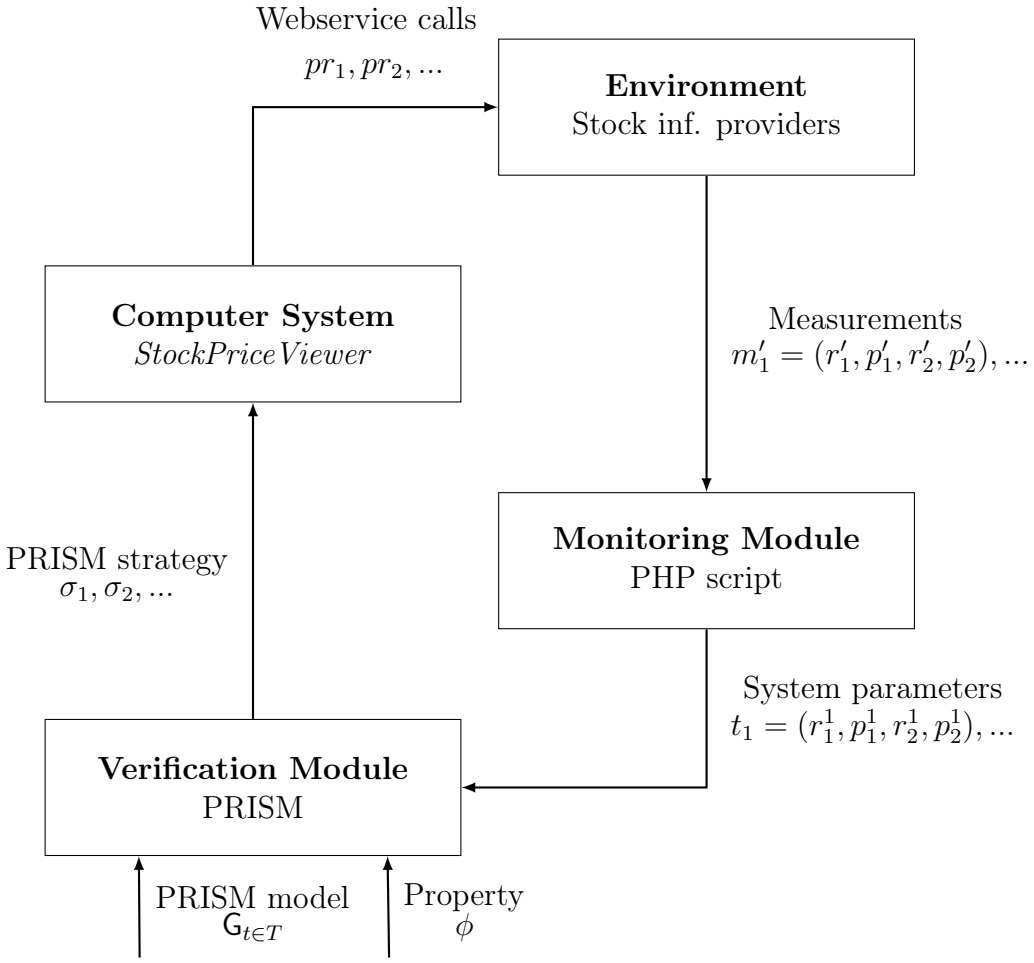


Figure 8.2: Verification and Controller Synthesis at Runtime.

8.3 Controller Synthesis for *StockPriceViewer*

The framework described in Chapter 4 is used to formalise the decision-making process in the presence of failures. We are able to decide on the optimal provider, taking into account both its response time as well as the failure probability. Figure 8.2 depicts an instance of the framework in the context of the *StockPriceViewer* application.

In the coming sections, we present how each of the methods developed in this thesis fits into the framework. A full run of the framework is done for permissive controller synthesis developed in Chapter 6. Using permissive controller synthesis, we show how to avoid a lack of portfolio updates in cases when the optimal provider becomes temporarily unavailable.

Below, we describe how each element of the framework has been implemented in the context of the *StockPriceViewer* application.

8.3.1 Computer System

The *StockPriceViewer* application plays the role of the computer system in the framework. Controllable actions in the computer system are represented in *StockPriceViewer* by actions that can be used to retrieve information about stocks from different stock information providers. Contacting a stock information provider is typically done by a simple HTTP request with a stock name as a parameter. The result of the query is a string of comma-separated values containing relevant stock information.

In *StockPriceViewer*, the choice of the provider has been hard-coded in the configuration and cannot be changed. We modified the application by replacing the module that is responsible for picking the provider. The choice of the optimal provider will be made on the basis of the PRISM strategy supplied by the verification module. The added module uses a simple HTTP request to query if the verification module computed a new PRISM strategy.

8.3.2 Monitoring Module

For each provider we monitor both the response time and failure probability. The monitoring is done by a simple PHP script. The script, using an HTTP request, retrieves stock information from a given provider and reports response time and failure information. Monitoring happens every second and the measured values are stored and averaged over a five minute period. The verification module can obtain the measurement data using an HTTP-based interface.

8.3.3 Verification Module

The verification module is implemented as a PHP script that runs PRISM-games every five minutes. We use a five minute time period because we want to generate a new strategy every time new data from the monitoring module becomes available.

The script runs PRISM-games, giving as the inputs the stochastic game model, parameters from the monitoring module and the property. The output of PRISM-games is a textual representation of the controller strategy; we use the standard syntax provided by PRISM. Below, we present a stochastic game model of the *StockPriceViewer* application.

Stochastic game model of *StockPriceViewer*

We carefully analysed the source-code of the *StockPriceViewer* and manually derived a model of the application. We model a sequence of stock portfolio updates, where the application has to decide which provider should be chosen and the environment may try

to spoil the choice. The full PRISM models for two scenarios can be found in Figures D.1, D.2, and D.3 in Appendix D. In Figure 8.3, we present a simplified version showing a single update of one stock with two possible data providers. In comparison, the model in Figure D.1 supports an arbitrary number of stocks and three different providers. The box in Figure 8.3 contains abbreviated information about actions available in states s_5, s_6, s_7, s_8 that is omitted from the figure.

In the model, the controller states (marked using $\Diamond$) represent choices that are under the control of the framework. In those states the application is able to choose between different stock information providers pr_1 and pr_2. Each of the providers is characterised by a different probability of failure (p_1^1 and p_2^1) and response time (r_1^1 and r_2^1). We use data from the monitoring module from Figure 8.2 for probability of failure and response time values.

The environment states (marked using $\Box$) represent the environment. When the given data provider fails, the environment can choose from two options. It can either *block* or *retry* the request. Blocking the request represents a situation when the provider blocks any further requests from our machine. For large portfolios, we can run hundreds of queries per second, and therefore we can be perceived as an Internet robot and thus automatically lose the ability to contact the provider. The retry action allows us to retry the request. We specify an upper bound on the number of retries, after which the provider is deemed as not working.

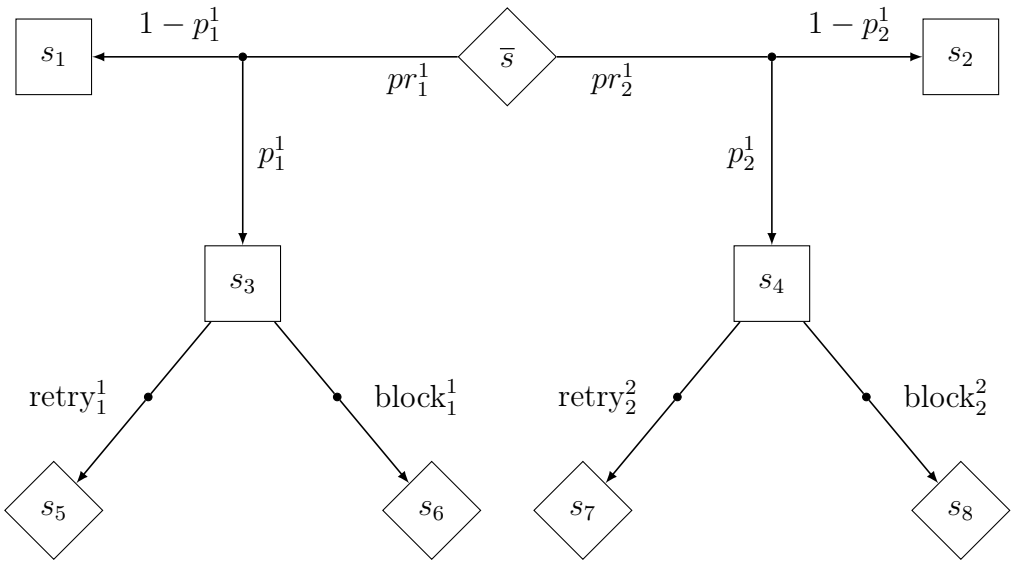
In Table 8.1, we show statistics about the model size and the number of transitions. We present the results for both the model with three providers (*android_3*) and four providers (*android_4*). We use *max_retry* to define the maximal number of retries, while *stock_to_query* represents the number of stocks. The number of stocks influences the state space size linearly. In contrast, an increase in the number of retries causes the state space to blow-up, and models for a large value of the parameter might be challenging to consider.

8.4 Experimental Results

In the next sections we describe how each of the methods described in this thesis was employed within the framework.

8.4.1 Incremental Model Construction

One of the parameters of the model of the *StockPriceViewer* application is *stock_to_query*, which represents the number of stocks in the portfolio. This is a parameter that depends



	$r(\overline{s})(pr_1^1) = r_1^1$	$r(\overline{s})(pr_2^1) = r_2^1$
$A(s_5) = \{pr_1^2, pr_2^2\}$	$r(s_5)(pr_1^2) = r_1^1$	$r(s_5)(pr_2^2) = r_2^1$
$A(s_6) = \{pr_2^3\}$		$r(s_6)(pr_2^3) = r_2^1$
$A(s_7) = \{pr_1^3, pr_2^4\}$	$r(s_7)(pr_1^3) = r_1^1$	$r(s_7)(pr_2^4) = r_2^1$
$A(s_8) = \{pr_1^4\}$	$r(s_8)(pr_1^4) = r_1^1$	

Figure 8.3: Stochastic game model of *StockPriceViewer* application parameterised with t_1 from Figure 8.2.

Name	Parameters		States	Transitions
	max_retry	*stock_to_query*		
android_3	1	10	481	861
		60	2,881	5,211
	2	10	1,918	3,804
		60	11,368	22,854
	3	10	4,897	10,137
		60	28,897	60,687
android_4	1	10	641	1,328
		60	3,841	8,128
	2	10	2,458	5,584
		60	14,608	33,934
	3	10	6,177	14,556
		60	36,577	88,156

Table 8.1: Size of the stochastic game model of the *StockPriceViewer* application.

on the user as he or she may add or remove monitored stocks during the application lifetime. Therefore, it might be desirable to compute several strategies for a sequence of *stock_to_query* values. By doing so, we can offer the application additional strategies in case the user adds or removes stocks during the time that it takes to generate a strategy. This process can be speeded-up employing the techniques developed in Chapter 5.

Results

The command used for running the tool from Chapter 5 can be found in Figure 8.4. We start by setting the value for system parameters using the standard PRISM-games *const* switch. To build models for a sequence of values of the *stock_to_query* parameter we use the PRISM-games syntax for running experiments. This is done by adding ":" between the parameter values, in our case *stock_to_query=2500:2501*. We use *web_stock_0_fail*, *web_stock_1_fail*, and *web_stock_2_fail* to specify the probability of failure of each of the providers in the model. Those values are computed by the monitoring module. The *max_retry* value comes from the configuration. The property that the generated strategy has to satisfy is set using the *prop* switch, but it does not play a role in the model construction process. For the purpose of our experiment, we use the *nover* switch that informs PRISM-games to only perform model construction and skip strategy generation. We do this for reasons of clarity to only focus on improvements in the model construction process. To inform PRISM-games to use incremental model construction algorithms we use the *incr* switch.

The result of running the command can be found in Figure 8.4. The aim of our

Model [parameters]	Parameter values	States	Model construction time (s)	
			PRISM (explicit)	Incr. (Alg. 3)
android_3 $[r, s]$	$3,500 - 501$	240,097−240,577	2.9	2.4
	$3,1000 - 1001$	480,097−480,577	8.2	4.9
	$3,1500 - 1501$	720,097−720,577	13.4	7.8
	$3,2000 - 2001$	960,097−960,577	21.7	11.3
	$3,2500 - 2501$	1,200,097−1,200,577	28.0	13.6

Table 8.2: Experimental results for incremental model construction for *StockPriceViewer*.

method was to seamlessly integrate within the PRISM-games model construction process, so that the user will only notice its presence through the reduced running times. The only information printed by our algorithms is the number of states that the algorithm re-explored in the old model while incrementally building the new model. This information is printed with *Re-explored* used as a prefix.

The results of running the incremental model construction algorithm on the model of *StockPriceViewer* can be seen in Table 8.2. In the table we first give the case study name, followed by parameter values used and the number of states for each value of the parameters. We compare the incremental model construction against the PRISM-games model construction algorithm.

Incremental model construction offers a significant improvement over the PRISM-games model construction algorithm. The main reason for this improvement can be found in the output generated by PRISM-games in Figure 8.4. We can see there that our method had to re-explore only 144 states out of $1,200,097$, and in the incremental step 480 states had to be added. In total there were only 624 states on which we had to evaluate all commands in the model. As we mentioned in Chapter 5, evaluating a command on a state can be a time-consuming process. In comparison, PRISM-games model construction evaluated all commands for $1,200,577$ states, causing a significant slowdown. While incremental model construction improves the performance of the model construction process, it still requires keeping the whole model in memory. In the next section, we apply learning-based methods that will only keep a small subset of states in the memory, leading to significant improvements in strategy computation time.

8.4.2 Learning-based Controller Synthesis

Before *StockPriceViewer* can start querying stock information providers, it needs to obtain a strategy from the verification module. This can be challenging in cases when computing

```
thesis-examples$: ./prism android_generated_3.smg android.props
                          -const 'max_retry=3,stock_to_query=2500:2501
                          web_stock_0_fail=0.001,
                          web_stock_1_fail=0.002,
                          web_stock_2_fail=0.003'
                          -prop 1 -nover
                          -incr

PRISM-games

---------------------------------------------------------------------

Building model...

Computing reachable states... 289393 532382 705973 1096788 1200097 states
Reachable states exploration and model construction done in 16.755 secs.
Sorting reachable states list...
Type:        SMG
States:      1200097 (1 initial)
Transitions: 2167527
Choices:     1807527
Max/avg:     3/1.51

Time for model construction: 17.746 seconds.

---------------------------------------------------------------------

Building model...

Re-explored: 144 states

Computing reachable states...1200577 states
Reachable states exploration and model construction done in 0.009 secs.
Sorting reachable states list...
Type:        SMG
States:      1200577 (1 initial)
Transitions: 2168394
Choices:     1808250
Max/avg:     3/1.51

Time for model construction: 9.438 seconds.
```

Figure 8.4: Running incremental model construction for *StockPriceViewer*.

the strategy takes a substantial amount of time. We found that the model that we developed in this chapter is sensitive to the values of the max_retry parameter. For large values of the parameter it may take several minutes before the verification module is able to compute the strategy that is used by the application. In the meantime, the values of system parameters used to parameterise the model may change, making the computed strategy obsolete. Below, we show a run of our tool developed in Chapter 7, which proved to only needs seconds to generate the optimal strategy for the model of *StockPriceViewer*.

Results

We first report the command that was used to run our tool and then present the results in Table 8.3. The command can be seen in Figure 8.5. Firstly, we set the value of all system parameters. As before, we use the *const* switch to set the values of system parameters. The property that we consider is set using the *prop* switch and maximises the probability of successfully updating a portfolio of stocks. As the precision parameter we use $\varepsilon = 10^{-8}$. The generated strategy will be stored in the *strategy.strat* file, which is set using the *exportadv* switch.

To run our algorithms we use the *-brtdp:heuristic=MD,verbose* switch. The *brtdp* switch informs PRISM-games that it should use our learning-based methods as the main solution method. The *heuristic* switch specifies the heuristic that will be used when exploring the state space; possible values are *MD, RR*, and *RTDP*. The *verbose* switch informs PRISM-games to print as much information as possible about the run of the algorithm.

When running our algorithms in verbose mode, we report several interesting statistics about the solution process. Every time MECs are collapsed, we print a message describing the time spent on collapsing. Every fixed number of steps, we print the number of states visited by the algorithm as well as the current upper and lower bounds on the value of the property in the initial state. After the optimal strategy has been computed, we print the number of reachable states under that strategy and again report upper and lower bounds on the value of the property.

A summary of running our methods on the *StockPriceViewer* model is presented in Table 8.3. Please note that the results have been averaged over a 20 runs due to the randomised nature of our algorithms. We first report the model name, the parameter value, the number of states in the model and the property. We report the PRISM-games time for computing the optimal strategy, as well as the fastest heuristic. In the last column we report the number of states visited by the heuristic.

We can see that, for values of max_retry higher than 20, PRISM-games can take

```
thesis-examples$: ./prism android_generated_3.smg android.props
                          -const web_stock_0_fail=0.001,
                          web_stock_1_fail=0.002,
                          web_stock_2_fail=0.003,
                          'max_retry=30, stock_to_query=10'
                          -prop 1 -epsilon 1e-8 -exportadv strategy.strat
                          '-brtdp:heuristic=MD,verbose'

PRISM-games

---------------------------------------------------------------------

Model checking: <<controller>> Pmax=? [ F stock_querued=stock_to_query ]
Model constants: max_retry=30,stock_to_query=10
Solution method: heuristics.search.HeuristicBRTDP
Using heuristic: M-D

Starting Prob0 (maxmin)...
Prob0 (maxmin) took 2 iterations and 0.0 seconds.


Starting MEC collapsing
MEC collapsing done 0.003 secs.
Visited states: 24
Lower bound: 0.0
Upper bound: 1.0

Visited states: 14051
Lower bound: 0.9999986977016975
Upper bound: 0.9999986977016975
Reachable under optimal strategy: 265

Learning-based model checking completed in 1.411 secs.

Result (result for coalition [controller]): 0.9999986977016975
```

Figure 8.5: Running learning-based controller synthesis for *StockPriceViewer*.

Name [parameters]	Parameter values	States	Property	Time(s)		Visited States
				PRISM	M-D	
android_3 $[r, s = 10]$	15	364,033	$\mathtt{P}_{max=?}[\mathtt{F}\ done]$	108.6	1.4	13,093
	20	832,168		320.3	1.4	13,389
	25	1,589,953		741.4	1.4	13,010
	30	2,707,138		*	1.5	13,403

* Algorithm did not terminate within 30 minute time-out.

Table 8.3: Experimental results for learning-based controller synthesis for *StockPriceViewer*.

several minutes to compute the optimal strategy. In contrast, the results provided by learning-based methods allow for the generation of the controller in just above a second. The reasons are clearly visible in Figure 8.5. We visit only 14051 states, which is a small subset of the $2,707,138$ states in the full model. Similarly, we run the MEC collapsing and precomputation algorithms on a model of just 24 states, which barely takes any time. The same precomputation algorithm run on the full model would take several minutes to complete. In the next section, we present a full run of the framework and show how permissive controller synthesis can improve the functionality of *StockPriceViewer*.

8.4.3 Permissive Controller Synthesis

When running *StockPriceViewer*, we often experienced temporary unavailability of the currently used stock information provider. This caused the application to stop updating the stock prices, which would only resume when the provider became online again. A simple solution to this problem would be to generate a new strategy for the model of *StockPriceViewer* without the malfunctioning provider. While it would solve our problem, it may take a prohibitively long time to compute such a strategy. By the time that a new strategy has been computed, the unavailable provider may become online again and a new strategy will have to be computed. Instead, to solve this problem we use techniques developed in Chapter 6. We will compute a multi-strategy that defines a list of providers that can be used. In case one of the providers becomes unavailable, the application can switch to a different provider without any recomputation.

Results

In our experiment we can consider a full implementation of the framework. We run *StockPriceViewer* on a Galaxy Nexus phone with Android 4.3. The monitoring module and the verification module run on a machine with a 2.8GHz Xeon processor and 32GB of RAM, running Fedora 14.

The monitoring module is a PHP script that, every second, contacts each of the available providers and stores the response time and information if the query failed. Values are averaged over a five minute period and available using a web interface for the verification module. The verification module is implemented as a PHP script, which calls PRISM-games over a command line; in Figure 8.6 we present the command used to generate a multi-strategy in our experiment.

The command starts with specifying system parameters in the PRISM model of the application. Below, we will expand how we computed those values. We use a model that allows four different providers and a property that minimises the expected time required

```
thesis-examples$: ./prism android_generated_4.smg android.props
                    -const 'web_stock_0_fail=0,
                     web_stock_1_fail=0.00002,
                     web_stock_2_fail=0.00003,
                     web_stock_3_fail=0.00004,
                     web_stock_0_response_time=100,
                     web_stock_1_response_time=200,
                     web_stock_2_response_time=600,
                     web_stock_3_response_time=700,
                     max_retry=1, stock_to_query=60'
                    -prop 2
                    -run_solver
                    -lp_solver gurobi_cl
                    -lp_output_file encoding.lp -lp_result_file multi_strat.sol
                    -exportadv multi_strategy.strat

PRISM-games

------------------------------------------------------------------------

Model checking: <<controller>> R<=66000 [ Fc stock_querued=stock_to_query ]
Model constants: max_retry=1,stock_to_query=60

Building model...

Computing reachable states... 3841 states
Generating LP encoding generation took: 6.28 seconds
Running solver: TimeLimit=300s Threads=1 ResultFile=multi_strat.sol encoding.lp

Set parameter TimeLimit to value 300s
Set parameter Threads to value 1

Gurobi Optimizer version 5.5.0 build 11815
Copyright (c) 2013, Gurobi Optimization, Inc.

Read LP format model from file encoding.lp
Variable types: 2185 continuous, 4509 integer (4509 binary)
Found heuristic solution: objective -0.1522843

Root relaxation: objective -1.544036e-01, 3705 iterations, 0.07 seconds

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0   -0.15440    0 1820   -0.15228   -0.15440  1.39%     -    1s
     0     0     cutoff    0        -0.15228   -0.15228  0.00%     -    2s

Explored 0 nodes (17945 simplex iterations) in 2.16 seconds
Thread count was 1 (of 4 available processors)

Optimal solution found (tolerance 1.00e-04)
Best objective -1.522843353995e-01, best bound -1.522843353995e-01, gap 0.0%

Wrote result file 'multi_strat.sol'
```

Figure 8.6: Running permissive controller synthesis for *StockPriceViewer*.

for updating information about 60 stocks.

To differentiate between multi-strategies, we use a penalty structure. In the case of *StockPriceViewer* our penalty structure assigns higher penalties for providers with short response times, therefore preferring to block slow providers. The definition of the penalty structure can be found in Figure D.4 in Appendix D.

We inform PRISM-games to run our algorithms by using the *run_solver* switch. The *lp_solver* switch is used to define which solver is used, possible options include *gurobi_cl* and *cplex*. The MILP encoding is saved in the file defined using *lp_output_file* and the output of the solver is saved in the file defined by *lp_result_file*. Based on the content of the *multi_strat.sol*, our tool will generate a PRISM representation of the multi-strategy, which will be saved in the file specified by the *exportadv* switch.

The multi-strategy used in the experiment has been generated within seconds. This is possible due to a relatively small model that we considered. For larger models, we may not be able to generate an optimal multi-strategy within a reasonable time threshold. In those cases, we could consider changing the *TimeLimit* parameter of the Gurobi solver. This is the parameter that controls the time after which the solver returns the current best solution. In our experiments the parameter has been hard-coded to 300 seconds and after that time the solver will return possibly non-optimal but a sound multi-strategy that can be used by *StockPriceViewer*.

In our experiment, we use four different stock information providers, each with a different probability of failure and response time. In order to control these values, we used a purpose-made proxy server. The proxy server is able to maintain the response time and probability of failure at specified levels and avoid fluctuation due to network configuration. The proxy server is also implemented as a PHP script. Data about current stock prices is gathered from Yahoo! Finance [1] website. We use 100, 200, 600, 700 milliseconds as response times and 0, 0.00002, 0.00003, 0.00004 as the failure probabilities.

In the experiment, we show how *StockPriceViewer* reacts to the occasional unavailability of the currently used provider when using a classical and a permissive controller. We focus our experiment on a single run of the framework, without recomputing the controller. The run takes five minutes. Every five seconds, *StockPriceViewer* tries to update the value of the portfolio, and every 30 seconds the currently used provider fails for a period of ten seconds. We use $max\_retry = 1$ and $stock\_to\_query = 60$ as parameters for the PRISM model.

When the currently used provider fails, the classical controller is unable to make an informed decision as to which provider should be chosen instead. We model this by

[1] http://finance.yahoo.com

picking the provider at random. The permissive controller will switch to the provider that is defined by the multi-strategy.

In Figure 8.7, we report the results of our experiment. On the x-axis we report the time since the beginning of the experiment. The y-axis represents the response time experienced by *StockPriceViewer* while performing portfolio updates. The dotted line represents the response times obtained using standard controller synthesis, while the solid line represents results that are obtained using a multi-strategy. Please note that in the experiment the stock portfolio consists of ten stocks, and therefore the reported times are proportionally higher compared to the response time of the provider in use.

At the beginning of the experiment, *StockPriceViewer* reports the same response times, as both controllers choose the same providers. After the first failure, the single-strategy controller randomly picks the last provider, causing a significant increase in the response time. On the other hand, the permissive multi-strategy controller picks the second provider, causing only a slight increase in the response time. After the first provider becomes available again, both controllers continue to pick the same providers. This leads to similar response times.

During the second failure, the single-strategy controller randomly chooses the second provider, but later in the experiments such a situation does not occur and the single-strategy controller performs much worse. We report an increase of the response time for the permissive controller midway through the experiment. This is attributed to network problems, as an investigation of the phone log revealed that the provider with 200 milliseconds response time was used. Overall, the permissive controller is able to successfully handle occasional unavailability of one of the providers. It takes 111.13 seconds for the single-strategy controller to perform 60 portfolio updates, compared to 88.03 seconds using the permissive controller.

8.5 Summary

In this chapter, we evaluated the techniques developed in this thesis on *StockPriceViewer*, an open-source stock monitoring application. We described how controllers can be used by *StockPriceViewer* to decide how to choose between multiple stock information providers and allow for updating stock information in a timely manner. Next, we showed how each element of our controller synthesis framework from Chapter 4 has been implemented. In Section 8.4, we evaluated all methods developed in this thesis using an implementation of our framework.

For each method, we described a scenario in which *StockPriceViewer* can benefit from applying the method and presented command line parameters needed to run it. For

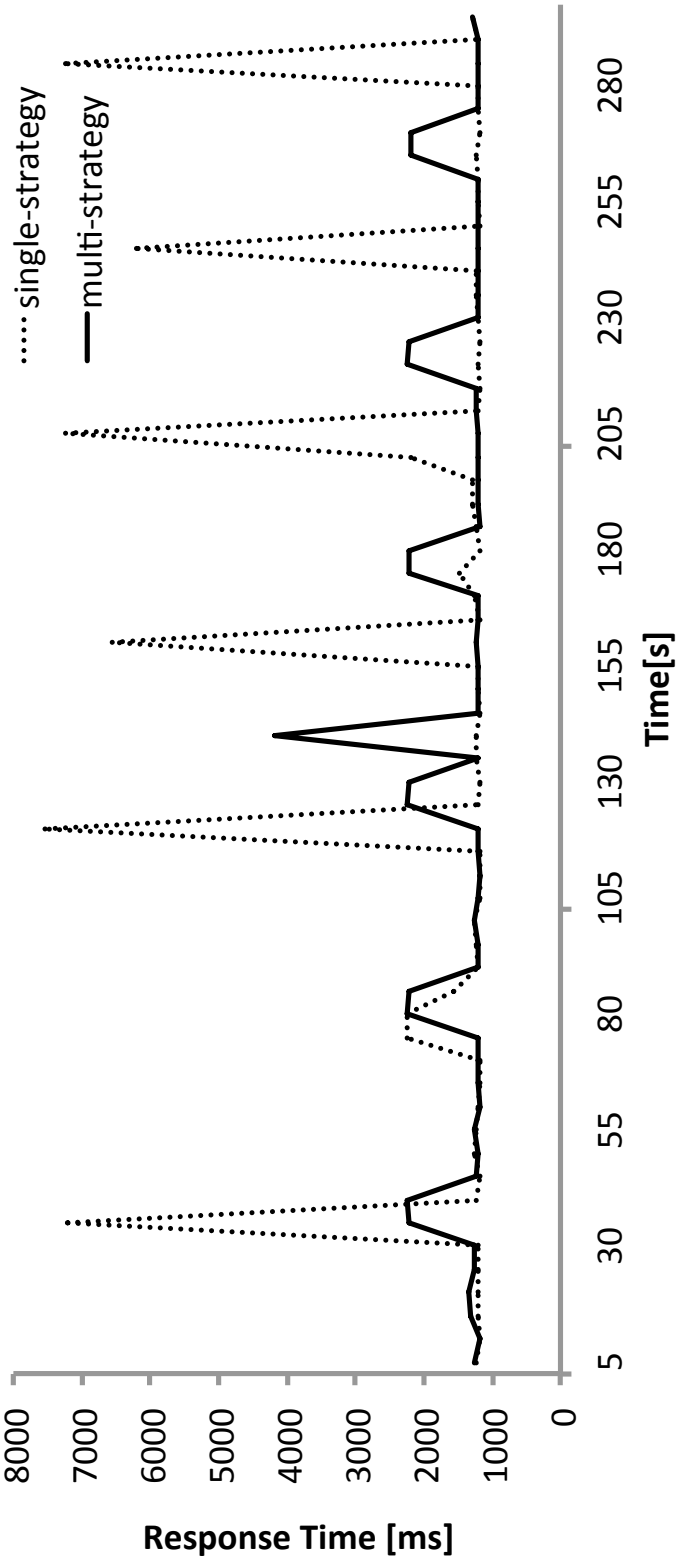


Figure 8.7: Improved robustness of the *StockPriceViewer* application.

permissive controller synthesis, we considered a full run of our framework. Our method has been used to improve *StockPriceViewer* ability to overcome temporary unavailability of stock information providers.

While we showed that each of the methods we developed can be successfully applied in the context of *StockPriceViewer*, it would be beneficial to also consider different case studies. One of the possible directions include cloud systems, where controller synthesis could be used to decide how many servers should be kept running or on which machines a service should be deployed

Chapter 9

Conclusions

9.1 Summary and Evaluation

The main aim of this thesis was to make runtime controller synthesis fast, robust and scalable. To improve performance of controller synthesis, we developed an incremental model construction method. Incremental model construction exploits the fact that applying controller synthesis at runtime often requires re-verifying models that share a similar structure and can be built incrementally. We addressed the robustness objective by developing permissive controller synthesis that is useful in cases when some of the actions of the running system become temporarily unavailable. The scalability objective has been tackled by means of learning-based methods drawn from fields such as planning and reinforcement learning. By generating random trajectories through the model, our method is able to approximate the value of the property while only visiting a possibly small number of states.

Another aim of this thesis was to develop techniques that not only expand knowledge about the theory of controller synthesis, but also perform well in practice. For each of the methods, we developed a full implementation that has been evaluated using an extensive set of PRISM case studies. Being able to run a large number of experiments allowed us to better understand the limitations of our work, and in many cases drove the research in a direction that promised better practical results. For example, optimisations presented in Section 6.3.4 are a direct consequence of disappointing experimental results that we obtained for the MILP encoding without any optimisations. Heuristics proposed in Section 7.5 have similar origins, where the RTDP heuristic was implemented first but did not perform well.

This thesis is only an initial attempt to formalise, solve and implement the problem of controller synthesis at runtime. The amount of work needed to better understand

the subject is vast, with particular emphasis on developing case studies and providing additional implementation for the framework from Chapter 4. In Chapter 8, we presented a case study that implements our framework and is used to evaluate each method developed in this thesis. While the results are encouraging, it is important to consider more case studies representing a wider range of possible scenarios. We expand on future directions in Section 9.2.

Below, we summarise each of the three main contributions of this thesis. We first explain the problem we were trying to address, then describe the solution, and summarise the novelty of our method.

In Chapter 5, we demonstrated that incremental model construction can yield significant performance gains in cases where we build a sequence of models that differ in values of system parameters. By analysing the high-level model representation, we were able to pinpoint all states that need to be re-visited. Starting from the previously built model and the set of states that needs be re-visited, we were able to build the new model incrementally. Typically, this required significantly fewer time-consuming operations, such as state evaluation or state storage, when compared to building the model from scratch.

While there exist methods for incremental model checking for probabilistic models [61, 94, 95], to the best of our knowledge this is the first time when incremental model construction has been proposed for PRISM's modelling language.

In Chapter 6, we introduced permissive controller synthesis to improve the robustness of the controllers used in the framework from Chapter 4. We defined a notion of a multi-strategy as a generalisation of strategies used in traditional controller synthesis. We defined penalties that allow for a comparison of multi-strategies to decide which one is more permissive. Finally, we formulated the permissive controller synthesis problem and proved that, even in the simplest case, it is an NP-hard problem. To be able to synthesise permissive controllers in practice, we developed an MILP encoding. Subsequently, we showed that, while the amount of computation needed for computing permissive controllers is higher than for classical controllers, it is still possible to generate controllers for models containing tens of thousands of states.

The novelty of permissive controller synthesis is both in the theory that we developed, as well as in the implementation that involves MILP. Permissive controllers in the context of non-stochastic games have already been pursued in [15, 11]. We believe that our work is the first to define permissive controllers for stochastic games. In the past, MILP has been used for probabilistic model checking [121]. To the best of our knowledge, it has not yet been used in the context of stochastic games that are larger than a few states [120], and so our MILP encoding has resulted in the first practical tool for generating robust multi-strategies for runtime use.

In Chapter 7, we developed a learning-based framework for (classical) controller synthesis for stochastic games. The motivation was to improve the scalability by relying on only partial exploration of the probabilistic model. We defined a framework based on RTDP and BRTDP algorithms that is applicable to a subclass of stochastic games and to arbitrary MDPs. Our method involves a heuristic-based exploration of the model state space, where each generated trajectory improves the value of the property being verified. Our implementation showed that partial exploration can be very effective for a number of PRISM case studies. We improved the performance of controller synthesis by several orders of magnitude compared to state-of-the-art PRISM implementation, while being able to give precise error bounds on the value of the property.

The main strength and novelty of this work derives from the highly optimized implementation and a choice of heuristics that work for models with different structural characteristics. Recently, heuristic-based methods for model checking and controller synthesis have been developed [67, 84]. To the best of our knowledge, we were the first to apply those methods to stochastic games. Moreover, in the case of MDPs and probabilistic model checking, we believe we provided the first efficient implementation of heuristic-based controller synthesis supported by an extensive experimental evaluation.

9.2 Future Work

Our work in this thesis can be extended in numerous ways. We divide this section by each chapter and suggest possible future research directions.

The framework from Chapter 4 can be extended in several ways. In this thesis, we did not extensively study the influence of the controller on the running system. A more control-theoretic approach is needed and issues such as stability should be considered. Our framework does not consider a specific implementation of the monitoring module; in our implementation we used a simple average to obtain the value of parameters. A more robust approach to data fusion should be pursued, possibly inspired by KAMI [52] or work presented in [22].

While we provided an implementation for each of our contributions, the framework as a whole was only partially implemented. It would be vital to provide a tool-like implementation. In Chapter 8, we considered a case study which uses the presented framework. The first step in extending this work would be to develop a case study where each of our methods can be run simultaneously. This contrasts with the work we present in the thesis where each method uses a different set of parameters. Subsequently, we should provide new case studies in order to be able to fully understand the strengths and weakness of using controller synthesis at runtime. Possible directions include cloud systems, webservice

composition, as well as considering randomly generated models.

We evaluated all our methods on one case study in Chapter 8. Each of our methods has been run separately with different model parameters. It would be beneficial to consider a case study where all three methods that we developed can be run simultaneously.

The incremental model construction from Chapter 4 addresses only a small part of the controller synthesis process. Another important part is effective quantitative verification methods. There exists some initial work on incremental quantitative verification [94], but those methods do not work in the presence of structural changes. It would be interesting to see if the techniques from Chapter 4 could be applied in this scenario. Another possibility is developing incremental precomputation algorithms for probabilistic models. Precomputation is used to compute the set of states that satisfy the property with probability 1 (equivalently with probability 0). Currently used precomputation algorithms [59] do not work incrementally and require recomputation even after a small change in the model structure.

At the level of the PRISM modelling language, one possibility is to consider a case where we can add or remove modules. In many PRISM models, modules are used to model a possibly variable number of system components. One example might be a model for the Microgrid management algorithm [30], where participating households are modelled using modules. In a runtime scenario the number of households is likely to change, which at this point is not supported by our method.

For permissive controllers, one of the open problems includes deciding if an optimal randomised multi-strategy for a dynamic penalty scheme exists. Another possibility is to consider not only memoryless multi-strategies, but also history-dependent ones. It seems likely that the penalty schemes that we proposed for memoryless strategies may not be adequate in the history-depended case, leading to another possible avenue to investigate.

One of the main features of MILP solvers is the ability to provide intermediate results of the computation. While possibly non-optimal, the result may be good enough to use in practice. Our current implementation uses a simple timeout to stop the computation. It would be worthwhile to use a more integrated approach, where each intermediate solution could be examined and the computation could be stopped based on some criterion other than a timeout.

The most important future direction for the work from Chapter 7 is adding support for computing the expected total reward. One of the main technical problems behind it is the fact that we cannot easily estimate the value of the upper bound of the expected reward in a state. This problem has been considered in the planning community and relaxation-based methods are one possible solution [99].

The on-the-fly EC method is a costly element of our framework. One could replace it

by precomputing ECs before starting the synthesis method. Decomposing into end components does not depend on the probability values on transitions in the model. Therefore, the precomputation could possibly be done using a SAT solver. A similar approach has been proposed by [77] but in the context of counter-example generation. Another possibility is to extend our framework to support other heuristic-based algorithms. Possible candidates include LAO* [69] and Bayesian RTDP [110].

9.3 Conclusion

In this thesis, we successfully developed and evaluated three techniques that address requirements for controller synthesis at runtime using stochastic games. For each of the techniques, we formulated the theory, developed the implementation, and performed the experimental evaluation.

The practicality of the presented methods was one of our main motivations. Our methods were proved to work well in practice and, when comparison was possible, we were able to obtain results that significantly improve over existing state-of-the-art methods.

The work done in this thesis is only a starting point for future research on controller synthesis at runtime. We proposed a number of possible directions in which to extend it. One of the most important aspects is to consider more case studies, which will ensure the relevance of our methods in practice.

Appendix A

Benchmarks

The techniques developed in this thesis have been evaluated on a set of PRISM case studies. In the coming sections, we provide a short description of each case study. All PRISM models of the case studies are available online on the website that has been prepared for this thesis [129]. We divided the case studies based on the type of probabilistic model that they employ.

A.1 DTMC Case Studies

crowds([112])

Crowds is a protocol aimed at providing anonymous web browsing. This is achieved by routing the web traffic through a set of randomly chosen routers. The PRISM model is used to verify the anonymity properties of the protocol. This case study is employed for incremental model construction and therefore we do not mention any specific PCTL property.

A.2 MDP Case Studies

mer([54])

The mer case study focuses on a resource arbiter module of the Mars Exploration Rover (mer). The resource arbiter is a piece of software that manages access to a resources available on-board the rover. The PRISM model is used to capture a possibly faulty communication channel between the arbiter and threads competing for the resources. The property that we consider is $\mathtt{P}_{max=?}[\,\mathtt{F}^{\leqslant 30}\ no\_deadlock\,]$, which expresses the maximum probability of a deadlock occurring within the first 30 steps of arbiter execution.

zeroconf([90])

Zeroconf is a protocol that can be used to dynamically assign an IP address to a device. The stochasticity in the protocol is introduced by using a random number generator to pick the initial IP address. After picking the IP address, a set of procedures is run that ensure that the address is not conflicting with other addresses in the network. The property that we verify on the PRISM model of the protocol is $\mathtt{P}_{min=?}[\,\mathtt{F}\ \mathit{configured}\,]$. This property expresses the minimum probability of successfully configuring the IP address.

firewire([93])

This case study aims at analysing properties of the leader election protocol of the IEEE 1394 High Performance Serial Bus (also known as FireWire). The purpose of the protocol is to dynamically elect a leader from the devices currently connected to the bus. The leader will later act as the manager of the bus. The model is probabilistic timed automaton interpreted as an MDP under digital clock semantics. The property that we analyse for this case study is $\mathtt{P}_{max=?}[\,\mathtt{F}\ \mathit{leader\_elected}\,]$, which expresses the maximal probability of electing a leader.

wlan([92])

The PRISM model developed in this case study models the two-way handshake mechanism of the IEEE 802.11 medium access control (WLAN protocol). The model includes two stations communicating with each other. The communication can be disrupted if both station transmit at the same time and create a collision. Similarly to the *firewire* case study, the model is probabilistic timed automaton interpreted as an MDP under digital clock semantics. We are interested in computing the maximum probability of the message being sent correctly. This can be captured using the property $\mathtt{P}_{max=?}[\,\mathtt{F}\ \mathit{sent\_correctly}\,]$.

A.3 Stochastic Game Case Studies

mdsm([30])

Microgrid Demand-Side Management (mdsm) is a case study concerning distributed energy management for a set of households that aim to minimise the peak energy demand. The algorithm is based on the work of Hildmann et al. [75]. In this thesis, we focus on two properties for this case study. The first ensures that one of the households deviates from the algorithm with a low probability, $\mathtt{P}_{\leqslant 0.01}[\,\mathtt{F}\ \mathit{deviated}\,]$. The second, $\mathtt{P}_{max=?}[\,\mathtt{F}^{\leqslant k}\mathit{first\_job\_arrived}\,]$, expresses the maximal probability that the household generates its first job within the first k steps. A job represents an energy consuming task that a household submits to the energy grid.

android

This case study is discussed in detail in Chapter 8.

cloud([21])

This case study is an extension of the work presented in [21]. We model a cloud system consisting of servers, virtual machines and the cloud administrator. The aim of the cloud administrator is to deploy a web application on one of the virtual machines. We introduced a game-theoretic aspect to the case study by considering a hostile behaviour of the environment that causes servers to fail due to hardware failures. One of the considered properties is $\mathtt{P}_{\geqslant 0.9999}[\,\mathtt{F}\ \mathit{deployed}]$, which ensures that the web application is deployed with high probability.

investor([100])

The investor case study models an investor buying futures contracts. The aim of the investor is to maximise his/her profit while the environment will try to decrease it. The investor is able to decide whether to invest or not at specific points in time. The environment can bar the investor from investing for a fixed period of time. The property used is $\mathtt{R}^{\mathit{profit}}_{\geqslant 4.98}\,[\,\mathtt{C}\,]$, which ensures a return of at least 90% of the maximal return.

team-form([33, 64]

This case study presented in [33] performs an analysis of a team formation protocol of [64]. The protocol governs how a set of agents in a distributed environment should assemble into teams such that they can perform a given set of tasks together. We consider a property

that is used to compute a strategy that ensures that the first team completes their task with high probability, namely, $\mathtt{P}_{\geqslant 0.9999}[\,\mathtt{F}\ \mathit{done}_1\,]$.

cdmsn([30, 109])

Similarly to the previous case study, here we work with a set of agents in a distributed environment. Saffre et al. [109] present a framework that facilitates a decision making process, where a set of agents need to agree on a particular decision. For this case study, we are interested in generating a strategy that ensures that agents agree on a decision with high probability. Such a property can be described using $\mathtt{P}_{\geqslant 0.9999}[\,\mathtt{F}\ \mathit{prefer}_1]$.

Appendix B

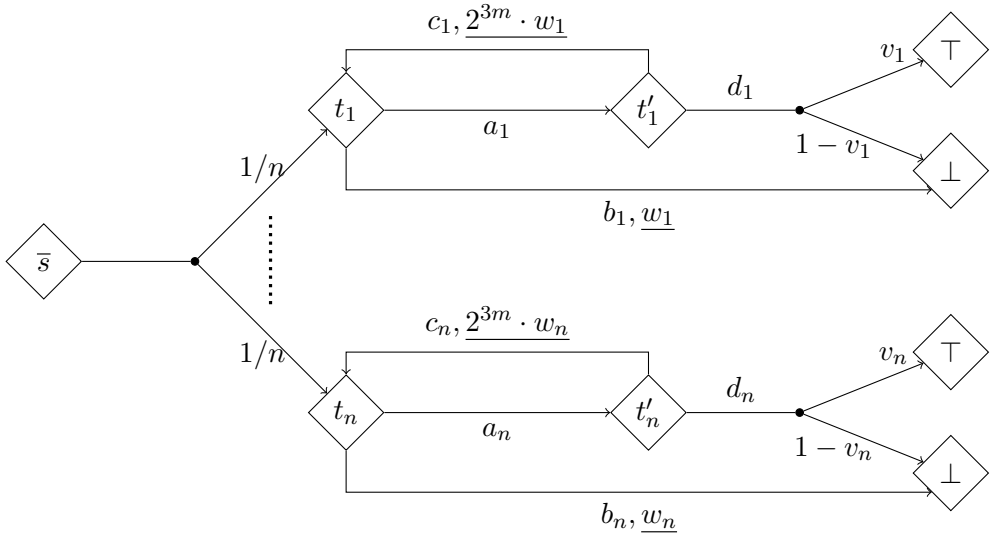
Proofs for Chapter 6

B.1 Proof of Theorem 4

Proof. We start with the case of randomised multi-strategies and static penalties, which is the most delicate part of the proof. Then we analyse the case of randomised multi-strategies and dynamic penalties, and finally show that this case can easily be modified for the remaining two combinations.

Randomised multi-strategies and static penalties. We give a reduction from the Knapsack problem. Let n be the number of items, each of which can either be or not be put in a knapsack, let v_i and w_i be the value and the weight of item i, respectively, and let V and W be the bounds on the value and weight of the items to be picked. We assume that $v_i \leqslant 1$ for every $1 \leqslant i \leqslant n$, and that all numbers v_i and w_i are given as fractions with denominator q.

Let us construct the following MDP, where m is chosen such that $n2^{-m} \leqslant \frac{1}{q}$ and $2^{-m} \cdot W \leqslant \frac{1}{q}$.



We define a reward structure r such that every path reaching $\top$ is assigned cumulative reward 1. The penalties are as given by the underlined expressions.

We show that there is a multi-strategy θ sound for the property $\mathtt{R}^r_{\geqslant V/n}[\,\mathtt{C}\,]$ such that $pen_{sta}(\psi, \theta) \leqslant W + 2^{-m} \cdot W$ if and only if the answer to the Knapsack problem is "yes".

In the direction $\Leftarrow$, let $I \subseteq \{1, \ldots, n\}$ be the set of items put in the knapsack. It suffices to define the multi-strategy θ by:

- $\theta(t'_i)(\{c_i, d_i\}) = 1 - 2^{-4m}$, $\theta(t'_i)(\{d_i\}) = 2^{-4m}$, $\theta(t_i)(\{a_i\}) = 1$ for $i \in I$
- $\theta(t'_i)(\{c_i, d_i\}) = 1$, $\theta(t_1)(\{a_i, b_i\}) = 1$ for $i \notin I$.

In the direction $\Rightarrow$, let us assume we have a multi-strategy θ satisfying the assumptions. Let $P(s \to s')$ denote the lower bound on the probability of reaching s' from s under a strategy which complies with the multi-strategy θ. Denote by $I \subseteq \{1, \ldots, n\}$ the indices i such that $P(t_i \to \top) \geqslant 2^{-m}$.

Let $\beta_i = \theta(t_i)(\{a_i\})$ and $\alpha_i = \theta(t'_1)(\{d_i\})$. Observe that:

$$y_i = \beta_i \cdot \sum_{j=0}^{\infty} ((1 - \alpha_i) \cdot \beta_i)^j \cdot \alpha_i \cdot v_i = \frac{\alpha_i \beta_i v_i}{1 - (1 - \alpha_i)\beta_i} = \frac{\alpha_i \beta_i v_i}{1 - \beta_i + \alpha_i \beta_i}$$

because the optimal strategy σ will pick b_i and c_i whenever they are available. Note that for $i \in I$, $\alpha_i \geqslant 2^{-m}(1 - \beta_i)$, since otherwise we have:

$$\frac{\alpha_i \beta_i v_i}{1 - \beta_i + \alpha_i \beta_i} < \frac{\alpha_i \beta_i}{1 - \beta_i + \alpha_i \beta_i} < \frac{2^{-m}(1 - \beta_i)\beta_i}{1 - \beta_i + 2^{-m}(1 - \beta_i)\beta_i} < \frac{2^{-m}\beta_i}{1 + 2^{-m}\beta_i} \leqslant 2^{-m}$$

Hence, $\alpha_i \geqslant 2^{-m}(1-\beta_i)$, and so:

$$pen_{loc}(\psi,\theta,t_i) + pen_{loc}(\psi,\theta,t_i') = \beta_i w_i + \alpha_i 2^{3m} w_i \geqslant \beta_i w_i + 2^{-m}(1-\beta_i)2^{3m} w_i \geqslant w_i$$

We have:

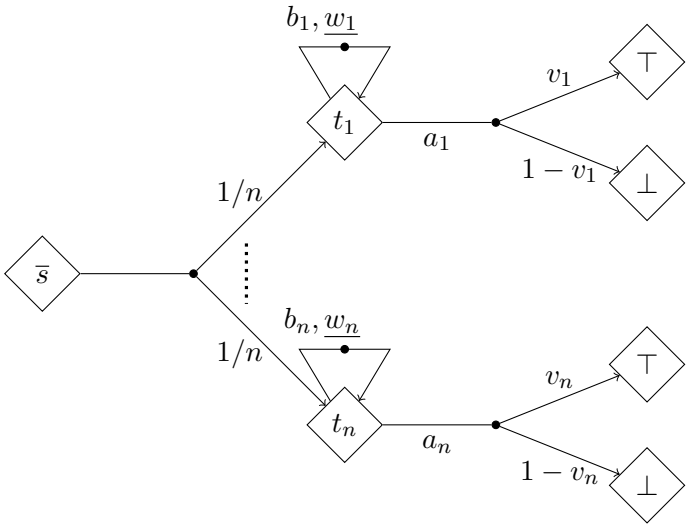
$$\sum_{i\in I} w_i \leqslant \sum_{i\in I} \big(pen_{loc}(\psi,\theta,t_i) + pen_{loc}(\psi,\theta,t_i')\big) \leqslant W + 2^{-m}\cdot W$$

and because $\sum_{i\in I} w_i$ and W are fractions with denominator q, by the choice of m, we can infer that $\sum_{i\in I} w_i \leqslant W$. Similarly:

$$\sum_{i\in I} \frac{1}{n} v_i \geqslant \sum_{i\in I} \frac{1}{n} P(t_i \to \top) \geqslant \Big(\frac{1}{n}\sum_{i=1}^{n} P(t_i \to \top)\Big) - \frac{1}{n}2^{-m}n \geqslant \frac{1}{n}V - 2^{-m}$$

and again, because $\sum_{i\in I} v_i$ and V are fractions with denominator q, by the choice of m we can infer that $\sum_{i\in I} v_i \geqslant V$. Hence, in the instance of the knapsack problem it suffices to pick exactly items from I to satisfy the restrictions.

Randomised multi-strategies with dynamic penalties. The proof is analogous to the proof above; we only need to modify the MDP and the computations. For an instance of the Knapsack problem given as before, we construct the following MDP:



We claim that there is a multi-strategy θ sound for the property $\mathtt{R}^{r}_{\geqslant V/n}[\,\mathtt{C}\,]$ such that $pen_{dyn}(\psi,\theta) \leqslant \frac{1}{n}W$ if and only if the answer to the Knapsack problem is "yes".

In the direction $\Leftarrow$, for $I \subseteq \{1,\ldots,n\}$ the set of items in the knapsack we define θ by $\theta(t_i)(\{a_i\}) = 1$ for $i \in I$ and by allowing all actions in every other state.

In the direction $\Rightarrow$, let us assume we have a multi-strategy θ satisfying the assumptions.

Let $P(s \to s')$ denote the lower bound on the probability of reaching s' from s under a strategy which complies with the multi-strategy θ. Denote $I \subseteq \{1, \ldots, n\}$ the indices i such that $\theta(t_i)(\{a_i\}) > 0$. Observe that $P(t_i \to \top) = v_i$ if $i \in I$ and $P(t_i \to \top) = 0$ otherwise. Hence:

$$\sum_{i \in I} \frac{1}{n} v_i = \sum_{i \in I} \frac{1}{n} P(t_i \to \top) = \frac{1}{n} \sum_{i=1}^{n} P(t_i \to \top) \geqslant \frac{1}{n} V$$

and for the penalty, denoting $x_i := \theta(t_i)(\{a_i\})$, we get:

$$\frac{1}{n} W \geqslant pen_{dyn}(\psi, \theta) = \frac{1}{n} \sum_{i=0}^{n} \sum_{j=0}^{\infty} (1 - x_i)^j x_i w_i = \frac{1}{n} \sum_{i \in I} \sum_{j=0}^{\infty} (1 - x_i)^j x_i w_i = \frac{1}{n} \sum_{i \in I} w_i \quad \text{(B.1)}$$

because the strategy that maximises the penalty will pick b_i whenever it is available. Hence, in the instance of the knapsack problem it suffices to pick exactly items from I to satisfy the restrictions.

Deterministic multi-strategies and dynamic penalties. The proof is identical to the proof for randomised multi-strategies and dynamic penalties above: observe that the multi-strategy constructed there from an instance of Knapsack is in fact deterministic.

Deterministic multi-strategies and static penalties. The proof is obtained by a small modification of the proof for randomised multi-strategies and dynamic penalties above. Instead of requiring $pen_{dyn}(\psi, \theta) \leqslant \frac{1}{n} W$ we require $pen_{sta}(\psi, \theta) \leqslant W$, and Equation B.1 changes to:

$$W \geqslant pen_{sta}(\psi, \theta) = \sum_{i=0}^{n} x_i w_i = \sum_{i \in I} w_i \,.$$

□

B.2 Proof of Theorem 5

Proof. We consider the deterministic and randomised cases separately.

Deterministic multi-strategies. We start by showing NP membership for deterministic multi-strategies. If the answer to the problem is "yes", then there is a witnessing deterministic multi-strategy, which is of polynomial size. We can guess such a strategy nondeterministically and then in polynomial time verify that the guess is correct. The fact that the multi-strategy is sound and that it achieves the required dynamic penalty can be verified using standard algorithms for computing expected total reward in MDPs. Static penalties can be checked by summing up the local penalties.

Randomised multi-strategies. Now we show that the permissive controller synthesis problem is in PSPACE if we restrict to randomised multi-strategies and static penalties. For dynamic penalties the proof is similar.

The proof proceeds by constructing a polynomial-size closed formula Ψ of the existential fragment of $(\mathbb{R}_{\geqslant 0}, +, \cdot, \leqslant)$ such that Ψ is true if and only if there is a multi-strategy ensuring the required penalty and reward. Because determining the validity of a closed formula of the existential fragment of $(\mathbb{R}_{\geqslant 0}, +, \cdot, \leqslant)$ is in PSPACE [27], we obtain the desired result.

For the rest of this section, fix an instance of the permissive controller problem as in Definition 8, with static penalties. We say that a multi-strategy is *winning* if it satisfies the conditions on θ in Definition 8.

For numbers $\vec{p} = (p_s)_{s \in S}$, where $0 \leqslant p_s \leqslant 1$ for every $s \in S$, let us consider a game $\mathsf{G}_{\vec{p}}$ which is obtained from G by applying the transformation from Section 6.3.3 for approximating randomised multi-strategies (see also Figure 6.5), where we fix $n = 2$ and substitute the numbers p_1 and p_2 in the gadget created for s with numbers p_s and $1 - p_s$. We claim that there is a randomised winning multi-strategy in G if and only if there exists a vector $\vec{p}$ such that there is a deterministic winning multi-strategy in $\mathsf{G}_{\vec{p}}$. The proof proceeds by establishing a direct correspondence between randomised multi-strategies in G and games $\mathsf{G}_{\vec{p}}$ and deterministic multi-strategies in them.

Further, let $\Psi[\mathsf{G}_{\vec{p}}]$ denote the conjunction of the constraints 6.1-6.8 from Figure 6.2 for the game $\mathsf{G}_{\vec{p}}$, together with the constraints:

$$\sum_{s \in S_\Diamond} \Big(p_s \cdot \big(\alpha(s,1,1) + \alpha(s,1,2)\big) + (1 - p_s) \cdot \big(\alpha(s,2,1) + \alpha(s,2,2)\big) \Big) \leqslant c$$

$$y_{s_i,b_j} \cdot \sum_{a \in A(s'_j)} (1 - y_{s'_j,a}) \cdot \psi(s'_j, a) = \alpha(s,i,j) \quad \text{for all } s \in S,\ i,j \in \{1,2\}$$

$$0 \leqslant p_s \leqslant 1 \quad \text{for all } s \in S$$

We get that $\Psi[\mathsf{G}_{\vec{p}}]$ is satisfiable if and only if there is a deterministic winning multi-strategy.

Note that the formulae $\Psi[\mathsf{G}_{\vec{p}}]$ for different $\vec{p}$ differ only at positions where the numbers of $\vec{p}$ are substituted. Hence, we can create a formula Ψ' which is obtained from $\Psi[G_{\vec{p}}]$, where each p_s is treated as a variable. From the above we get that the formula Ψ' is satisfiable if and only if there is a randomised winning multi-strategy, and hence we finish the proof by putting $\Psi \equiv \exists \vec{x} \Psi'$ where $\vec{x}$ are all variables of Ψ'. □

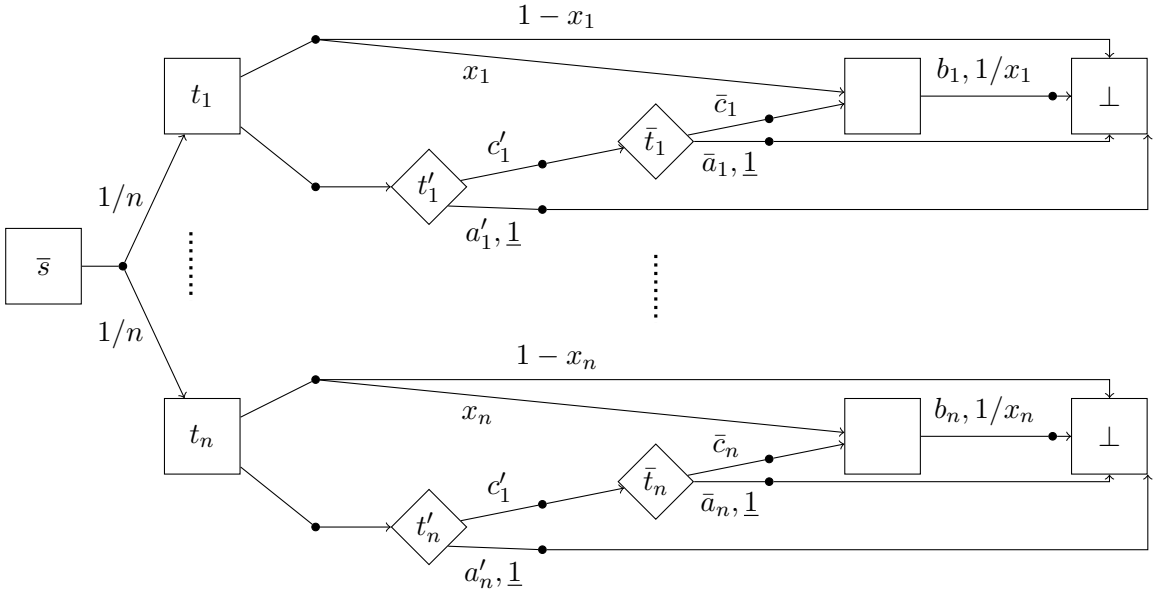


Figure B.1: The game for the proof of Theorem 6.

B.3 Proof of Theorem 6

Proof. Let $x_1,\ldots,x_n$ and y be numbers giving the instance of the square-root-sum problem, i.e. we aim to determine whether $\sum_{i=1}^{n} \sqrt{x_i} \leqslant y$. We construct the game from Figure B.1.

The penalties are as given by the underlined numbers, and the rewards $1/x_i$ are awarded under the actions b_i.

Static penalties. We first give the proof for static penalties. We claim that there is a multi-strategy θ sound for the property $\mathtt{R}^{r}_{\geqslant 1}[\,\mathtt{C}\,]$ such that $pen_{sta}(\psi, \theta) \leqslant y$ if and only if $\sum_{i=1}^{n} \sqrt{x_i} \leqslant y$.

In the direction $\Leftarrow$, let us define a multi-strategy θ by $\theta(t'_i)(\{c'_i\}) = \theta(\bar{t}_i)(\{\bar{c}_i\}) = \sqrt{x_i}$ and $\theta(t'_i)(\{a'_i, c'_i\}) = \theta(\bar{t}_i)(\{\bar{a}_i, \bar{c}_i\}) = 1 - \sqrt{x_i}$, and allowing all actions in all remaining states. We then have $pen_{sta}(\psi, \theta) = \sum_{i=1}^{n} 2 \cdot \sqrt{x_i}$ and the reward achieved is:

$$\frac{1}{n} \sum_{i=1}^{n} \min\{x_i \cdot \frac{1}{x_i}, \sqrt{x_i} \cdot \sqrt{x_i} \frac{1}{x_i}\} \quad = \quad 1.$$

In the direction $\Rightarrow$, let θ be an arbitrary multi-strategy sound for the property $\mathtt{R}^{r}_{\geqslant 1}[\,\mathtt{C}\,]$ satisfying $pen_{sta}(\psi, \theta) \leqslant 2 \cdot y$. Let $z'_i = \theta(t'_i)(\{c'_i\})$ and $\bar{z}_i = \theta(\bar{t}_i)(\{\bar{c}_i\})$. The reward achieved is:

$$\frac{1}{n} \sum_{i=1}^{n} \min\{x_i \cdot \frac{1}{x_i}, z'_i \cdot \bar{z}_i \frac{1}{x_i}\} = \frac{1}{n} \sum_{i=1}^{n} \min\{1, z'_i \cdot \bar{z}_i \frac{1}{x_i}\}$$

which is greater or equal to 1 if and only if $z'_i \cdot \bar{z}_i \geqslant a_i$ for every i. We show that $z'_i + \bar{z}_i \geqslant$

$2 \cdot \sqrt{x_i}$: If both z'_i and $\bar{z}_i$ are greater than $\sqrt{x_i}$, we are done. The case $z'_i, \bar{z}_i \leqslant \sqrt{x_i}$ cannot take place. As for the remaining case, w.l.o.g., suppose that $z'_i = \sqrt{x_i} + p$ and $\bar{z}_i = \sqrt{x_i} - q$ for some non-negative p and q. Then $(\sqrt{x_i} + p) \cdot (\sqrt{x_i} - q) = x_i + (p - q)\sqrt{x_i} - pq$, and for this to be at least x_i we necessarily have $p \geqslant q$, and so $z'_i + \bar{z}_i = \sqrt{x_i} + p + \sqrt{x_i} - q \geqslant 2 \cdot \sqrt{x_i}$. Hence, we get that:

$$\sum_{i=1}^{n} 2 \cdot \sqrt{x_i} \leqslant \sum_{i=1}^{n} \left(z'_i + \bar{z}_i\right) = pen_{sta}(\psi, \theta) \leqslant 2 \cdot y.$$

Dynamic penalties. We now proceed with dynamic penalties, where the analysis is similar. Let us use the same game as before, but in addition assume that the penalty assigned to actions c'_i and $\bar{c}'_i$ is equal to 1. We claim that there is a multi-strategy θ sound for the property $\mathtt{R}^{r}_{\geqslant 1}[\,\mathtt{C}\,]$ such that $pen_{dyn}(\psi, \theta) \leqslant 2 \cdot y/n$ if and only if $\sum_{i=1}^{n} \sqrt{x_i} \leqslant y$.

In the direction $\Leftarrow$, let us define a multi-strategy θ as before, and obtain $pen_{dyn}(\psi, \theta) = \frac{1}{n} \sum_{i=1}^{n} 2 \cdot \sqrt{y_i}$.

In the direction $\Rightarrow$, let θ be an arbitrary multi-strategy sound for the property $\mathtt{R}^{r}_{\geqslant 1}[\,\mathtt{C}\,]$ satisfying $pen_{dyn}(\psi, \theta) \leqslant 2 \cdot y/n$. Let $z'_i = \theta(t'_i)(\{c'_i\})$, $\bar{z}_i = \theta(\bar{t}_i)(\{\bar{c}_i\})$, $u'_i = \theta(t'_i)(\{a'_i\})$, and $\bar{u}_i = \theta(\bar{t}_i)(\{\bar{a}_i\})$.

As before we can show that $z'_i + \bar{z}_i \geqslant 2 \cdot \sqrt{x_i}$, and so:

$$\frac{1}{n} \sum_{i=1}^{n} 2 \cdot \sqrt{x_i} \leqslant \frac{1}{n} \sum_{i=1}^{n} \left(z'_i + \bar{z}_i\right) \leqslant \frac{1}{n} \sum_{i=1}^{n} \left((z'_i + u'_i) + (1 - u'_i) \cdot (\bar{z}_i + \bar{u}_i)\right)$$
$$= pen_{dyn}(\psi, \theta) \leqslant 2 \cdot y/n.$$

□

B.4 Proof of Theorem 7

Before proving the correctness of the encoding (stated in Theorem 7), we prove the following auxiliary lemma that characterises the reward achieved under a multi-strategy in terms of a solution of a set of inequalities.

Lemma 5. *Let* $\mathsf{G} = \langle S_\Diamond, S_\Box, S, \overline{s}, A, \delta, \mathcal{L} \rangle$ *be a stochastic game,* $\phi = \mathtt{R}^{r}_{\geqslant b}[\,\mathtt{C}\,]$ *a property,* (ψ, sta) *a* static *penalty scheme and* θ *a deterministic multi-strategy. Consider the system of inequalities* $x_s \leqslant \min_{a \in \theta(s)} \sum_{s' \in S} \delta(s, a)(s') x_{s'} + r(s, a)$ *for* $s \in S$*. Then the following holds:*

- $\bar{x}_s = \inf_{\sigma \lhd \theta, \pi} E^{\sigma,\pi}_{\mathsf{G},s}(r)$ *is a solution to the above inequalities.*

- *Suppose a solution $\bar{x}_s$ to the above inequalities satisfies that whenever $\bar{x}_s > 0$, for every $\sigma \triangleleft \theta$ and every π there is a path $\omega = s_0 a_0 \dots s_n a_n$ starting in s and satisfying $Pr^{\sigma,\pi}_{\mathsf{G},s}(\omega) > 0$ and $r(s_n, a_n) > 0$. Then $\bar{x}_s \leqslant \inf_{\sigma \triangleleft \theta, \pi} E^{\sigma,\pi}_{\mathsf{G},s}(r)$ for all s.*

Proof. The game G together with θ determines a Markov decision process $\mathsf{G}^\theta = \langle \emptyset, S_\Diamond \cup S_\Box, \overline{s}, A, \delta', \mathcal{L} \rangle$, in which the choices disallowed by θ are removed, i.e. $\delta'(s,a) = \delta(s,a)$ for every $s \in S_\Box$ and every $s \in S_\Diamond$ with $a \in \theta(s)$, and $\delta'(s,a)$ is undefined otherwise. We have:

$$\inf_{\sigma \triangleleft \theta, \pi} E^{\sigma,\pi}_{\mathsf{G},s}(r) = \inf_{\bar{\sigma}} E^{\bar{\sigma}}_{\mathsf{G}^\theta,s}(r)$$

since, for any strategy pair $\sigma \triangleleft \theta$ and π in G, there is a strategy $\bar{\sigma}$ in G^θ which is defined, for every finite path ω of G^θ ending in t, by $\bar{\sigma}(\omega) = \sigma(\omega)$ or $\bar{\sigma}(\omega) = \pi(\omega)$, depending on whether $t \in S_\Diamond$ or $t \in S_\Box$, and which satisfies $E^{\sigma,\pi}_{\mathsf{G},s}(r) = E^{\sigma}_{\mathsf{G}^\theta,s}(r)$. Similarly, a strategy $\bar{\sigma}$ for G^θ induces a compliant strategy σ and a strategy π defined for every finite path ω of G ending in $S_\Diamond$ (resp. $S_\Box$) by $\sigma(\omega) = \bar{\sigma}(\omega)$ (resp. $\pi(\omega) = \bar{\sigma}(\omega)$).

The rest is then the following simple application of results from the theory of Markov decision processes. The first item of the lemma follows from [107, Lemma 7.1.3]. For the second part of the lemma, observe that if, $\inf_{\sigma \triangleleft \theta, \pi} E^{\sigma,\pi}_{\mathsf{G},s}(r)$ is infinite, then the claim holds trivially. Otherwise, from the assumption on the existence of ω we have that, under any compliant strategy, there is a path $\omega' = s_0 a_0 s_1 \dots s_n$ of length at most $|S|$ in G^θ such that $\inf_{\sigma \triangleleft \theta, \pi} E^{\sigma,\pi}_{\mathsf{G},s_n}(r) = 0$ (otherwise the reward would be infinite) and so $\bar{x}_{s_n} = 0$. We can thus apply [107, Proposition 7.3.4]. □

Proof. We prove that every multi-strategy θ induces a satisfying assignment to the variables such that the static penalty under θ is $\sum_{s \in S_\Diamond} \sum_{a \in A(s)} (1 - y_{s,a}) \cdot \psi(s,a)$, and vice versa. The theorem then follows from the rescaling of rewards and penalties that we performed.

We start by proving that, given a sound multi-strategy θ, we can construct a satisfying assignment $\{\bar{y}_{s,a}, \bar{x}_s, \bar{\alpha}_s, \bar{\beta}_{s,a,t}, \bar{\gamma}_t\}_{s,t \in S, a \in A}$ to the constraints from Figure 6.2. For $s \in S_\Diamond$ and $a \in A(s)$, we set $\bar{y}_{s,a} = 1$ if $a \in \theta(s)$, and otherwise we set $\bar{y}_{s,a} = 0$. This gives satisfaction of contraint (6.2). For $s \in S_\Box$ and $a \in A(s)$, we set $\bar{y}_{s,a} = 1$, ensuring satisfaction of (6.7). We then put $\bar{x}_s = \inf_{\sigma \triangleleft \theta, \pi} E^{\sigma,\pi}_{\mathsf{G},s}(r)$. By the first part of Lemma 5 we get that constraints (6.1), (6.3) (for $a \in \theta(s)$) and (6.4) are satisfied. Constraint (6.3) for $a \notin \theta(s)$ is satisfied because in this case $\bar{y}_{s,a} = 0$, and so the right-hand side is always at least 1.

We further set $\bar{\alpha}_s = 1$ if $x_s > 0$ and $\bar{\alpha}_s = 0$ if $x_s = 0$, thus satisfying constraint (6.5). Let d_s be the maximum distance to a positive reward, i.e., d is defined inductively by

putting $d_s = 0$ if we have $r(s,a) > 0$ for all $a \in A(s)$, and otherwise:

$$d_s = 1 + \min_{a \in \theta(s), r(s,a)=0} \max_{\delta(s,a)(t)>0} d_t.$$

Put $d_s = \bot$ if d_s was not defined by the above. For s such that $d_s \neq \bot$, we put $\bar{\gamma}_s = d_s/|S|$, and for every a we choose t such that $d_t < d_s$, and set $\bar{\beta}_{s,a,t} = 1$, leaving $\bar{\beta}_{s,a,t} = 0$ for all other t. For s such that $d_s = \bot$ we define $\bar{\gamma}_s = 0$ and for all a and t put $\bar{\beta}_{s,a,t} = 0$. This ensures the satisfaction of the remaining constraints.

In the opposite direction, assume that we are given a satisfying assignment. Firstly, we create a game G' from G by making any states s with $\bar{x}_s = 0$ terminal. Any sound multi-strategy in G' directly gives a sound multi-strategy in G.

We construct θ for G' by putting $\theta(s) = \{a \in A(s) \mid \bar{y}_{s,a} = 1\}$ for all $s \in S_\Diamond$ with $\bar{x}_s > 0$. First, by the choice of the objective function to minimise, the multi-strategy yields the penalty $\sum_{s \in S_\Diamond} \sum_{a \in A(s)} (1 - \bar{y}_{s,a}) \cdot \psi(s,a)$. Next, we will show that θ satisfies the assumption of the second part of Lemma 5, from which we get that:

$$\inf_{\sigma \lhd \theta, \pi} E^{\sigma,\pi}_{\mathsf{G}',s}(r) \geqslant \bar{x}_s$$

which, together with constraint (6.1) being satisfied, gives us the desired result.

Consider any s such that $\inf_{\sigma \lhd \theta, \pi} E^{\sigma,\pi}_{\mathsf{G}',s}(r) > 0$. Then we have $\bar{x}_s > 0$ (by the definition of G'). Let us fix any $\sigma \lhd \theta$ and any π. We show that there is a path ω satisfying the assumption of the lemma. We build $\omega = s_1 \dots s_n a_n$ inductively, to satisfy: (i) $r(s_n, a_n) > 0$, (ii) $\bar{x}_{s_i} \geqslant \bar{x}_{s_{i-1}}$ for all i, and (iii) for any sub-path $s_i a_i \dots s_j$ with $\bar{x}_{s_i} = \bar{x}_{s_j}$ we have that $\bar{\gamma}_{s_k} < \bar{\gamma}_{s_{k-1}}$ for all $i+1 \leqslant k \leqslant j$.

We set $s_0 = s$. Assume we have defined a prefix $s_0 a_0 \dots s_i$ to satisfy conditions (ii) and (iii). We put a_i to be the action picked by σ (or π) in s_i. If $r(s_i, a_i) > 0$, we are done.

Otherwise, we pick s_{i+1} as follows:

- If there is $s' \in \mathit{supp}(\delta(s_i, a_i))$ with $\bar{x}_{s'} > \bar{x}_s$, then we put $s_{i+1} = s'$. Such a choice again satisfies (ii) and (iii) by definition.

- If we have $\bar{x}_{s'} = \bar{x}_s$ for all $s' \in \mathit{supp}(\delta(s_i, a_i))$, then any choice will satisfy (ii). To satisfy the other conditions, we pick s_{i+1} so that $\bar{\beta}_{s_i,a_i,s_{i+1}} = 1$ is true. We argue that such an s_{i+1} can be chosen. We have $\bar{x}_{s_i} > 0$ and so $\bar{\alpha}_s = 1$ by constraint (6.5). We also have $\bar{y}_{s,a} = 1$: for $s \in S_\Diamond$ this follows from the definition of θ, for $s \in S_\Box$ from constraint (6.7). Hence, since constraint (6.6) is satisfied, there must be s_{i+1} such that $\bar{\beta}_{s_i,a,s_{i+1}} = 1$. Then, we apply constraint (6.8) (for $s = s_i$, $t = s_{i+1}$ and $a = a_i$) and, since the last two summands on the right hand side are 0, we get $\bar{\gamma}_{s_{i+1}} < \bar{\gamma}_{s_i}$, thus satisfying (iii).

Note that the above construction must terminate after at most $|S|$ steps since, due to conditions (ii) and (iii), no state repeats on ω. Because the only way of terminating is satisfaction of (i), we are done. □

B.5 Proof of Theorem 8

Proof. We show that any sound multi-strategy with finite penalty $\bar{z}_{\bar{s}}$ gives rise to a satisfying assignment with the objective value $\bar{z}_{\bar{s}}$, and vice versa. Then, (b) follows directly, and (a) follows by the assumption that there is *some* sound multi-strategy.

Let us prove that for any sound multi-strategy θ we can construct a satisfying assignment to the constraints. For constraints (6.1) to (6.8), the construction works exactly the same as in the proof of Theorem 7. For the newly added variables, i.e. z_s and ℓ_s, we put $\bar{\ell}_s = pen_{loc}(\psi, \theta, s)$, ensuring satisfaction of constraint (6.9), and:

$$\bar{z}_s = \sup_{\sigma \triangleleft \theta, \pi} E^{\sigma,\pi}_{\mathsf{G},s}(\psi^{\theta}_{rew})$$

which, together with [107, Section 7.2.7], ensures that constraints (6.10) and (6.11) are satisfied.

In the opposite direction, given a satisfying assignment we construct θ for G' exactly as in the proof of Theorem 7. As before, we can argue that constraints (6.1) to (6.8) are satisfied under any sound multi-strategy. We now need to argue that the multi-strategy satisfies $pen_{dyn}(\psi, \theta, s) \geqslant \bar{z}_{\bar{s}}$. It is easy to see that $pen_{loc}(\psi, \theta, s) = \bar{\ell}_s$. Moreover, by [107, Section 7.2.7] the penalty is the least solution to the inequalities:

$$z'_s \geqslant \max_{a \in \theta(s)} \sum_{s' \in S} \delta(s,a)(s) \cdot z'_{s'} + \bar{\ell}_s \qquad \text{for all } s \in S_{\Diamond} \tag{B.2}$$

$$z'_s \geqslant \max_{a \in A(s)} \sum_{s' \in S} \delta(s,a)(s) \cdot z'_{s'} \qquad \text{for all } s \in S_{\Box}. \tag{B.3}$$

We can replace (B.2) with:

$$z'_s \geqslant \max_{a \in A(s)} \sum_{s' \in S} \delta(s,a)(s) \cdot z'_{s'} + \bar{\ell}_s - c \cdot (1 - \bar{y}_{s,a}) \tag{B.4}$$

since for $a \in \theta(s)$ we have $c \cdot (1 - \bar{y}_{s,a}) = 0$ and otherwise $c \cdot (1 - \bar{y}_{s,a})$ is greater than $\sum_{s' \in S} \delta(s,a)(s) \cdot z'_{s'} + \bar{\ell}_s$ in the least solution to (B.2) and (B.3), by the definition of c. Finally, it suffices to observe that the set of solutions to (B.3) and (B.4) is the same as the set of solutions to (6.10) and (6.11). □

B.6 Proof of Theorem 9

Proof. In our proofs, we use $E^{\sigma,\pi}_{\mathsf{G},\overline{s}}(r{\downarrow}s)$ for the expected total reward accumulated before the first visit to s, defined by:

$$E^{\sigma,\pi}_{\mathsf{G},\overline{s}}(r{\downarrow}s) = \int_{\omega=s_0a_0s_1a_1\ldots\in IPath_{\overline{s}}} \sum_{i=0}^{fst(s,\omega)-1} r(s_i,a_i)\, dPr^{\sigma,\pi}_{\mathsf{G},\overline{s}}$$

where $fst(s,\omega)$ is $\min\{i \mid s_i = s\}$ if $\omega = s_0a_0s_1a_1\ldots$ contains s_i, and ∞ otherwise.

If the (dynamic) penalty under θ is infinite, then the solution is straightforward: we can simply take θ' which, in every state, allows a single action so that the reward is maximised. Hence, we will assume that the penalty is finite.

Let θ be a multi-strategy allowing $n > 2$ different sets $A_1,\ldots,A_n$ with non-zero probabilities $\lambda_1,\ldots,\lambda_n$ in $s_1 \in S_\Diamond$. We construct a multi-strategy θ' that in s_1 allows only two of the sets $A_1,\ldots,A_n$ with non-zero probability, and in other states behaves like θ.

We first prove the case of dynamic penalties and then describe the differences for static penalties. Supposing that $\inf_{\sigma\lhd\theta,\pi} E^{\sigma,\pi}_{\mathsf{G},s_1}(r) \leqslant \inf_{\sigma\lhd\theta',\pi} E^{\sigma,\pi}_{\mathsf{G},s_1}(r)$, we have that the total reward is:

$$\begin{aligned}
\inf_{\sigma\lhd\theta,\pi} E^{\sigma,\pi}_{\mathsf{G},\overline{s}}(r) &= \inf_{\sigma\lhd\theta,\pi} \Big(E^{\sigma,\pi}_{\mathsf{G},\overline{s}}(r{\downarrow}s_1) + Pr^{\sigma,\pi}_{\mathsf{G},\overline{s}}(\mathtt{F}\ s_1)\cdot E^{\sigma,\pi}_{\mathsf{G},s_1}(r)\Big)\\
&= \inf_{\sigma\lhd\theta,\pi} \Big(E^{\sigma,\pi}_{\mathsf{G},\overline{s}}(r{\downarrow}s_1) + Pr^{\sigma,\pi}_{\mathsf{G},\overline{s}}(\mathtt{F}\ s_1)\cdot \inf_{\sigma'\lhd\theta,\pi'} E^{\sigma',\pi'}_{\mathsf{G},s_1}(r)\Big)\\
&\leqslant \inf_{\sigma\lhd\theta,\pi} \Big(E^{\sigma,\pi}_{\mathsf{G},\overline{s}}(r{\downarrow}s_1) + Pr^{\sigma,\pi}_{\mathsf{G},\overline{s}}(\mathtt{F}\ s_1)\cdot \inf_{\sigma'\lhd\theta',\pi'} E^{\sigma',\pi'}_{\mathsf{G},s_1}(r)\Big)\\
&= \inf_{\sigma\lhd\theta',\pi} \Big(E^{\sigma,\pi}_{\mathsf{G},\overline{s}}(r{\downarrow}s_1) + Pr^{\sigma,\pi}_{\mathsf{G},\overline{s}}(\mathtt{F}\ s_1)\cdot \inf_{\sigma'\lhd\theta',\pi'} E^{\sigma',\pi'}_{\mathsf{G},s_1}(r)\Big)\\
&= \inf_{\sigma\lhd\theta',\pi} E^{\sigma,\pi}_{\mathsf{G},\overline{s}}(r).
\end{aligned}$$

Hence, it suffices to define θ' so that $\inf_{\sigma\lhd\theta,\pi} E^{\sigma,\pi}_{\mathsf{G},s_1}(r) \leqslant \inf_{\sigma\lhd\theta',\pi} E^{\sigma,\pi}_{\mathsf{G},s_1}(r)$. Similarly, for the penalties, it is enough to ensure $\sup_{\sigma\lhd\theta,\pi} E^{\sigma,\pi}_{\mathsf{G},s_1}(\psi^\theta_{rew}) \geqslant \sup_{\sigma\lhd\theta',\pi} E^{\sigma,\pi}_{\mathsf{G},s_1}(\psi^{\theta'}_{rew})$.

Let P_i and R_i, where $i \in \{1,...,n\}$, be the penalties and rewards from θ after allowing A_i against an optimal opponent strategy, i.e.:

$$\begin{aligned}
P_i &= \sum_{a\notin A_i} \psi(s,a) + \sup_{\sigma\lhd\theta,\pi} \max_{a\in A_i} \sum_{s'\in S} \delta(s_1,a)(s')\cdot E^{\sigma,\pi}_{\mathsf{G},s'}(\psi^\theta_{rew})\\
R_i &= \inf_{\sigma\lhd\theta,\pi} \min_{a\in A_i} (r(s,a) + \sum_{s'\in S} \delta(s_1,a)(s')\cdot E^{\sigma,\pi}_{\mathsf{G},s'}(r)).
\end{aligned}$$

We also define $R = \inf_{\sigma\lhd\theta,\pi} E^{\sigma,\pi}_{\mathsf{G},s_1}(r)$ and $P = \sup_{\sigma\lhd\theta,\pi} E^{\sigma,\pi}_{\mathsf{G},s_1}(\psi^\theta_{rew})$ and have $R = \sum_{i=1}^n \lambda_i R_i$ and $P = \sum_{i=1}^n \lambda_i P_i$.

Let $S_0 \subseteq S$ be those states for which there are $\sigma \triangleleft \theta$ and π ensuring a return to s_1 without accumulating any reward. We say that A_i is *progressing* if for all $a \in A_i$ we have $r(s_1, a) > 0$ or $supp(\delta(s_1, a)) \cap S_0 = \emptyset$. We note that A_i is progressing whenever $r_i > r_\lambda$ (since any a violating the condition above could have been used by the opponent to force $r_i \leqslant r_\lambda$).

For each tuple $\mu = (\mu_1, \ldots, \mu_n) \in \mathbb{R}^n$, let $R^\mu = \mu_1 R_1 + \cdots + \mu_n R_n$ and $P^\mu = \mu_1 P_1 + \cdots + \mu_n P_n$. Then the set $T = \{(R^\mu, P^\mu) \mid 0 \leqslant \mu_i \leqslant 1, \mu_1 + \cdots + \mu_n = 1\}$ is a bounded convex polygon, with vertices given by images (R^{e_i}, P^{e_i}) of unit vectors (i.e., Dirac distributions) $e_i = (0, \ldots, 0, 1, 0, \ldots, 0)$, and containing $(R^\lambda, P^\lambda) = (R, P)$. To each corner (R_j, P_j) we associate the (non-empty) set $I_j = \{i \mid (R^{e_i}, P^{e_i}) = (R_j, P_j)\}$ of indices.

We will find $\alpha \in (0, 1)$ and $1 \leqslant u, v \leqslant n$ such that one of A_u or A_v is progressing, and define the multi-strategy θ' to pick A_u and A_v with probabilities α and $1 - \alpha$, respectively. We distinguish several cases, depending on the shape of T:

1. T has non-empty interior. Let $(R_1, P_1), \ldots, (R_m, P_m)$ be its corners in anticlockwise order. Since all λ_i are positive, (R, P) is in the interior of T. Now consider the point (R, P') directly below (R, P) on the boundary of T, i.e. $P' = \min\{P'' \mid (R, P'') \in T\}$. If (R, P') is not a corner point, it is a convex combination of adjacent corners $(R, P') = \alpha(R_j, P_j) + (1-\alpha)(R_{j+1}, P_{j+1})$, and we pick such α and $u \in I_j$ and $v \in I_{j+1}$. If (R, P') happens to be a corner (R_j, P_j) we can (since $P_j < P$) instead choose sufficiently small $\alpha > 0$ so that $R \geqslant \alpha R_j + (1 - \alpha) R_{j+1}$ and $P \leqslant \alpha P_j + (1 - \alpha) P_{j+1}$ and again pick $u \in I_j$ and $v \in I_{j+1}$. In either case, we have A_v progressing since $R_{j+1} > R$.

2. T is a vertical line segment, i.e. it is the convex hull of two extreme points (R, P_0) and (R, P_1) with $P_0 < P_1$. In case $R = 0$, we can simply always allow some A_i with $i \in I_0$, minimising the penalty and still achieving reward 0.

   If $R > 0$, there must be at least one progressing A_u. Since all λ_i are positive, (R, P) lies inside the line segment, and in particular $P > P_0$. We can therefore choose some v and $\alpha \in (0, 1)$ such that $P \leqslant \alpha \cdot P_u + (1 - \alpha) \cdot P_v$.

3. T is a non-vertical line segment, i.e. it is the convex hull of two extreme points (R_0, P_0) and (R_1, P_1) with $R_0 < R_1$. Since all λ_i are positive, (R, P) is not one of the extreme points, i.e. $(R, P) = \alpha(R_0, P_0) + (1 - \alpha)(R_1, P_1)$ with $0 < \alpha < 1$. We can therefore choose $u \in I_0, v \in I_1$. Again, since $R_1 > R$, A_v is progressing.

4. T consists of a single point (R, P). This can be treated like the second case: either $R = 0$, and we can allow any combination, or $R > 0$, and there is some progressing A_u, and we then pick arbitrary v and α.

We now want to show that the reward of the updated multi-strategy is indeed no worse than before. Pick an action a that realises the minimum and strategies σ and π that realise the infimum in the definition of R_i (such strategies indeed exist). Define:

$$c_i = \sum_{s'} \delta(s_1, a)(s') \cdot Pr^{\sigma,\pi}_{\mathsf{G},s'}(\mathtt{F}\ s_1) \tag{B.5}$$

$$d_i = r(s, a) + \sum_{s'} \delta(s_1, a)(s') \cdot E^{\sigma,\pi}_{\mathsf{G},s'}(r{\downarrow}s_1) \tag{B.6}$$

We have $R_i = c_i \cdot R + d_i$ for every $1 \leqslant i \leqslant n$. We also define primed versions c', d', R' and R'_i exactly as their non-primed counterparts, but substituting θ with θ' in the definition, and subsequently have $R'_i = c'_i \cdot R' + d'_i$. Then:

$$\begin{aligned}
R' - R &= (\alpha R'_u + (1-\alpha)R'_v) - \textstyle\sum_i \lambda_i R_i \\
&= (\alpha R'_u + (1-\alpha)R'_v) - (\alpha R_u + (1-\alpha)R_v) + (\alpha R_u + (1-\alpha)R_v) - \textstyle\sum_i \lambda_i R_i \\
&\geqslant (\alpha R'_u + (1-\alpha)R'_v) - (\alpha R_u + (1-\alpha)R_v) \\
&\qquad \text{(by the choice of } \alpha, u, v) \\
&= (\alpha(c'_u R' + d'_u) + (1-\alpha)(c'_v R' + d'_v)) - (\alpha(c_u R + d_u) + (1-\alpha)(c_v R + d_v)) \\
&\geqslant (\alpha(c'_u R' + d'_u) + (1-\alpha)(c'_v R' + d'_v)) - (\alpha(c'_u R + d'_u) + (1-\alpha)(c'_v R + d'_v)) \\
&\qquad (c_i R + d_i \leqslant c'_i R + d'_i \text{ due to the optimality of original opponent strategy}) \\
&= (\alpha c'_u + (1-\alpha)c'_v)(R' - R),
\end{aligned}$$

i.e. $(1 - \alpha c'_u - (1-\alpha)c'_v)(R' - R) \geqslant 0$. By finiteness of rewards and the choice of $\theta(s_1)$, at least one of the return probabilities c'_u, c'_v is less than 1, and thus so is $\alpha c'_u + (1-\alpha)c'_v$, therefore $R' \geqslant R$.

We can show that the penalty under θ' is at most as big as the penalty under θ in exactly the same way (note that in addition using $\psi^{\theta'}_{rew}$ instead of ψ^{θ}_{rew} for c', d', R' and R'_i). For static penalties, the proof that the new multi-strategy is no worse than the old one is straightforward from the choice of $\theta'(s_1)$. □

B.7 Proof of Theorem 10

Proof. Let θ be a multi-strategy and fix s_1 such that θ takes two different actions B and C with probability $p \in (0, 1)$ and $1 - p$ where $B \not\subseteq C$ and $C \not\subseteq B$. If $\inf_{\sigma \lhd \theta, \pi} E^{\sigma,\pi}_{\mathsf{G},s_1}(r) = 0$, then we can in fact allow deterministically the single set $A(s_1)$ and we are done. Hence, suppose that the reward accumulated from s_1 is non-zero.

Suppose, w.l.o.g., that:

$$\min_{a\in B} r(s_1,a) + \sum_{s'\in S} \delta(s_1,a)(s') \cdot \inf_{\sigma\lhd\theta,\pi} E^{\sigma,\pi}_{\mathsf{G},s'}(r) \leqslant \min_{a\in C} r(s_1,a) + \sum_{s'\in S} \delta(s_1,a)(s') \cdot \inf_{\sigma\lhd\theta,\pi} E^{\sigma,\pi}_{\mathsf{G},s'}(r). \tag{B.7}$$

It must be the case that, for some $D \in \{B, C\}$, we have:

$$\min_{a\in D} r(s_1,a) + \inf_{\sigma\lhd\theta,\pi} \sum_{s'\in S} \delta(s_1,a)(s') \cdot E^{\sigma,\pi}_{\mathsf{G},s'}(r{\downarrow}s_1) > 0 \tag{B.8}$$

(otherwise the minimal reward accumulated from s_1 is 0, since there is a compliant strategy that keeps returning to s_1 without ever accumulating any reward), and if the inequality in (B.7) is strict, then (B.8) holds for $D = C$. W.l.o.g. suppose that the above property holds for C. We define θ' by modifying θ and picking $B \cup C$ with probability p, C with $(1-p)$, and B with probability 0.

Under θ, the minimal reward achievable by some compliant strategy is given as the least solution to the following equations [107, Theorem 7.3.3]:

$$\begin{aligned} x_s &= \sum_{A\in supp(\theta(s))} \theta(s)(A) \cdot \min_{a\in A} \sum_{s'\in S} r(s,a) + \delta(s,a)(s') \cdot x_{s'} && \text{for } s \in S_\Diamond \\ x_s &= \min_{a\in A(s)} \sum_{s'\in S} \delta(s,a)(s') \cdot x_{s'} && \text{for } s \in S_\Box \end{aligned}$$

The minimal rewards x'_s achievable under θ' are defined analogously. In particular, for the equation with s_1 on the left-hand side we have:

$$\begin{aligned} x_{s_1} &= p \cdot \min_{a\in B} r(s_1,a) + \sum_{s'\in S} \delta(s_1,a)(s') \cdot x_{s'} + (1-p) \cdot \min_{a\in C} r(s_1,a) + \sum_{s'\in S} \delta(s_1,a)(s') \cdot x_{s'} \\ x'_{s_1} &= p \cdot \min_{a\in B\cup C} r(s_1,a) + \sum_{s'\in S} \delta(s_1,a)(s') \cdot x'_{s'} + (1-p) \cdot \min_{a\in C} r(s_1,a) + \sum_{s'\in S} \delta(s_1,a)(s') \cdot x'_{s'}. \end{aligned}$$

We show that in the least solution $\bar{x}$ to x is also the least solution to x'.

First, note that $\bar{x}$ is clearly a solution to any equation with $s \neq s_1$ on the left-hand side, since these equations remain unchanged in both sets of equations. As for the equation with s_1, we have $\min_{a\in B} \sum_{s'} r(s_1,a) + \delta(s_1,a)(s') \cdot \bar{x}_{s'} \leqslant \min_{a\in C} \sum_{s'} r(s_1,a) + \delta(s_1,a)(s') \cdot \bar{x}_{s'}$, and so necessarily $\min_{a\in B} \sum_{s'} r(s_1,a) + \delta(s_1,a)(s') \cdot \bar{x}_{s'} = \min_{a\in B\cup C} \sum_{s'} r(s_1,a) + \delta(s_1,a)(s') \cdot \bar{x}_{s'}$.

To see that $\bar{x}$ is the *least* solution to x', we show that (i) for all s, if $\inf_{\sigma\lhd\theta',\pi} E^{\sigma,\pi}_{\mathsf{G},s}(r) = 0$ then $\bar{x}_s = 0$; and (ii) there is a unique fixpoint satisfying $\bar{x}_s = 0$ for all s such that $\inf_{\sigma\lhd\theta',\pi} E^{\sigma,\pi}_{\mathsf{G},s}(r) = 0$.

For (i), suppose $\bar{x}_s > 0$. Let σ' be a strategy compliant with θ', and π an arbitrary

strategy. Suppose $Pr^{\sigma',\pi}_{\mathsf{G},s}(\mathtt{F}\ s_1) = 0$, then there is a strategy σ compliant with θ which behaves exactly as σ' when starting from s, and by our assumption on the properties of $\bar{x}_s$ we get that $E^{\sigma,\pi}_{\mathsf{G},s}(r) > 0$ and so $E^{\sigma',\pi}_{\mathsf{G},s}(r) > 0$. Now suppose that $Pr^{\sigma',\pi}_{\mathsf{G},s}(\mathtt{F}\ s_1) > 0$. For this case, we have argued already when defining θ' that the reward under any strategy compliant with θ' is non-zero when starting in s_1, and so $E^{\sigma',\pi}_{\mathsf{G},s}(r) > 0$.

Point (ii) can be obtained by an application of [107, Proposition 7.3.4]. □

B.8 Proof of Theorem 11

Proof. We deal with the cases of static and dynamic penalties separately. For static penalties, let $s \in S$ and $\theta(s)(A_{s,0}) = q_0$, $\theta(s)(A_{s,1}) = q_1$ for $A_{s,0} \subseteq A_{s,1} \subseteq A(s)$. Modify θ by rounding q_0 up and q_1 down to the nearest multiple of $\frac{1}{M}$. The result is again sound (any strategy compliant after the modification is also compliant before the modification), and the penalty changes by at most $\frac{1}{M}\sum_{a\in A(s)}\psi(s,a)$. Repeat for all s.

Now let us consider dynamic penalties. Intuitively, the claim follows since by making small changes to the multi-strategy, while not (dis)allowing any new actions, we only cause small changes to the reward and penalty.

Let θ be a multi-strategy and t a state. By Theorem 9, we can assume $supp(\theta(t)) = \{A_1, A_2\}$. W.l.o.g. suppose:

$$\inf_{\sigma\lhd\theta,\pi}\min_{a\in A_1} r(s,a) + \sum_{s'}\delta(s,a)(s')\cdot E^{\sigma,\pi}_{\mathsf{G},s'}(r) \geqslant \inf_{\sigma\lhd\theta,\pi}\min_{a\in A_2} r(s,a) + \sum_{s'}\delta(s,a)(s')\cdot E^{\sigma,\pi}_{\mathsf{G},s'}(r).$$

For $0 < x < \theta(t)(A_2)$, we define a multi-strategy θ_x by $\theta_x(t)(A_1) = \theta(t)(A_1) + x$ and $\theta_x(t)(A_2) = \theta(t)(A_2) - x$, and $\theta_x(s) = \theta(s)$ for all $s \neq t$. We show that $\inf_{\sigma\lhd\theta_x,\pi} E^{\sigma,\pi}_{\mathsf{G},\bar{s}}(r) \geqslant \inf_{\sigma\lhd\theta,\pi} E^{\sigma,\pi}_{\mathsf{G},\bar{s}}(r)$. Consider the following functional $F_x : (S \to \mathbb{R}_{\geqslant 0}) \to (S \to \mathbb{R}_{\geqslant 0})$, constructed for the multi-strategy θ_x:

$$F_x(f)(s) = \sum_{A\in supp(\theta(s))}\theta_x(s)\cdot\min_{a\in A}\sum_{s'\in S} r(s,a) + \delta(s,a)(s')\cdot f(s') \qquad \text{for } s \in S_\Diamond$$

$$F_x(f)(s) = \min_{a\in A(s)}\sum_{s'\in S} r(s,a) + \delta(s,a)(s')\cdot f(s') \qquad \text{for } s \in S_\Box$$

Let f be the function assigning $\inf_{\sigma\lhd\theta,\pi} E^{\sigma,\pi}_{\mathsf{G},t}(r)$ to s. Observe that $f(s) = 0$ whenever $\inf_{\sigma\lhd\theta_x,\pi} E^{\sigma,\pi}_{\mathsf{G},t}(r)$; this follows since $x < \min\{\theta(t)(A_1), \theta(t)(A_2)\}$ and so both θ and θ_x allow the same actions with non-zero probability. Also, $F_x(f)(s) \geqslant f(s)$: for $s \neq t$ in fact $F_x(f)(s) = f(s)$ because the corresponding functional F for θ coincides with F_x on s; for $s = t$, we have $F_x(f)(s) \geqslant f(s)$ since $\min_{a\in A_1} r(s,a) + \sum_{s'}\delta(s,a)(s')\cdot f(s') \geqslant$

$\min_{a\in A_2} r(s,a) + \sum_{s'} \delta(s,a)(s')\cdot f(s')$ by the properties of A_1 and A_2 and since x is non-negative. Hence, we can apply [107, Proposition 7.3.4] and obtain that θ_x ensures at least the same reward as θ. Thus, by increasing the probability of allowing A_1 in t the soundness of the multi-strategy is preserved.

Further, for any strategy σ' compliant with θ_x and any π, the penalty when starting in t, i.e. $E^{\sigma',\pi}_{\mathsf{G},t}(\psi^{\theta_x}_{rew})$, is equal to:

$$pen_{loc}(\psi,\theta_x,t) + \sum_{a\in A}\xi'(t)(a)\sum_{t'\in S}\delta(t,a)(t')\cdot\left(E^{\sigma',\pi}_{\mathsf{G},t'}(\psi^{\theta_x}_{rew}{\downarrow}t) + Pr^{\sigma',\pi}_{\mathsf{G},t'}(\mathtt{F}\ t)\cdot E^{\sigma',\pi}_{\mathsf{G},t}(\psi^{\theta_x}_{rew})\right)$$

for $\xi' = \sigma'\cup\pi$. There is a strategy σ compliant with θ which differs from σ' only on t, where $\sum_{a\in A}|\sigma'(s,a)-\sigma(s,a)| \leqslant x$. We have, for any π:

$$\begin{aligned}
&E^{\sigma,\pi}_{\mathsf{G},t}(\psi^{\theta}_{rew})\\
&= pen_{loc}(\psi,\theta,t) + \sum_{a\in A}\xi(t,a)\sum_{s\in S}\delta(t,a)(s)\cdot(E^{\sigma,\pi}_{\mathsf{G},s}(\psi^{\theta}_{rew}{\downarrow}t) + Pr^{\sigma,\pi}_{\mathsf{G},s}(\mathtt{F}\ t)\cdot E^{\sigma,\pi}_{\mathsf{G},t}(\psi^{\theta}_{rew}))\\
&\geqslant pen_{loc}(\psi,\theta_x,t) - x + \sum_{a\in A}(\xi'(t,a)-x)\sum_{s\in S}\delta(t,a)(s)\cdot(E^{\sigma',\pi}_{\mathsf{G},s}(\psi^{\theta_x}_{rew}{\downarrow}t) + Pr^{\sigma',\pi}_{\mathsf{G},s}(\mathtt{F}\ t)\cdot E^{\sigma,\pi}_{\mathsf{G},t}(\psi^{\theta_x}_{rew}))
\end{aligned}$$

where $\xi = \sigma\cup\pi$ and the rest is as above.

Thus:

$$\begin{aligned}
E^{\sigma',\pi}_{\mathsf{G},t}(\psi^{\theta_x}_{rew}) &= \frac{pen_{loc}(\psi,\theta_x,t) + \sum_{a\in A}\xi'(t)(a)\sum_{t'\in S}\delta(t,a)(t')\cdot E^{\sigma',\pi}_{\mathsf{G},t'}(\psi^{\theta_x}_{rew}{\downarrow}t)}{1-\sum_{a\in A}\xi'(t)(a)\sum_{t'\in S}\delta(t,a)(t')\cdot Pr^{\sigma',\pi}_{\mathsf{G},t'}(\mathtt{F}\ t)}\\
E^{\sigma,\pi}_{\mathsf{G},t}(\psi^{\theta}_{rew}) &\geqslant \frac{pen_{loc}(\psi,\theta,t) - x + \sum_{a\in A}(\xi'(t)(a)-x)\sum_{t'\in S}\delta(t,a)(t')\cdot E^{\sigma',\pi}_{\mathsf{G},t'}(\psi^{\theta}_{rew}{\downarrow}t)}{1-\sum_{a\in A}(\xi'(t)(a)-x)\sum_{t'\in S}\delta(t,a)(t')\cdot Pr^{\sigma',\pi}_{\mathsf{G},t'}(\mathtt{F}\ t)}
\end{aligned}$$

and so $E^{\sigma',\pi}_{\mathsf{G},t}(\psi^{\theta_x}_{rew}) - E^{\sigma,\pi}_{\mathsf{G},t}(\psi^{\theta}_{rew})$ goes to 0 as x goes to 0. Hence, $pen_{dyn}(\psi,\theta_x) - pen_{dyn}(\psi,\theta)$ goes to 0 as x goes to 0.

The above gives us that, for any error bound ε and a fixed state s, there is an x such that we can modify the decision of θ in s by x, not violate the soundness property and increase the penalty by at most $\varepsilon/|S|$. We thus need to pick M such that $1/M \leqslant x$. To finish the proof, we repeat this procedure for every state s. □

Appendix C

Proofs for Chapter 7

C.1 Proof of Theorem 12

Assumption 1. *Consider Algorithm 4 with* UPDATE *defined in Algorithm 4, but now with line 22 being "**until** false", i.e. iterating the outer repeat loop ad infinitum. Denote the functions U and L after i iterations by U_i and L_i, respectively.*

Now, we prove two Lemmas that are crucial for proving Theorem 12.

Lemma 6. *For every $i \in \mathbb{N}$, all $s \in S$ and $a \in A$,*

$$U_1(s,a) \geqslant \cdots \geqslant U_i(s,a) \geqslant V(s,a) \geqslant L_i(s,a) \geqslant \cdots \geqslant L_1(s,a).$$

Proof. Simple induction. □

Lemma 7. $\lim_{i\to\infty}(U_i(\overline{s}) - L_i(\overline{s})) = 0$ *almost surely.*

Proof. Let $a_i(s) \in A(s)$ maximise $U_i(s,a)$ if $s \in S_\Diamond$ and minimise $L_i(s,a)$ when $s \in S_\Box$. We define $\Delta_i(s) := U_i(s,a_i(s)) - L_i(s,a_i(s))$. Since $\Delta_i(s) \geqslant \max_a U_i(s,a) - \max_a L_i(s,a)$ when $s \in S_\Diamond$ and $\Delta_i(s) \geqslant \min_a U_i(s,a) - \min_a L_i(s,a)$ if $s \in S_\Box$ (expression of line 22 in the original Algorithm 4), it is sufficient to prove that $\lim_{i\to\infty} \Delta_i(\overline{s}) = 0$ almost surely.

By Lemma 6, the limits $\lim_{i\to\infty} U_i(s,a)$ and $\lim_{i\to\infty} L_i(s,a)$ are well defined and finite. Thus $\lim_{i\to\infty} \Delta_i(s)$ is also well defined and we denote it by $\Delta(s)$ for every $s \in S$.

Let $\Sigma_{U,L}$ be the set of pairs of all memoryless strategies in G which occur as $(\sigma_{U_i}, \pi_{L_i})$ for infinitely many i. Each pair $(\sigma_{U_i}, \pi_{L_i}) \in \Sigma_{U,L}$ induces a chain with reachable state space $S_{(\sigma,\pi)}$ and uses actions $A_{(\sigma,\pi)}$. Note that under $(\sigma,\pi) \in \Sigma_{U,L}$, all states of $S_{(\sigma,\pi)}$ will be almost surely visited infinitely often if infinitely many simulations are run. Similarly, all actions of $A_{(\sigma,\pi)}$ will be used almost surely infinitely many times. Let $S_\infty = \bigcup_{(\sigma,\pi)\in\Sigma_{U,L}} S_{(\sigma,\pi)}$ and let $A_\infty = \bigcup_{(\sigma,\pi)\in\Sigma_{U,L}} A_{(\sigma,\pi)}$. During almost all computations of the learning algorithm,

all states of S_∞ are visited infinitely often, and all actions of A_∞ are used infinitely often. By definition of Δ, for every $t \in S_\infty$ and $a \in A_\infty$ it holds that $\Delta(t) = \sum_{s \in S_\infty} \delta(t,a)(s) \cdot \Delta(s)$ almost surely.

Let $\Delta = \max_{s \in S_\infty} \Delta(s)$ and $D = \{s \in S_\infty \mid \Delta(s) = \Delta\}$. To obtain a contradiction, consider a computation of the learning algorithm such that $\Delta > 0$ and $\Delta(t) = \sum_{s \in S_\infty} \delta(t,a)(s) \cdot \Delta(s)$ for all $t \in S$ and $a \in A(t)$. Then $\mathbf{1}, \mathbf{0} \notin D$ and thus D cannot contain any EC by assumption. By definition of EC we get:

$$\exists t \in D : \forall a \in A(t) : \mathit{supp}(\delta(t,a)) \not\subseteq D$$

and thus for every $a \in A(t)$ we have $t_a \notin D$ with $\delta(t,a)(t_a) > 0$. Since $t_a \notin D$ we have $\Delta(t_a) < \Delta$. Now for every $a \in A(t) \cap A_\infty$ we have:

$$\begin{aligned}
\Delta(t) &= \sum_{s \in S_\infty, s \neq t_a} \delta(t,a)(s) \cdot \Delta(s) + \delta(t,a)(t_a) \cdot \Delta(t_a) \\
&< \sum_{s \in S_\infty, s \neq t_a} \delta(t,a)(s) \cdot \Delta + \delta(t,a)(t_a) \cdot \Delta \\
&= \Delta
\end{aligned}$$

a contradiction with $t \in D$.

□

Proof. As a corollary of Proof C.1, Algorithm 4 terminates for any $\varepsilon > 0$. Further, $U_i \geqslant V \geqslant L_i$ pointwise and invariantly for every i, by the first lemma, the returned result is correct.

□

C.2 Proof of Lemma 4

Proof. **Point 1.** The function MAKETERMINAL$(s_{(R,B)}, \mathbf{0})$ is called in Algorithm 6 in two cases:

- If there are no actions available in state $s_{(R,B)}$ and $R \cap T = \emptyset$ and $\mathit{player}(R) = \Diamond$. It follows that the support of all the actions that were enabled in states in R in the stochastic game G remains in R, i.e, there is no action leaving the set R. As $R \cap T = \emptyset$, it follows that for all states in R the probability to reach the target state is 0, and therefore $\forall s \in R : \ V_{\mathsf{G}}(s) = 0$.

- If $R \cap T = \emptyset$ and $player(R) = \Box$. From the definition of one-player EC it follows that for every $s \in R$ there exists an action $a \in A(s)$ such that $supp(a) \subseteq R$. As $R \cap T = \emptyset$, if we pick that action we will never reach a state in T causing the reachability probability to be 0. As $player(R) = \Box$ and player $\Box$ wants to minimise such probability, it will always pick such actions, from which it follows $\forall s \in R : \ V_{\mathsf{G}}(s) = 0$.

The function MAKETERMINAL$(s_{(R,B)}, \mathbf{1})$ is called in Algorithm 6 in two cases:

- If $R \cap T \neq \emptyset$ and $player(R) = \Diamond$. A strategy in the stochastic game G that plays in state $s \in R$ all the actions $A(s) \cap B$ uniformly at random will visit all the states in R almost surely. It follows that from every state $s \in R$ the target set is reached almost surely. As $player(R) = \Diamond$ and player $\Diamond$ wants to maximize such probability, it will always pick such actions, from which it follows that $\forall s \in R : \ V_{\mathsf{G}}(s) = 1$.

- If $R \cap T \neq \emptyset$ and there are no actions available in state $s_{(R,B)}$. It follows that the support of all the actions that were enabled in states in R in the stochastic game G remains in R, i.e, there is no action leaving the set R. As $R \cap T \neq \emptyset$, it follows that for all states in R the probability to reach the target state is 1, and therefore $\forall s \in R : \ V_{\mathsf{G}}(s) = 1$.

Points 2 and 3. These two points follow from Theorem 2 of [36]. □

C.3 Proof of Theorem 13

Proof. Consider Algorithm 7 with line 30 being "**until** false", i.e. iterating the outer repeat loop ad infinitum. As the stochastic game G may change during computation of the learning algorithm, we denote by $\mathsf{G}_i = \langle S_i, \xi_i, \overline{s}, A_i, \delta_i, \mathcal{L}_i \rangle$ the current stochastic game after i iterations of the outer repeat-until cycle of Algorithm 7. Each G_i is obtained from G by possibly several collapses of end-components; we denote new states that are added to the game during collapsing by ξ_i. Recall that in the stochastic game G' obtained by collapsing (R, B), the state $s_{(R,B)}$ corresponds to the set of states R and, in particular, $V_{\mathsf{G}}(s_{(R,B)}, a) = V_{\mathsf{G}'}(s, a)$ for all actions a that are enabled both in $s_{(R,B)}$ and in s. Thus, slightly abusing notation, we may consider states of each G_i to be sets of states of the original stochastic game G. So, given a state $\xi \in S_i$ of G_i, we write $s \in \xi$ to say that the state $s \in S$ of G belongs to (or corresponds to) the state ξ.

Note that $V_{\mathsf{G}}(s, a) = V_{\mathsf{G}_i}(\xi, a)$ for $s \in \xi \in S_i$ and all $a \in A_i(\xi)$. Thus, in what follows we use $V(s, a)$ to denote $V_{\mathsf{G}}(s, a)$. We also denote by U_i and L_i the functions U and L

after i iterations. Observe that $U_i, L_i : S_i \times A_i \to [0,1]$. We extend U_i and L_i to states of S by $U_i(s,a) := U_i(\xi,a)$ and $L_i(s,a) := L_i(\xi,a)$ for $s \in \xi \in S_i$ and all $a \in A_i(\xi)$. We also use $A_i(s)$ to denote $A_i(\xi)$ for $s \in \xi \in S_i$.

Lemma 8. *For all $s \in S$, every $i \in \mathbb{N}$ and all $a \in A_i(s)$,*

$$U_1(s,a) \geqslant \cdots \geqslant U_i(s,a) \geqslant V(s,a) \geqslant L_i(s,a) \geqslant \cdots \geqslant L_1(s,a)$$

Proof. A simple induction applies if one-player ECs are not collapsed in the i-th iteration of the outer cycle of Algorithm 7. Otherwise, if they are collapsed, then the claim follows from the fact that collapsing preserves the values of U, L, and V (see lines 6 and 7 of Algorithm 6). □

Lemma 9. $\lim_{i\to\infty}(U_i(\overline{s}) - L_i(\overline{s})) = 0$ *almost surely.*

Proof. Let $a_i(s) \in A(s)$, and maximise $U_i(s,a)$ if $s \in S_\Diamond$ and minimise $L_i(s,a)$ when $s \in S_\Box$. We define $\Delta_i(s) := U_i(s,a_i(s)) - L_i(s,a_i(s))$. Since $\Delta_i(s) \geqslant \max_a U_i(s,a) - \max_a L_i(s,a)$ when $s \in S^\Diamond$ and $\Delta_i(s) \geqslant \min_a U_i(s,a) - \min_a L_i(s,a)$ if $s \in S_\Box$ (expression of line 20 in the original Algorithm 4), it is sufficient to prove that $\lim_{i\to\infty}\Delta_i(\overline{s}) = 0$ almost surely.

As the function On-the-fly-EC can collapse one-player ECs only finitely many times, every computation of the learning algorithm eventually stays with a fixed stochastic game $\mathsf{G}' = \langle S', \xi', \overline{s}', A', \delta'\mathcal{L}'\rangle$, i.e., almost surely $\mathsf{G}' = \mathsf{G}_k = \mathsf{G}_{k+1} = \cdots$ for some k. Note that G' is obtained by a series of collapses of end-components of G. We call the moment from which the stochastic game does not change the *fixing point*.

Let us denote by S' the set of states of G'. Note that for every $\xi \in S'$ and for all $s, s' \in \xi$ we have $\Delta(s) = \Delta(s')$ since $\Delta_i(s) = \Delta_i(s')$ for i greater than the fixing point. We denote by $\Delta(\xi)$ the value $\Delta(s)$ for some (all) $s \in \xi$. For every $\xi \in S'$ we denote by $A'(\xi)$ the set of actions enabled in the state ξ of G'. Also, the initial state, $\overline{\xi}$, of G' is the only state of G' that contains $\overline{s}$.

Let $\Sigma_{U,L}$ be the set of pairs of all memoryless strategies in G' which occur as $(\sigma_{U_i}, \pi_{L_i})$ for infinitely many i *after* the fixing point. Each pair $(\sigma,\pi) \in \Sigma_{U,L}$ induces a chain with reachable state space $S'_{(\sigma,\pi)}$ and uses actions $A'_{(\sigma,\pi)}$. Note that, under $(\sigma,\pi) \in \Sigma_{U,L}$, all states of $S'_{(\sigma,\pi)}$ will be almost surely visited infinitely often if infinitely many simulations are run. Similarly, all actions of $A'_{(\sigma,\pi)}$ will be used almost surely infinitely many times. Let $S'_\infty = \bigcup_{(\sigma,\pi)\in\Sigma_{U,L}} S'_i$ and let $A'_\infty = \bigcup_{(\sigma,\pi)\in\Sigma_{U,L}} A'_{(\sigma,\pi)}$. During almost all computations of the learning algorithm, all states of S'_∞ are visited infinitely often, and all actions of A'_∞ are used infinitely often.

Let $\Delta = \max_{\xi \in S'_\infty} \Delta(\xi)$ and $D = \{\xi \in S'_\infty \mid \Delta(\xi) = \Delta\}$. To obtain a contradiction, assume that $\Delta > 0$, which implies that $\mathbf{1}, \mathbf{0} \notin D$. We claim that D cannot contain a subset D' forming an end-component with any set of actions from A'_∞. Indeed, assume the opposite is true, and (D', G) is such an end component in G'. At least one pair of the strategies $(\sigma, \pi) \in \Sigma_{U,L}$ visits a state of D' infinitely many times. As all state-action pairs $(\xi, a) \in D' \times G(\xi)$ satisfy $U_i(\xi, a) = 1$ for all i and $k_i \geqslant |S|$, almost surely a simulation of (σ, π) of length k_i visits the whole component (D', G). This means that On-the-fly-EC is called while currently explored path contains the component (D', G), which in turn means that (D', G) gets collapsed, a contradiction with the assumption that the learning procedure stays fixed on G' after the fixing point.

By definition of one-player EC we get:

$$\exists \xi \in D : \forall a \in A'(\xi) \cap A'_\infty : \mathit{supp}(\delta(\xi, a)) \not\subseteq D$$

because otherwise D will form a "closed" component with some actions of A'_∞, and hence would contain an one-player EC . Thus for every $a \in A'(\xi) \cap A'_\infty$ we have $\xi_a \notin D$ with $\delta(\xi, a)(\xi_a) > 0$. Since $\xi_a \notin D$ we have $\Delta(\xi_a) < \Delta$. Now for every $a \in A'(\xi) \cap A'_\infty$ we have:

$$\begin{aligned}
\Delta(\xi) &= \sum_{\xi' \in S'_\infty} \delta(\xi, a)(\xi') \cdot \Delta(\xi') \\
&= \sum_{\xi' \in S'_\infty, \xi' \neq \xi_a} \delta(\xi, a)(\xi') \cdot \Delta(\xi) + \delta(\xi, a)(\xi_a) \cdot \Delta(\xi_a) \\
&< \sum_{\xi' \in S'_\infty, \xi' \neq \xi_a} \delta(\xi, a)(\xi') \cdot \Delta + \delta(\xi, a)(\xi_a) \cdot \Delta \\
&= \Delta
\end{aligned}$$

a contradiction with $\xi \in D$. □

As a corollary, Algorithm 7 with Update defined in Algorithm 6 and extended with calls to On-the-fly-EC almost surely terminates for any $\varepsilon > 0$. Further, $U_i \geqslant V \geqslant L_i$ pointwise and invariantly for every i by the first claim, the returned result is correct. □

Appendix D

PRISM model of *StockPriceViewer* Application

```
smg

player env
    [web_stock_0_fail],    [web_stock_1_fail],    [web_stock_2_fail]
    [retry_stock], [loop], [no_retry_stock]
endplayer
player controller
    [web_stock_0],    [web_stock_1],    [web_stock_2]
endplayer
const int   max_retry=1;
const int   stock_to_query=60;
const int   web_stock_number=3;
const double web_stock_0_fail;
const double web_stock_1_fail;
const double web_stock_2_fail;
const double web_stock_0_response_time;
const double web_stock_1_response_time;
const double web_stock_2_response_time;
const int stock_in_portfolio=10;
//————————————————————————————————
module QueryStock
   web_stock_0_retry : [0..max_retry + 1] init 0;
   web_stock_1_retry : [0..max_retry + 1] init 0;
   web_stock_2_retry : [0..max_retry + 1] init 0;

   pc : [0..5] init 0;
   last_stock_webservice : [0..web_stock_number + 1] init 0;
   stock_querued : [0..stock_to_query + 1] init 0;
   [web_stock_0] (pc=0)&(stock_querued<stock_to_query)&(web_stock_0_retry<max_retry) →
       (pc′=1)&(last_stock_webservice′=0);
   [web_stock_0_fail] (pc=1)&(last_stock_webservice=0) →
       (web_stock_0_fail) : (pc′=2) +
       (1 − web_stock_0_fail) : (pc′=0)&(stock_querued′=min(stock_querued + 1, stock_to_query));
   [retry_stock] (pc=2)&(last_stock_webservice=0) →
       (pc′=0)&(web_stock_0_retry′=min(web_stock_0_retry + 1, max_retry));
   [no_retry_stock] (pc=2)&(last_stock_webservice=0) →
       (pc′=0)&(web_stock_0_retry′=max_retry);
   [loop] (pc=2)&(last_stock_webservice=0)&(web_stock_0_retry=max_retry) →
       (pc′=0);
   [web_stock_1] (pc=0)&(stock_querued<stock_to_query)&(web_stock_1_retry<max_retry) →
      (pc′=1)&(last_stock_webservice′=1);
   [web_stock_1_fail] (pc=1)&(last_stock_webservice=1) →
       (web_stock_1_fail) : (pc′=2) +
       (1 − web_stock_1_fail) : (pc′=0)&(stock_querued′=min(stock_querued + 1, stock_to_query));
   [retry_stock] (pc=2)&(last_stock_webservice=1) →
       (pc′=0)&(web_stock_1_retry′=min(web_stock_1_retry + 1, max_retry));
   [no_retry_stock] (pc=2)&(last_stock_webservice=1) →
       (pc′=0)&(web_stock_1_retry′=max_retry);
   [loop] (pc=2)&(last_stock_webservice=1)&(web_stock_1_retry=max_retry) →
       (pc′=0);
    [web_stock_2] (pc=0)&(stock_querued<stock_to_query)&(web_stock_2_retry<max_retry) →
       (pc′=1)&(last_stock_webservice′=2);
   [web_stock_2_fail] (pc=1)&(last_stock_webservice=2) →
      (web_stock_2_fail) : (pc′=2) +
       (1 − web_stock_2_fail) : (pc′=0)&(stock_querued′=min(stock_querued + 1, stock_to_query));
   [retry_stock] (pc=2)&(last_stock_webservice=2) →
       (pc′=0)&(web_stock_2_retry′=min(web_stock_2_retry + 1, max_retry));
   [no_retry_stock] (pc=2)&(last_stock_webservice=2) →
       (pc′=0)&(web_stock_2_retry′=max_retry);
   [loop] (pc=2)&(last_stock_webservice=2)&(web_stock_2_retry=max_retry) →
       (pc′=0);
   [loop] (pc=0)&(stock_querued=stock_to_query) →
       true;
   [loop] (pc=0)&(web_stock_0_retry=max_retry)&(web_stock_1_retry=max_retry)&(web_stock_2_retry=max_retry) →
       true;
endmodule
```

Figure D.1: PRISM model *android_3* of *StockPriceViewer* application supporting three different providers.

```
smg

player env
    [web_stock_0_fail], [web_stock_1_fail], [web_stock_2_fail], [web_stock_3_fail],
    [retry_stock], [loop], [no_retry_stock]
endplayer
player controller
    [web_stock_0], [web_stock_1], [web_stock_2], [web_stock_3]
endplayer
const int   max_retry=1;
const int   stock_to_query=60;
const int   web_stock_number=4;
const double web_stock_0_fail;
const double web_stock_1_fail;
const double web_stock_2_fail;
const double web_stock_3_fail;
const double web_stock_0_response_time;
const double web_stock_1_response_time;
const double web_stock_2_response_time;
const double web_stock_3_response_time;
const int   stock_in_portfolio=10;
//————————————————————————————
module QueryStock
   web_stock_0_retry : [0..max_retry+1] init 0;
   web_stock_1_retry : [0..max_retry+1] init 0;
   web_stock_2_retry : [0..max_retry+1] init 0;
   web_stock_3_retry : [0..max_retry+1] init 0;

   pc : [0..5] init 0;
   last_stock_webservice : [0..web_stock_number+1] init 0;
   stock_queried : [0..stock_to_query+1] init 0;
   [web_stock_0] (pc=0)&(stock_queried<stock_to_query)&(web_stock_0_retry<max_retry) →
      (pc'=1)&(last_stock_webservice'=0);
   [web_stock_0_fail] (pc=1)&(last_stock_webservice=0) →
      (web_stock_0_fail) : (pc'=2) +
       (1 − web_stock_0_fail) : (pc'=0)&(stock_queried'=min(stock_queried+1, stock_to_query));
   [retry_stock] (pc=2)&(last_stock_webservice=0) →
      (pc'=0)&(web_stock_0_retry'=min(web_stock_0_retry+1, max_retry));
   [no_retry_stock] (pc=2)&(last_stock_webservice=0) →
      (pc'=0)&(web_stock_0_retry'=max_retry);
   [loop] (pc=2)&(last_stock_webservice=0)&(web_stock_0_retry=max_retry) →
      (pc'=0);

   [web_stock_1] (pc=0)&(stock_queried<stock_to_query)&(web_stock_1_retry<max_retry) →
      (pc'=1)&(last_stock_webservice'=1);
   [web_stock_1_fail] (pc=1)&(last_stock_webservice=1) →
      (web_stock_1_fail) : (pc'=2) +
       (1 − web_stock_1_fail) : (pc'=0)&(stock_queried'=min(stock_queried+1, stock_to_query));
   [retry_stock] (pc=2)&(last_stock_webservice=1) →
      (pc'=0)&(web_stock_1_retry'=min(web_stock_1_retry+1, max_retry));
   [no_retry_stock] (pc=2)&(last_stock_webservice=1) →
      (pc'=0)&(web_stock_1_retry'=max_retry);
   [loop] (pc=2)&(last_stock_webservice=1)&(web_stock_1_retry=max_retry) →
       (pc'=0);
   [web_stock_2] (pc=0)&(stock_queried<stock_to_query)&(web_stock_2_retry<max_retry) →
      (pc'=1)&(last_stock_webservice'=2);
   [web_stock_2_fail] (pc=1)&(last_stock_webservice=2) →
      (web_stock_2_fail) : (pc'=2) +
       (1 − web_stock_2_fail) : (pc'=0)&(stock_queried'=min(stock_queried+1, stock_to_query));
   [retry_stock] (pc=2)&(last_stock_webservice=2) →
      (pc'=0)&(web_stock_2_retry'=min(web_stock_2_retry+1, max_retry));
   [no_retry_stock] (pc=2)&(last_stock_webservice=2) →
      (pc'=0)&(web_stock_2_retry'=max_retry);
   [loop] (pc=2)&(last_stock_webservice=2)&(web_stock_2_retry=max_retry) →
      (pc'=0);
```

Figure D.2: PRISM model *android_4* of *StockPriceViewer* application supporting four different providers (continued in Figure D.3).

```
[web_stock_3] (pc=0)&(stock_queued<stock_to_query)&(web_stock_3_retry<max_retry) →
    (pc'=1)&(last_stock_webservice'=3);
[web_stock_3_fail] (pc=1)&(last_stock_webservice=3) →
    (web_stock_3_fail) : (pc'=2) +
    (1 − web_stock_3_fail) : (pc'=0)&(stock_queued'=min(stock_queued + 1, stock_to_query));
[retry_stock] (pc=2)&(last_stock_webservice=3) →
    (pc'=0)&(web_stock_3_retry'=min(web_stock_3_retry + 1, max_retry));
[no_retry_stock] (pc=2)&(last_stock_webservice=3) →
    (pc'=0)&(web_stock_3_retry'=max_retry);
[loop] (pc=2)&(last_stock_webservice=3)&(web_stock_3_retry=max_retry) →
    (pc'=0);
[loop] (pc=0)&(stock_queued=stock_to_query) →
    true;
[loop] (pc=0)&(web_stock_0_retry=max_retry)&(web_stock_1_retry=max_retry)&
    (web_stock_2_retry=max_retry)&(web_stock_3_retry=max_retry) →
    true;
endmodule
```

Figure D.3: PRISM model *android_4* of *StockPriceViewer* application supporting four different providers.

```
rewards "response_time"

[web_stock_0] true : web_stock_0_response_time * stock_in_portfolio;
[web_stock_1] true : web_stock_1_response_time * stock_in_portfolio;
[web_stock_2] true : web_stock_2_response_time * stock_in_portfolio;
[web_stock_3] true : web_stock_3_response_time * stock_in_portfolio;

endrewards

penalties "penalties"

[web_stock_0] true : 1/(web_stock_0_response_time * stock_in_portfolio);
[web_stock_1] true : 1/(web_stock_1_response_time * stock_in_portfolio);
[web_stock_2] true : 1/(web_stock_2_response_time * stock_in_portfolio);
[web_stock_3] true : 1/(web_stock_3_response_time * stock_in_portfolio);

endpenalties

Property :

<<controller>> R≤66000 [ C ]
```

Figure D.4: Rewards, penalties and the property for the model of *StockPriceViewer* application.

Bibliography

[1] H. Aljazzar. *Directed diagnostics of system dependability models.* PhD thesis, University of Konstanz, 2009.

[2] H. Aljazzar, M. Kuntz, F. Leitner-Fischer, and S. Leue. Directed and heuristic counterexample generation for probabilistic model checking: a comparative evaluation. In *Proc. 1st Workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems (QUOVADIS'10)*, pages 25–32. ACM, 2010.

[3] H. Aljazzar, F. Leitner-Fischer, S. Leue, and D. Simeonov. DiPro: A tool for probabilistic counterexample generation. In *Proc. 18th International SPIN conference on Model checking software (SPIN'11)*, pages 183–187. Springer, 2011.

[4] H. Aljazzar and S. Leue. Generation of counterexamples for model checking of Markov decision processes. In *Proc. 6th International Conference on Quantitative Evaluation of Systems (QEST'09)*, pages 197–206. IEEE, 2009.

[5] C. Baier, F. Ciesinski, and M. Grosser. Probmela: A modelling language for communicating probabilistic processes. In *Proc. 2nd International Conference on Formal Methods and Models for Co-Design (MEMOCODE'04)*, pages 57–66. IEEE, 2004.

[6] C. Baier, M. Grosser, M. Leucker, B. Bollig, and F. Ciesinski. Controller synthesis for probabilistic systems. In *Proc. 3rd International Conference on Theoretical Computer Science (TCS'06)*, pages 493–506. Kluwer, 2004.

[7] C. Baier and J.-P. Katoen. *Principles of Model Checking.* MIT Press, 2008.

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

[9] A. G. Barto, S. J. Bradtke, and S. P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1âĂŞ2):81–138, 1995.

[10] G. Behrmann, A. David, K. G. Larsen, P. Pettersson, W. Yi, and M. Hendriks. Uppaal 4.0. In *Proc. 3rd International Conference on Quantitative Evaluation of Systems (QEST'06)*, pages 125–126. IEEE, 2006.

[11] J. Bernet, D. Janin, and I. Walukiewicz. Permissive strategies: From parity games to safety games. *RAIRO-Theoretical Informatics and Applications*, 36(3):261–275, 2002.

[12] D. Bertsekas. *Dynamic Programming and Optimal Control*, Volumes 1 and 2. Athena Scientific, 1995.

[13] G. Blair, N. Bencomo, and R. B. France. Models@ run.time. *Computer*, 42(10):22–27, 2009.

[14] B. Bonet and H. Geffner. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *Proc. 13th The International Conference on Automated Planning and Scheduling (ICAPS'03)*, pages 12–21. AAAI, 2003.

[15] P. Bouyer, M. Duflot, N. Markey, and G. Renault. Measuring permissivity in finite games. In *Proc. 20th International Conference on Concurrency Theory (CONCUR'09)*, pages 196–210. Springer, 2009.

[16] P. Bouyer, N. Markey, J. Olschewski, and M. Ummels. Measuring permissiveness in parity games: Mean-payoff parity games revisited. In *Proc. 9th International Symposium on Automated Technology for Verification and Analysis (ATVA'11)*, pages 135–149. Springer, 2011.

[17] T. Brazdil, K. Chatterjee, M. Chmelik, V. Forejt, J. Kretinsky, M. Kwiatkowska, D. Parker, and M. Ujma. Verification of Markov decision processes using learning algorithms. In *Proc. 12th International Symposium on Automated Technology for Verification and Analysis (ATVA'14)*, pages 98–114. Springer, 2014.

[18] R. Calinescu. General-purpose autonomic computing. In *Autonomic Computing and Networking*, pages 3–30. Springer, 2009.

[19] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola. Self-adaptive software needs quantitative verification at runtime. *Communications of the ACM*, 55(9):69–77, 2012.

[20] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic QoS management and optimisation in service-based systems. *IEEE Transactions on Software Engineering*, 37(3):387–409, 2011.

[21] R. Calinescu, K. Johnson, and S. Kikuchi. Compositional reverification of probabilistic safety properties for large-scale complex IT systems. In *Proc. 17th Monterey Workshop on Development, Operation and Management of Large-Scale Complex IT Systems*, pages 303–329. Springer, 2012.

[22] R. Calinescu, K. Johnson, and Y. Rafiq. Using observation ageing to improve markovian model learning in QoS engineering. In *Proc. 2nd International Conference on Performance engineering (ICPE'11)*, pages 505–510. ACM, 2011.

[23] R. Calinescu, K. Johnson, and Y. Rafiq. Developing self-verifying service-based systems. In *Proc. 28th International Conference on Automated Software Engineering (ASE'13)*, pages 734–737. IEEE, 2013.

[24] R. Calinescu and S. Kikuchi. Formal methods@ runtime. In *Proc. 16th Monterey Workshop on Foundations of Computer Software*, pages 122–135. Springer, 2011.

[25] R. Calinescu and M. Kwiatkowska. Using quantitative analysis to implement autonomic IT systems. In *Proc. 31st International Conference on Software Engineering (ICSE'09)*, pages 100–110. IEEE, 2009.

[26] J. Cámara, G. A Moreno, and D. Garlan. Stochastic game analysis and latency awareness for proactive self-adaptation. In *Proc. 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14)*, pages 155–164. ACM, 2014.

[27] J. Canny. Some algebraic and geometric computations in PSPACE. In *Proc. 20th Symposium on Theory of Computing (STOC'88)*, pages 460–467. ACM, 1988.

[28] K. Chatterjee and M. Henzinger. Faster and dynamic algorithms for maximal end-component decomposition and related graph problems in probabilistic verification. In *Proc. 22nd Symposium on Discrete Algorithms (SODA'11)*, pages 1318–1336. SIAM, 2011.

[29] K. Chatterjee and T. Henzinger. A survey of stochastic ω-regular games. *Journal of Computer and System Sciences*, 78(2):394–413, 2012.

[30] T. Chen, V. Forejt, M. Kwiatkowska, D. Parker, and A. Simaitis. Automatic verification of competitive stochastic systems. *Formal Methods in System Design*, 43(1):61–92, 2013.

[31] T. Chen, V. Forejt, M. Kwiatkowska, D. Parker, and A. Simaitis. PRISM-games: A model checker for stochastic multi-player games. In *Proc. 19th International*

Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13), pages 185–191. Springer, 2013.

[32] T. Chen, E. M. Hahn, T. Han, M. Kwiatkowska, H. Qu, and L. Zhang. Model repair for Markov decision processes. In *Proc. 7th International Symposium on Theoretical Aspects of Software Engineering (TASE'13)*, pages 85–92. IEEE, 2013.

[33] T. Chen, M. Kwiatkowska, D. Parker, and A. Simaitis. Verifying team formation protocols with probabilistic model checking. In *Proc. 12th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA XII 2011)*, pages 190–297. Springer, 2011.

[34] T. Chen, M. Kwiatkowska, A. Simaitis, and C. Wiltsche. Synthesis for multi-objective stochastic games: An application to autonomous urban driving. In *Proc. 10th International Conference on Quantitative Evaluation of Systems (QEST'13)*, pages 322–337. Springer, 2013.

[35] F. Ciesinski and C. Baier. Liquor: A tool for qualitative and quantitative linear time analysis of reactive systems. In *Proc. 3rd International Conference on Quantitative Evaluation of Systems (QEST'06)*, pages 131–132. IEEE, 2006.

[36] F. Ciesinski, C. Baier, M. Groesser, and J. Klein. Reduction techniques for model checking Markov decision processes. In *Proc. 5th International Conference on Quantitative Evaluation of Systems (QEST'08)*, pages 45–54. IEEE, 2008.

[37] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *14th International Conference on Computer Aided Verification (CAV'02)*, pages 359–364. Springer, 2002.

[38] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logics. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[39] E. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

[40] E. Clarke and E. A. Emerson. *Design and synthesis of synchronization skeletons using branching time temporal logic*. Springer, 1982.

[41] A. Condon. The complexity of stochastic games. *Information and Computation*, 96(2):203–224, 1992.

[42] A. Condon. On algorithms for simple stochastic games. *Advances in computational complexity theory*, 13:51–73, 1993.

[43] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, et al. *Introduction to algorithms*, volume 2. MIT Press, 2001.

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

[45] C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *Journal of the ACM*, 42(4):857–907, 1995.

[46] F. Dannenberg, E. M. Hahn, and M. Kwiatkowska. Computing cumulative rewards using fast adaptive uniformisation. In *Proc. 11th Conference on Computational Methods in Systems Biology (CMSB'13)*, pages 33–49. Springer, 2013.

[47] L. de Alfaro. *Formal Verification of Probabilistic Systems*. PhD thesis, Stanford University, 1997.

[48] C. Dehnert, N. Jansen, R. Wimmer, E. Abraham, and J.-P. Katoen. Fast debugging of PRISM models. In *Proc. 12th International Symposium on Automated Technology for Verification and Analysis (ATVA'12)*, pages 146–162. Springer, 2014.

[49] K. Draeger, V. Forejt, M. Kwiatkowska, D. Parker, and M. Ujma. Permissive controller synthesis for probabilistic systems. In *Proc. 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14)*, pages 531–546. Springer, 2014.

[50] K. Draeger, V. Forejt, M. Kwiatkowska, D. Parker, and M. Ujma. Permissive controller synthesis for probabilistic systems. *Logical Methods in Computer Science*, 2015. (Accepted for publication with minor corrections).

[51] M. Duflot, M. Kwiatkowska, G. Norman, and D. Parker. A formal analysis of Bluetooth device discovery. *International Journal on Software Tools for Technology Transfer*, 8(6):621–632, 2006.

[52] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model evolution by run-time parameter adaptation. In *Proc. 31st International Conference on Software Engineering (ICSE'09)*, pages 111–121. IEEE, 2009.

[53] K. Etessami, M. Kwiatkowska, M. Vardi, and M. Yannakakis. Multi-objective model checking of Markov decision processes. *Logical Methods in Computer Science*, 4(4):1–21, 2008.

[54] L. Feng, M. Kwiatkowska, and D. Parker. Automated learning of probabilistic assumptions for compositional reasoning. In *Proc. 14th International Conference on Fundamental Approaches to Software Engineering (FASE'11)*, pages 2–17. Springer, 2011.

[55] J. Filar and K. Vrieze. *Competitive Markov Decision Processes*. Springer, 1997.

[56] A. Filieri and C. Ghezzi. Further steps towards efficient runtime verification: Handling probabilistic cost models. In *Software Engineering: Rigorous and Agile Approaches (FormSERA'12)*, pages 2–8. IEEE, 2012.

[57] A. Filieri, C. Ghezzi, A. Leva, and M. Maggio. Self-adaptive software meets control theory: A preliminary approach supporting reliability requirements. In *26th International Conference on Automated Software Engineering (ASE'11)*, pages 283–292. IEEE, 2011.

[58] A. Filieri, C. Ghezzi, and G. Tamburrelli. Run-time efficient probabilistic model checking. In *Proc. 33rd International Conference on Software Engineering (ICSE'11)*, pages 341–350. ACM, 2011.

[59] V. Forejt, M. Kwiatkowska, G. Norman, and D. Parker. Automated verification techniques for probabilistic systems. In *Formal Methods for Eternal Networked Software Systems (SFM'11)*, pages 53–113. Springer, 2011.

[60] V. Forejt, M. Kwiatkowska, G. Norman, D. Parker, and H. Qu. Quantitative multi-objective verification for probabilistic systems. In *Proc. 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'11)*, pages 112–127. Springer, 2011.

[61] V. Forejt, M. Kwiatkowska, D. Parker, H. Qu, and M. Ujma. Incremental runtime verification of probabilistic systems. In *Proc. 3rd International Conference on Runtime Verification (RV'12)*, pages 314–319. Springer, 2012.

[62] V. Forejt, M. Kwiatkowska, D. Parker, H. Qu, and M. Ujma. Incremental runtime verification of probabilistic systems. Technical Report RR-12-05, Department of Computer Science, University of Oxford, 2012.

[63] M. R. Garey, R. L. Graham, and D. S. Johnson. Some NP-complete geometric problems. In *Proc. 8th Symposium on Theory of Computing (STOC'76)*, pages 10–22. ACM, 1976.

[64] M. Gaston and M. desJardins. Agent-organized networks for dynamic team formation. In *Proc. 4th International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'05)*, pages 230–237. ACM, 2005.

[65] S. Gerasimou, R. Calinescu, and A. Banks. Efficient runtime quantitative verification using caching, lookahead, and nearly-optimal reconfiguration. In *Proc. 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14)*, pages 115–124. ACM, 2014.

[66] C. Ghezzi, M. Pezze, and G. Tamburrelli. Adaptive REST applications via model inference and probabilistic model checking. In *International Symposium on Integrated Network Management (IM'13)*, pages 1376–1382. IEEE, 2013.

[67] S. Haddad and B. Monmege. Reachability in MDPs: Refining convergence of value iteration. In *Proc. 8th International Workshop on Reachability Problems (RP'14)*, pages 125–137. Springer, 2014.

[68] E. M. Hahn, H. Hermanns, B. Wachter, and L. Zhang. PARAM: A model checker for parametric Markov models. In *Proc. 22nd International Conference on Computer Aided Verification (CAV'10)*, pages 660–664. Springer, 2010.

[69] E. A. Hansen and S. Zilberstein. LAOâĹŮ: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129(1):35–62, 2001.

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

[71] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

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

[73] T. A Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *Proc. 10th International SPIN Workshop (SPIN'03)*, pages 235–239. Springer, 2003.

[74] T. Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet. Approximate probabilistic model checking. In *Proc. 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, pages 307–329. Springer, 2004.

[75] H. Hildmann and F. Saffre. Influence of variable supply and load flexibility on demand-side management. In *Proc. 8th International Conference on the European Energy Market (EEM'11)*, pages 63–68. IEEE, 2011.

[76] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[77] N. Jansen, E. Abraham, B. Zajzon, R. Wimmer, J. Schuster, J.-P. Katoen, and B. Becker. Symbolic counterexample generation for discrete-time Markov chains. In *Proc. 9th International Symposium on Formal Aspects of Component Software (FACS'13)*, pages 134–151. Springer, 2013.

[78] N. Karmarkar. A new polynomial-time algorithm for linear programming. In *Proc. 16th Symposium on Theory of computing (STOC'84)*, pages 302–311. ACM, 1984.

[79] R. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Springer, 1972.

[80] J.-P. Katoen, M. Khattri, and I. Zapreev. A Markov reward model checker. In *Proc. 2nd International Conference on Quantitative Evaluation of Systems (QEST'05)*, pages 243–244. IEEE, 2005.

[81] M. Kattenbelt, M. Kwiatkowska, G. Norman, and D. Parker. A game-based abstraction-refinement framework for Markov decision processes. *Formal Methods in System Design*, 36(3):246–280, 2010.

[82] J. Kemeny, J. Snell, and A. Knapp. *Denumerable Markov Chains*. D. Van Nostrand Company, 1966.

[83] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[84] A. Kolobov, Mausam, D. S. Weld, and H. Geffner. Heuristic search for generalized stochastic shortest path MDPs. In *Proc. 21th International Conference on Automated Planning and Scheduling (ICAPS'11)*, pages 130–137. AAAI, 2011.

[85] O. Kupferman and M. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

[86] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic model checking and power-aware computing. In *Proc. 7th International Workshop on Performability Modeling of Computer and Communication Systems (PMCCS'05)*, pages 6–9, 2005.

[87] M. Kwiatkowska, G. Norman, and D. Parker. Analysis of a gossip protocol in PRISM. *ACM SIGMETRICS Performance Evaluation Review*, 36(3):17–22, 2008.

[88] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, pages 585–591. Springer, 2011.

[89] M. Kwiatkowska, G. Norman, and D. Parker. The PRISM benchmark suite. In *Proc. 9th International Conference on Quantitative Evaluation of Systems (QEST'12)*, pages 203–204. IEEE, 2012.

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

[91] M. Kwiatkowska, G. Norman, and R. Segala. Automated verification of a randomized distributed consensus protocol using Cadence SMV and PRISM. In *Proc. 13th International Conference on Computer Aided Verification (CAV'01)*, pages 194–206. Springer, 2001.

[92] M. Kwiatkowska, G. Norman, and J. Sproston. Probabilistic model checking of the IEEE 802.11 wireless local area network protocol. In *Proc. 2nd Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM/PROBMIV'02)*, pages 169–187. Springer, 2002.

[93] M. Kwiatkowska, G. Norman, and J. Sproston. Probabilistic model checking of deadline properties in the IEEE 1394 FireWire root contention protocol. *Formal Aspects of Computing*, 14(3):295–318, 2003.

[94] M. Kwiatkowska, D. Parker, and H. Qu. Incremental quantitative verification for Markov decision processes. In *Proc. 41st International Conference on Dependable Systems and Networks (DSN-PDS'11)*, pages 359–370. IEEE, 2011.

[95] M. Kwiatkowska, D. Parker, H. Qu, and M. Ujma. On incremental quantitative verification for probabilistic systems. In *HOWARD-60: A Festschrift on the Occasion of Howard Barringer's 60th Birthday*, pages 245–257. Easychair, 2014.

[96] M. Lahijanian, J. Wasniewski, S. B. Andersson, and C. Belta. Motion planning and control from temporal logic specifications with probabilistic satisfaction guarantees. In *Proc. International Conference on Robotics and Automation (ICRA'10)*, pages 3227–3232. IEEE, 2010.

[97] D. Lehmann and S. Shelah. Reasoning with time and chance. *Information and Control*, 53(3):165–198, 1982.

[98] W.S. Levine. *The Control Handbook*. Electrical Engineering Handbook. Taylor & Francis, 1996.

[99] Masuam and A. Kolobov. *Planning with Markov decision processes: An AI Perspective*. Morgan & Claypool, 2012.

[100] A. McIver and C. Morgan. Results on the quantitative mu-calculus qMu. *ACM Transactions on Computational Logic*, 8(1), 2007.

[101] H. B. McMahan, M. Likhachev, and G. J. Gordon. Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In *Proc. 22nd International Conference on Machine learning (ICML'05)*, pages 569–576. ACM, 2005.

[102] A. Naskos, E. Stachtiari, A. Gounaris, P. Katsaros, D. Tsoumakos, I. Konstantinou, and S. Sioutas. Cloud elasticity using probabilistic model checking. *arXiv preprint arXiv:1405.4699*, 2014.

[103] C. H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Dover Publications, 1998.

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

[105] A. Pnueli. The temporal logic of programs. In *Proc. 18th Annual Symposium on Foundations of Computer Science (FOCS'77)*, pages 46–57. IEEE, 1977.

[106] A. Pnueli. On the extremely fair treatment of probabilistic algorithms. In *Proc. 15th Symposium on Theory of Computing (STOC'83)*, pages 278–290. ACM, 1983.

[107] M. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley and Sons, 1994.

[108] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th International Symposium on Programming*, pages 337–351. Springer, 1982.

[109] F. Saffre and A. Simaitis. Host selection through collective decision. *ACM Transactions on Autonomous and Adaptive Systems*, 7(1):4:1–4:16, 2011.

[110] S. Sanner, R. Goetschalckx, K. Driessens, and G. Shani. Bayesian real-time dynamic programming. In *Proc. 21st International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 1784–1789. AAAI, 2009.

[111] L. S. Shapley. Stochastic games. *Proceedings of the National Academy of Sciences of the United States of America*, 39(10):1095, 1953.

[112] V. Shmatikov. Probabilistic model checking of an anonymity system. *Journal of Computer Security*, 12(3/4):355–377, 2004.

[113] A. P. Sistla and E Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

[114] G. Steel. Formal analysis of PIN block attacks. *Theoretical Computer Science*, 367(1-2):257–270, 2006.

[115] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[116] S. Tripakis. *The analysis of timed systems in practice*. PhD thesis, Universite Joseph Fourier, 1998.

[117] A. Ulusoy, T. Wongpiromsarn, and C. Belta. Incremental control synthesis in probabilistic environments with temporal logic constraints. In *Proc. 51st Conference on Decision and Control (CDC'12)*, pages 7658–7663. IEEE, 2012.

[118] A. Ulusoy, T. Wongpiromsarn, and C. Belta. Incremental controller synthesis in probabilistic environments with temporal logic constraints. *International Journal of Robotics Research*, 33(8):1130–1144, 2014.

[119] M. Vardi. Automatic verification of probabilistic concurrent finite state programs. In *Proc. 26th Annual Symposium on Foundations of Computer Science (FOCS'85)*, pages 327–338. IEEE, 1985.

[120] Y. Vorobeychik and S. P. Singh. Computing stackelberg equilibria in discounted stochastic games. In *Proc. 26th Conference on Artificial Intelligence (AAAI'12)*, pages 1478–1484. AAAI, 2012.

[121] R. Wimmer, N. Jansen, E. Abraham, B. Becker, and J.-P. Katoen. Minimal critical subsystems for discrete-time Markov models. In *Proc. 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12)*, pages 299–314. Springer, 2012.

[122] T. Wongpiromsarn, A. Ulusoy, C. Belta, E. Frazzoli, and D. Rus. Incremental temporal logic synthesis of control policies for robots interacting with dynamic agents. In *Proc. 25th International Conference on Intelligent Robots and Systems (IROS'12)*, pages 229–236. IEEE, 2012.

[123] T. Wongpiromsarn, A. Ulusoy, C. Belta, E. Frazzoli, and D. Rus. Incremental synthesis of control policies for heterogeneous multi-agent systems with linear temporal logic specifications. In *Proc. International Conference on Robotics and Automation (ICRA'13)*, pages 5011–5018. IEEE, 2013.

[124] H. Younes and R. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In *Proc. 14th International Conference on Computer Aided Verification (CAV'02)*, pages 223–235. Springer, 2002.

[125] VisualVM profiler website, `http://visualvm.java.net/`.

[126] PRISM models for Chapter 5, `http://www.prismmodelchecker.org/files/thesismujma/#incremental`.

[127] PRISM models for Chapter 6, `http://www.prismmodelchecker.org/files/thesismujma/#permissive`.

[128] PRISM models for Chapter 7, `http://www.prismmodelchecker.org/files/thesismujma/#heuristics`.

[129] PRISM models, `http://www.prismmodelchecker.org/files/thesismujma/`.

[130] StockPriceViewer, `http://code.google.com/p/android-stock-price-viewer/`.

[131] CPLEX solver website, `http://www.ilog.com/products/cplex/`.

[132] Gurobi solver website, `http://www.gurobi.com/`.