Quantitative Verification and Synthesis

Christian von Essen

To cite this version:

Christian von Essen. Quantitative Verification and Synthesis. Numerical Analysis [cs.NA]. Université de Grenoble, 2014. English. NNT : 2014GRENM090 . tel-01548501

HAL Id: tel-01548501
https://theses.hal.science/tel-01548501

Submitted on 27 Jun 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial :

Présentée par

Christian von Essen

Thèse dirigée par **Saddek Bensalem**
et codirigée par **Barbara Jobstmann**

préparée au sein **du Laboratoire VERIMAG**
et de **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique (EDMSTII)**

Quantitative Verification and Synthesis

Thèse soutenue publiquement le **28 avril 2014**,
devant le jury composé de :

Dr. Alain Girault
INRIA Rhône-Alpes, Président
Prof. Marta Zofia Kwiatkowska
University of Oxford, Rapporteur
Prof. Jean-François Raskin
Université Libre de Bruxelles, Rapporteur
Prof. Ahmed Bouajjani
Université Paris Diderot, Examinateur
Prof. Saddek Bensalem
Université Joseph Fourier, Verimag, Directeur de thèse
Dr. Barbara Jobstmann
EPFL, Verimag, Jasper, Co-Directeur de thèse

Once upon a time, in a land far far away…

Abstract

This thesis contributes to the theoretical study and application of quantitative verification and synthesis.

We first study strategies that optimize the ratio of two rewards in MDPs. The goal is the synthesis of efficient controllers in probabilistic environments. We prove that deterministic and memoryless strategies are sufficient. Based on these results we suggest 3 algorithms to treat explicitly encoded models. Our evaluation of these algorithms shows that one of these is clearly faster than the others. To extend its scope, we propose and implement a symbolic variant based on binary decision diagrams, and show that it cope with millions of states.

Second, we study the problem of program repair from a quantitative perspective. This leads to a reformulation of program repair with the requirement that only faulty runs of the program be changed. We study the limitations of this approach and show how we can relax the new requirement. We devise and implement an algorithm to automatically find repairs, and show that it improves the changes made to programs.

Third, we study a novel approach to a quantitative verification and synthesis framework. In this, verification and synthesis work in tandem to analyze the quality of a controller with respect to, e.g., robustness against modeling errors. We also include the possibility to approximate the Pareto curve that emerges from combining the model with multiple rewards. This allows us to both study the trade-offs inherent in the system and choose a configuration to our liking. We apply our framework to several case studies. The major case study is concerned with the currently proposed next generation airborne collision avoidance system (ACAS X). We use our framework to help analyze the design space of the system and to validate the controller as currently under investigation by the FAA. In particular, we contribute analysis via PCTL and stochastic model checking to add to the confidence in the controller.

Acknowledgements

I would first of all like to thank my advisers Barbara Jobstmann and Saddek Bensalem. Barbara was always there to support me in my research and otherwise. Saddek gave me the freedom to follow my ideas and find applications for them. I am also very grateful to Marta Kwiatkowska, Jean-François Raskin and Ahmed Bouajjani and Alain Girault who, in their role as my jury, took on the task of reading this lengthy document and attending the defense. I hope I can pay their effort with an interesting thesis and an exciting presentation.

I want to thank the whole former and present Verimag group for making my stay as fun and entertaining as it was. Special thanks go to Jannik Dreier, Mathilde Duclos and Julien Le Guen, for welcoming me in Grenoble and many entertaining exploits.

I want to say thank you to Aditya Nori and Sriram Rajamani for welcoming me in India and at Microsoft Research. They allowed me to explore the application of formal methods to sampling and the Indian cuisine. A special thanks goes out to Dimitra Giannakopoulou and the whole NASA Ames team. They allowed me to see that there is hope for the application of formal methods to exciting and important real-life topics. Dimitra's enthusiasm is contagious, her belief in me incredibly supporting and her people skills unsurpassed. She taught me that working over two continents poses no big challenge, if only the motivation is high enough.

I want to thank my family for supporting me in all my decisions and for always being welcoming. They gave me the feeling that I am able to do the things to which I aspire. I want to thank my wife Elena for putting up with many a grumpy mood and for much moral support and belay service (I bet the feeling is mutual). She is an inspiration in many ways, and I wish I was as tough as her.

Christian

Contents

1 Introduction **1**
1.1 On quantitative verification and synthesis 1
1.2 Relation to artificial intelligence 4
1.3 Outline and contributions 4
1.4 Preliminaries 6
1.5 State of the art 28
1.6 Tools 33

2 Efficient Systems in Probabilistic Environments **35**
2.1 Introduction 35
2.2 The system and its environment 37
2.3 Analysis 44
2.4 Algorithms 54
2.5 Symbolic implementation 68
2.6 Conclusion 83

3 Program repair without regret **85**
3.1 Introduction 85
3.2 On languages 86
3.3 Example 88
3.4 Repair 90
3.5 Discussion and limitations 98
3.6 Empirical results 105
3.7 Future work and conclusions 111

4 Quantitative verification and synthesis framework **113**
4.1 Introduction 113
4.2 Implementation description 115
4.3 Discretization of spaces and distributions 121
4.4 Specifying models in Java 125
4.5 Approximating Pareto curves 135

4.6 Case studies 141

5 Analyzing the Next Generation Airborne Collision Avoidance System 159

5.1 Introduction 159

5.2 The ACAS X system 161

5.3 Verification 170

5.4 ACAS X design challenges 176

5.5 Implementation 183

5.6 Conclusions and Future Work 184

6 Conclusion 185

6.1 Future work 187

1 Introduction

In which we introduce our subject matter, study the difference between quantitative and qualitative, give an overview of related work and drive a poor little robot crazy.

Résumé

Ce chapitre est une introduction dans la thèse. Nous considérerons la motivation de la vérification et la synthèse quantitatives. Ensuite nous montrerons les relations de ce sujet avec l'intelligence artificielle. Finallement nous indroduirons notation appliquée dans le cadre de ce travail et motivée par un petit robot, qui doit nettoyer un gros appartement.

1.1 On quantitative verification and synthesis

Synthesis. Synthesis aims to automatically generate a program or system from a higher-level specification. These specifications leave a lot of details open, and it is the synthesizer's task to resolve the non-determinism such that the specification is fulfilled. This higher level allows a programmer or designer to express his wishes concisely while leaving implementation details to an assistant as willing as he is stupid (the computer). This form of abstraction becomes ever more important as the programs that we write become ever more complex because of the arrival of multi-processor systems, heterogeneous systems, pressing security questions, ever more computers in life-critical systems etc. Programs also influence the lives of ever more people, so ever more people should be able to influence programs. A high-level language and a synthesizer might be able to lower the bar of creating custom programs. Take Excel as

an example. it allows many users that do not know how to program to create spread-sheets and now special-purpose programs customized to their needs.

Synthesis looks promising in the area of embedded systems. Firstly, these systems are often small and not equipped for interactive development and hence debugging becomes especially challenging. Secondly, embedded systems are the most prevalent computer systems today, ranging from thermometers to vehicles on Mars. Finally, embedded systems, by their very nature, have to be customized to each new kind of hardware they entail. Removing unnecessary bugs altogether is therefore desirable and cost-effective.

Qualitative synthesis. Specifications are usually given with qualitative meaning, i.e., they classify systems either as good (meaning the system satisfies the specification) or as bad (meaning the system violates the specification). In this thesis we explore how we can add more information to this process. We call this "quantitative synthesis" Quantitative specifications assign to each system a value that provides additional information.

Quantitative synthesis. Manichaeism was a religion that postulated that the world is the battle-ground for good and evil — black and white. To us, it appears that there are many shades of gray — quantitative information is important to us in the real world. We can either just pass an exam (qualitative), or pass it well (quantitative). A thesis can be acceptable or cum laude (quantitative), but both are enough for graduation (qualitative). Traditionally, quantitative techniques have been used to analyze properties like response time, throughput, or reliability (cf. [dA97, Hav98, BK08, KNP09]).

Recently, quantitative reasoning has been used to state preference relations between systems satisfying the same qualitative specification [BCHJ09]. For example, we can compare systems with respect to robustness, i.e., how reasonable they behave under unexpected behaviors of their environments [BGHJ09]. A preference relation between systems is particularly useful in synthesis, because it allows the user to guide the synthesizer and ask for "the best" system. In many settings a better system comes with a higher price. For example, consider an assembly line that can be operated in several speeds i.e., the number of units produced per time unit. We would prefer a controller that produces as many units as possible. However, running the line in a faster mode increases the power consumption and the probability to fail, resulting in higher repair costs.

We want to synthesize reactive systems, i.e., systems that react to signals from their environment indefinitely. We are looking for a system that behaves optimal globally, i.e., on all possible behaviors of its environment. Having defined a local evaluation criterion, e.g., the efficiency of a system on a single environment input, we need to define a global evaluation criterion. There are several possible ways, e.g., worst case, best case, or average. In the worst and best-case scenarios we assume that our system operates in an antagonistic or in a cooperative environment. To define the average, we need to define a probability distribution. In our case, we use probability distributions over the behavior of the environment: assembly lines need repair randomly, network protocols have randomly behaving participants, servers receive random requests, etc. Modeling environments with probabilistic behavior allows us to assume that the environment is not hostile, i.e., it is not in fact trying to do its worst. Probabilistic modeling also allows us to encode knowledge or expectations on environment behavior. Embedded systems sometimes are only required to operate in given conditions that we can model probabilistically. A server, for example, is only required to work given an expected average number of requests, while a denial of service attack lies outside of its specification[1]. A different component takes care of shielding the server in case of such an attack. Lastly, assuming probabilistic behavior makes quantitative synthesis questions often more tractable than their qualitative counterparts, admitting synthesis algorithms with expected polynomial instead of exponential run-time.

This thesis also takes the idea of quantitative synthesis further. While it is obvious that any cost or reward measure can be used for quantitative specifications, we also explore how we can use additional information that is not directly encoded in the qualitative specification to guide the synthesis process. We focus on the use of the semantics of the program as quantitative information in program repair. Instead of just repairing a program according to a qualitative specification, we try to find a semantically close repair.

Finally, this thesis shows the power of quantitative formal methods. We demonstrate how we can use an embedded domain specific language in a common programming language to describe a reactive system. We then show how we can use this language to model, synthesize and analyze a real-life collision

[1] We can still model a denial of service attack. We assume that a DDOS attack comes with a small probability. If it comes, the environment suddenly changes its behavior drastically.

avoidance system for airplanes.

1.2 Relation to artificial intelligence

Synthesis, and particularly the techniques and ideas used for quantitative synthesis and verification, is also a topic in the field of artificial intelligence (among other fields, e.g., operations research). Named *planning*, artificial intelligence tries to find (i.e,. synthesize) optimal decisions. These vary greatly, from optimal treatment for patients or motion planning for robots. One prominent field that is very close to what is done in this thesis is called reinforcment learning, which is concerned with defining the decisions for an agent so as to optimize a given reward.

These two fields can learn a lot from each other, and proliferation and collaboration have started. For example, [MLOP07, HvdHvR09, KGFP09] have proposed to use linear temporal logic[2] to specify desirable sequences of states, instead of just desirable states, as is common in artificial intelligence. In the other direction, probabilistic model checking has adapted and improved techniques originally developed for reinforcement learning [CBGK08]. Artificial intelligence is used to support program verification in, e.g., [SNA12, NR11, BN11]. In addition, artificial intelligence often tries to find controllers in a world not fully known, while there have been no practical advances from the field of verification and synthesis, yet. There are theoretical results [FGL12] and experimental implementations [BCW$^+$10], but it is not accepted practice.

To summarize, we believe that the budding collaboration in these fields is promising, and hope to see closer collaboration.

1.3 Outline and contributions

We will now outline the thesis and summarize the contributions of each chapter. Most chapters are based on published, peer-reviewed work and we will later point out which part of each chapter is unpublished.

- Rest of Chapter 1: We will continue with necessary preliminaries and then discuss both the theoretical and the practical state of the art of quantitative verification and synthesis.

[2] A specification language stemming from the field of formal methods, and discussed later.

- Chapter 2: This chapter contributes theoretical results for the ratio objective for Markov decision processes (MDPs) as well as practical results for quantitative synthesis.

    1. We show that the ratio objective is well-defined for MDPs, and contribute three algorithms on explicitly encoded MDPs: two algorithms based on linear programming and one based on policy iteration. We also implement the algorithm based on binary decision diagrams in the probabilistic model checking tool PRISM [KNP11].
    2. We also present a framework to automatically construct a system with an efficient average-case behavior with respect to a reward and a cost model in a probabilistic environment. To the best of our knowledge, this is the first approach that allows synthesizing efficient systems automatically. We analyze our framework, proving that we can indeed find efficient systems. This analysis is the foundation for the algorithms.

- Chapter 3: In this chapter, we refine the definition of repair of (reactive) programs. We achieve this by requiring that a repair be semantically close to the original program. By this, we show that quantitative verification and synthesis does not necessarily mean rewards or probabilities.

  We analyze the limits of our approach, and explore why a repair might be impossible. We also provide several examples, showing that our new notion of repair has advantages in practice. Finally, we provide an algorithm and a prototype implementation that finds a repair, if one exists.

- Chapter 4: In this chapter, we present a novel framework designed for quantitative and qualitative synthesis and analysis, in which the synthesized strategies can be model checked in different models/assumptions. We also propose a novel way for describing models in a Java embedded domain specific language. We further show that we can approximate Pareto curves of MDPs with many kinds of rewards via value iteration, which allows us to side-step linear programming. We apply this system to several case studies taken from the field of automotive engineering and from artificial intelligence.

- Chapter 5: We apply the framework defined in Chapter 4 to a real-world case study. We analyze and contribute to the design of the next-generation airborne collision avoidance system currently under review at the Federal Aviation Administration.

New results versus published results

- Chapter 2 is based on [vEJ11, vEJ12]. New and unpublished are the symbolic algorithms and the proof that shows that none of the two policy algorithms that we propose is necessarily better than the other.
- Chapter 3 is based on [vEJ13]. This thesis adds further analysis w.r.t optimal repair, and also suggests other methods to define close repairs.
- Chapter 4 is completely new and unpublished
- Chapter 5 is based on [vEG14]

1.4 Preliminaries

As we go from topics well known to topics more esoteric, our definitions and explanations will go from the brief to the extensive. This thesis needs to assume some level of familiarity with basic math. It assumes that concepts such as sequences, tuples, sets, functions and first-order logic are understood. Other concepts, like Markov decision processes, are not taken for granted. The subject of this thesis is dwelt upon, as is customary.

Mathy basics

We consider zero to be a very natural number, and therefore define $\mathbb{N} = \{0, 1, 2, \ldots\}$. We refrain from defining the set of real numbers and just denote it by $\mathbb{R}$. Let A, B be sets. Then $A \times B = \{(a, b) \mid a \in A, b \in B\}$ denotes the *cross product* or *Cartesian product* of A and B. By $2^A = \{A' \mid A' \subseteq A\}$ we denote the *powerset* of A, i.e., the set of subsets of A. Let $\emptyset \neq A_0, \ldots, A_n \subseteq A$ be non-empty subsets of A. If the subsets cover A and are pairwise disjoint, i.e., $\bigcup_{0 \leq i \leq n} A_i = A$ and $\forall 0 \leq i \neq j \leq n : A_i \cap A_j = \emptyset$, then the A_i are said to form a *partition* of A, and we call the A_i *block*s. We say that a partition A is finer than a partition A' if all blocks $A_i \in A$ are contained in a block

$A'_j \in A'$, i.e., $A_i \subseteq A'_j$. Given functions $f : A \to B$ and $g : B \to C$ we denote by $g \circ f : A \to C$ the composition of f and g, i.e., $(g \circ f)(a) = g(f(a))$.

The set of n-dimensional vectors over a set A is defined as $A^n := \{(a_1, ..., a_n) \mid a_i \in A, 1 \leq i \leq n\}$. For a vector $a \in A^n$ we denote by a_i the ith component of a, starting from 1, i.e., $a = (a_1, \ldots, a_n)$. By $a < b$ for two vectors $a, b \in A^n$ we denote the fact that there is a component $1 \leq i \leq n$ such that $a_i < b_i$ and that for all $1 \leq j \leq n$ we have $a_j \leq b_j$.

A *σ-algebra* over a set A is a set $\mathcal{F} \subseteq 2^A$ such that (i) $\emptyset \in \mathcal{F}$, (ii) $E \in \mathcal{F}$ implies $A \setminus E \in \mathcal{F}$ for any $E \in \mathcal{F}$, and (iii) the union of any countable set of elements of $\mathcal{F}$ $E_1, E_2, \dots \in \mathcal{F}$ is also in $\mathcal{F}$, i.e., $\bigcup E_i \in \mathcal{F}$. Let $\mathcal{F}, \mathcal{F}'$ be σ-algebras over A and A', respectively. A function $f : A \to A'$ is called *measurable* if the pre-image of every element in $\mathcal{F}'$ is an element of $\mathcal{F}$, i.e., if $f^{-1}(E) := \{a \in A | f(a) \in E\} \in \mathcal{F}$ for all $E \in \mathcal{F}'$.

Probability theory. A *probability space* is defined by a tuple $\mathcal{P} := (\Omega, \mathcal{F}, \mu)$, where Ω is the set of *outcomes or samples*, $\mathcal{F} \subseteq 2^\Omega$ is a σ-algebra defining the set of *measurable events*, and $\mu : \mathcal{F} \to [0, 1]$ is a *probability measure* assigning a probability to each event such that $\mu(\Omega) = 1$ and for each countable set $E_1, E_2, \dots \in \mathcal{F}$ of disjoint events we have $\mu(\bigcup E_i) = \sum \mu(E_i)$. Given a measurable function $f : \Omega \to \mathbb{R} \cup \{+\infty, -\infty\}$, we use $\mathbb{E}_\mathcal{P}[f]$ to denote the expected value of f under μ, i.e.,

$$\mathbb{E}_\mathcal{P}[f] = \int_\Omega f \; d\mu.$$

If $\mathcal{P}$ is clear from the context, then we drop the subscript or replace it with the structure that defines $\mathcal{P}$. The integral used here is the Lebesgue integral, which is commonly used to define the expected value of a random variable.

By $\mathrm{D}(X)$ we denote the set of probability measures over set X. For a finite set X, $\mathrm{D}(X) = \{f : X \to [0, 1] \mid \sum_{x \in X} f(x) = 1\}$ holds.

Words. Let A be a finite set. Then $A^n := \{a_0 a_1 \ldots a_{n-1} \mid a_i \in A\}$ denotes the set of *words over* A of length $n \in \mathbb{N}$. The *empty word* is denoted by ϵ, and therefore $A^0 = \{\epsilon\}$. The set of all words of finite length is denoted as $A^* := \bigcup_{n \in \mathbb{N}} A^n$. By $A^\omega := \{a_0 a_1 \cdots \mid a_i \in A\}$ we denote the set of infinite words. Given two words $v \in A^*, w \in A^* \cup A^\omega$, we denote by $v \leq w$ the fact that v is a *prefix* of w. Given a word w, we denote by w_i the letter at position i, where we start counting at 0. Further, by $w_{\geq i}$ we denote the postfix of w starting at position i, i.e., $w_i w_{i+1} \cdots$. Analogously, by $w_{>i}$ we denote the postfix of w starting after position i, i.e., $w_{i+1} w_{i+2} \cdots$, by $w_{<i}$ we denote the

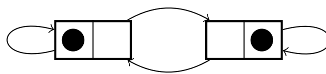


Figure 1.1: Robot in a two room apartment

prefix of w ending with the ith letter, i.e., $w_0 \cdots w_{i-1}$, and by $w_{\leq i}$ we denote the prefix ending with the $i+1$th letter, i.e., $w_0 \cdots w_i$.

Structures

Transition Systems. In this thesis, we concern ourselves with systems that can take on a finite number of states. The most basic of such systems is a *transition system*.

Definition 1.1 (Transition System) Let $\mathrm{S} \neq \emptyset$ be a finite set, and let $\Delta \subseteq \mathrm{S} \times \mathrm{S}$ be a relation between elements of S such that $\forall \mathrm{s} \in \mathrm{S} \, \exists \mathrm{s}' \in \mathrm{S} : (\mathrm{s}, \mathrm{s}') \in \Delta$. Let finally $\mathrm{s}_0 \in \mathrm{S}$ be an element of S. Then $\mathrm{L} = (\mathrm{S}, \Delta, \mathrm{s}_0)$ is called a *transition system* where S are the states of the transition system, Δ is the transition relation and s_0 is the initial state. The transition relation dictates that the system can move from state $\mathrm{s} \in \mathrm{S}$ to state $\mathrm{s}' \in \mathrm{S}$ if $(\mathrm{s}, \mathrm{s}') \in \Delta$. For convenience, we assume that each state has at least one outgoing transition, i.e., $\forall \mathrm{s} \in \mathrm{S} \, \exists \mathrm{s}' \in \mathrm{S} : (\mathrm{s}, \mathrm{s}') \in \Delta$.

We use transition systems to represent the structures in this thesis.

Example 1.1 Consider Figure 1.1 as an example in which a robot can move between two rooms of a flat[3]. A state is defined by the location of the robot (i.e., left or right room). We depict the two rooms by two squares, while we indicate the location of the robot by a black circle. In each step, the robot can move from one room to the other, or remain in the same room. The arrows between the two states in the figure depict how the robot can move. Formally, we model this with an TS with $\mathrm{S} = \{\mathrm{Left}, \mathrm{Right}\}$. The transition relation is $\Delta = \{(\mathrm{Left}, \mathrm{Left}), (\mathrm{Left}, \mathrm{Right}), (\mathrm{Right}, \mathrm{Left}), (\mathrm{Right}, \mathrm{Right})\}$

State $\mathrm{s}_0 \in \mathrm{S}$ acts as the start state, i.e., the state in which all *run*s of the transition system start.

[3] This and the following robot examples have been inspired by "Vacuum World" in [RN10].

Definition 1.2 (Runs) An infinite run of L starting in a state s_0 is an infinite word $\rho = s_0\, s_1 \cdots \in S^\omega$ over S whose first element is the state s_0 and whose adjacent pairs $s_i, s_{i+1}, i \geq 0$ are transitions in L, i.e., $(s_i, s_{i+1}) \in \Delta$. A finite run of L is any prefix of an infinite run of L. We denote by $\Omega_s^\omega(L)$ the set of infinite runs of L starting in s, and by $\Omega_s^*(L)$ the set of finite runs of L starting in s.

To illustrate these concepts, we extend the robot example with dirt. To be more precise, not only can a room now contain a robot, but a room can become dirty as well. If a room is dirty and the robot is in it, then the robot can clean the room.

Example 1.2 Formally, each room may be dirty or clean, i.e., the state space is now $S = \{\text{Left}, \text{Right}\} \times \{\text{Clean}, \text{Dirty}\} \times \{\text{Clean}, \text{Dirty}\}$. As before, the robot may move from one room to the other, or stay where it currently is. Additionally, it may now try to clean a room. This will remove any dirt from the room. We depict the full transition system except for self loops in Figure 1.2 on Page 10. An example of an infinite run would start in (Left, Clean, Clean), meaning that the robot is in the left room and there is no dirt. Now the neighbours come to visit, and their kid dirties both rooms, i.e., we move to state (Left, Dirty, Dirty). In our example, it is more important to clean the right room, so the robot first goes to that room, i.e., to state (Right, Dirty, Dirty), and cleans it, i.e., we move to state (Right, Dirty, Clean). Now the robot moves to the left room, and the neighbours behave themselves, i.e., we move to state (Left, Dirty, Clean). Now the robot cleans that room, i.e., we move to state (Left, Clean, Clean). We have now described the prefix of a run. Note that we cannot move between states arbitrarily. For example, we cannot move from state (Right, Dirty, Dirty) to state (Left, Clean, Clean). The run we have just described can now be extended to an infinite run by following the transitions of the graph.

Although the system is small (only 8 states) it is already not obvious how the robot should behave. An obvious task for controller synthesis is to find a controller for the robot so that it cleans dirty rooms. But this simple formulation still leaves a lot of choice. What is the robot supposed to do if there is no dirty room? It could either idle, or walk to the other room if it suspects that it is more likely that the other room will become dirty soon (because it

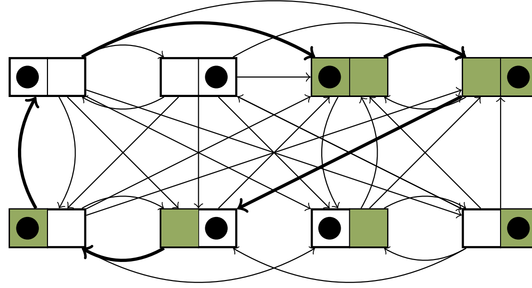


Figure 1.2: Transition system of a two room apartment; robot is black circle; colors square is dirty; self-loops are left out

is, for example, the room in which the kids play with the dog). Or it could be motivated to move to the other room because it is more important that the other room be clean (because it is the room in which we welcome guests). Or it has to visit the other room regularly to recharge its battery. In addition, we have a clear separation of roles in our cleaning robot example: we have the robot that tries to clean rooms on the one hand, and we have somebody or something deposing dirt in our flat. We will therefore now distinguish between the protagonist, which we want to control, and the antagonist (e.g., the environment, the kids or the dog), which tries to act against us.

Stochastic games. The most complex structure this thesis uses is a *stochastic game* [Sha53]. We define the as unifying framework for the later structures (i.e., two player games and Markov decision processes).

Definition 1.3 (Stochastic game) A stochastic game is a game played between two players (Player 0 and Player 1), in which chance plays a role as well. It is defined as $\mathcal{S} = (\mathrm{S}, \mathrm{S}_0, \mathrm{S}_1, \mathrm{S}_p, \mathrm{s}_0, \Delta, p)$ such that $(\mathrm{S}, \mathrm{s}_0, \Delta)$ is a transition system. Sets $\mathrm{S}_0, \mathrm{S}_1$ and S_p partition S to define to which player (Player 0, Player 1 or chance) a state belongs. Function $p : \mathrm{S}_p \to \mathrm{D}(\mathrm{S})$ defines the probability distribution used in a state in which chance rules. That is, $p(s)(s')$ is the probability of going from state s to state s'. Instead of $p(s)(s')$ we sometimes write $p(s' \mid s)$. We demand that $\forall\, \mathrm{s} \in \mathrm{S}_p\, \forall\, \mathrm{s}' \in \mathrm{S} : p(\mathrm{s}' \mid \mathrm{s}) > 0 \iff (\mathrm{s}, \mathrm{s}') \in \Delta$.

We will now redefine example Example 1.2 as a stochastic game.

Example 1.3 As example state space we use $\mathrm{S} = \{0, 1, c\} \times \{\mathrm{Left}, \mathrm{Right}\} \times \{\mathrm{Clean}, \mathrm{Dirty}\} \times \{\mathrm{Clean}, \mathrm{Dirty}\}$, where the first component indicates to which player the state belongs. In our example, Player 0 controls the robot (i.e.,

cleaning, idling and movement), Player 1 controls the addition of dirt and chance controls the unfortunate event that the vacuum cleaner is leaky and loses dirt, i.e., makes the room the robot occupies dirty. In addition, the players take turns. First, the environment (Player 1) decides if it wants to make a room dirty. Then the robot may vacuum clean a room or move. Then, chance may decide to make the room occupied by the robot dirty.

In Figure 1.3 we show a part of this stochastic game. States are depicted as in Figure 1.2, but now we also mark which state belongs to which player by either surrounding them with a box (Player 1), a circle (Player 0) or a diamond (Chance). Our run starts in the state in the upper left corner, in which it is the turn of Player 1 and both rooms are clean. Player 1 decides which (if any) of the rooms becomes dirty, and it is the turn of Player 0 (i.e., the robot) afterwards. For our example, we move to the state in which the left room is dirty. The robot may now decide to either do nothing, clean the room, or go to the other room. For our example we look at the two states in which the robot either cleaned the room or was just idling. After the turn of Player 0 it is now up to chance to decide if the robot leaks. In the state in which the robot did not do anything, chance has no choice but to move to the state in which the left room is still dirty. In the state in which the robot cleaned the room, chances transitions with a 99% chance to the state in which the room stays clean, and with a 1% chance dirties the just cleaned room.

Stochastic games assume that Player 0 and Player 1 play against each other. Often, we do not want to assume an antagonistic environment, but prefer a purely random environment. Such a stochastic game (i.e., one without states controlled by Player 1), will be described next.

Markov decision process. A *Markov decision process* (*MDP*) is a stochastic game in which the sole player plays against chance, but not another player. They have been used in game theory, operations research etc. for a long time. They have seen much success because the real world appears to us as a Markov decision process: we are a player in the game of life, taking actions based on what we perceive as our state (see for example [CN06] for examples of search engines and customer classification, [BHP97] for an approach to baseball and [KLD$^+$02] for an application in biology; see [Whi93] for a further survey, including agriculture, finance, sales and epidemics, among others). The outcome of our actions seems more or less random to us.

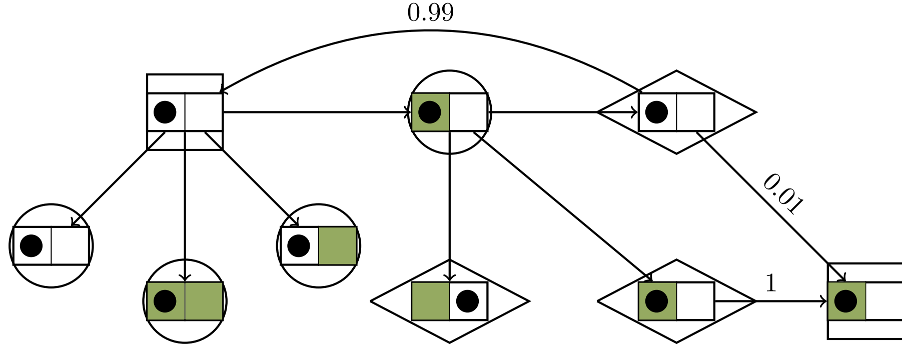


Figure 1.3: Example of a part of the stochastic cleaning robot game. States surrounded by a box belong to Player 1, states surrounded by a circle belong to Player 0, states surrounded by a diamond belong to chance. Numbers on edges going out from nodes belonging to chance are probabilities.

Definition 1.4 (Markov decision process) A Markov decision process is a stochastic game $\mathcal{S} = (\mathrm{S}, \mathrm{S}_0, \mathrm{S}_1, \mathrm{S}_p, \mathrm{s}_0, \Delta, p)$ in which $\mathrm{S}_1 = \emptyset$. Further, we demand that states following a player state be random states, and vice versa. Formally, we demand that the games forms a bipartite graph with S_0 forming one partition and S_p the other. That is, we demand that $\mathrm{s} \in \mathrm{S}_0 \wedge (\mathrm{s}, \mathrm{s}') \in \Delta \implies \mathrm{s}' \in \mathrm{S}_p$ and that $\mathrm{s} \in \mathrm{S}_p \wedge (\mathrm{s}, \mathrm{s}') \in \Delta \implies \mathrm{s}' \in \mathrm{S}_0$ hold. It is thus sufficient to define an MDP by the tuple $\mathcal{M} = (\mathrm{S}_0, \mathrm{s}_0, \overline{A}, \mathrm{S}_p, p)$, where $\overline{A} : \mathrm{S}_0 \to 2^{\mathrm{S}_p}$ defines what decisions are available to a player in what state. We then call S_p the *actions*, and $\overline{A}$ the *action function*. In addition, we often write $p(\mathrm{s}, a)(\mathrm{s}')$ or $p(\mathrm{s}' \mid \mathrm{s}, a)$ for the probability of moving from s via a to s′.

In our robot example, it is probably overly pessimistic to assume that the kids and neighbours dump dirt maliciously. Instead of this assumption, we now assign a probability to the event that somebody dirties one of our rooms, if it is not dirty already.

Example 1.4 As set of states, we now use $\mathrm{S}_0 = \{\mathrm{Left}, \mathrm{Right}\} \times \{\mathrm{Clean}, \mathrm{Dirty}\} \times \{\mathrm{Clean}, \mathrm{Dirty}\}$. For nondeterministic (as opposed to random) transitions, we have the same options as before: the robot can move from one room to the other, idle or clean a room if it is dirty. We assume that the left room becomes dirty with a probability of 10% if it is not dirty already, and that the right room becomes dirty with a probability of 1%.

We depict a part of this MDP in Figure 1.4. We left out the actions that move the robot to the other room, and those states in which the robot is in the

right room. When we start in the state in which both rooms are clean, then two actions are available: we can move to the other room (not depicted), or Idle, i.e., wait for something to happen. After we have chosen our action, it is the turn of the environment, which randomly decides what to do next. It is most likely that both rooms remain clean. The next likely case is that the left room becomes dirty (with probability 10%). The right room becomes dirty with a probability of 1%. The least likely case is that both rooms become dirty at the same time, with probability $1\% \cdot 10\% = 0.1\%$. In the case that both rooms are dirty, the robot has two actions available: it can either ignore the dirt, in which case the only possible outcome is that both rooms remain dirty; or it can clean the room in which it is located, in which case the only possible outcome is that the room is clean now.

This simple setup already provides interesting possible controllers for the robot. If both rooms are clean, then it probably makes more sense to move to the room that is more likely to become dirty. This is only true, though, if it is more important that that room be clean. Otherwise it might make more sense to move to the more important room. Analogously, if both rooms are dirty, does it make more sense to clean the room in which the robot currently is, or is it better to go to the other room first? After all, if we first clean the room that is probably going to become dirty quickly, then this room might become dirty again while we subsequently clean the other room. Another possibility is to have the robot patrol the rooms (i.e., switch from right to left and vice versa) constantly, and only clean the room if we encounter dirt. It is obvious that the optimality of a controller depends on how important we consider different characteristics, e.g.: amount of time a room stays dirty, movement costs of robots and difference importance of rooms. In any case, once we have selected a controller there is no decision remaining. What remains is a structure that contains only random transitions. We will consider those structures next.

Markov chain. A *Markov chain* (*MC*) is an MDP in which there is no decision to make. This may be the case because there really is no decision to make (e.g., because we have to work on our thesis during the coming weekend; no choice there), or because all decisions have been made already (e.g., because we know exactly what our plans for the weekend are, but we had a choice before).

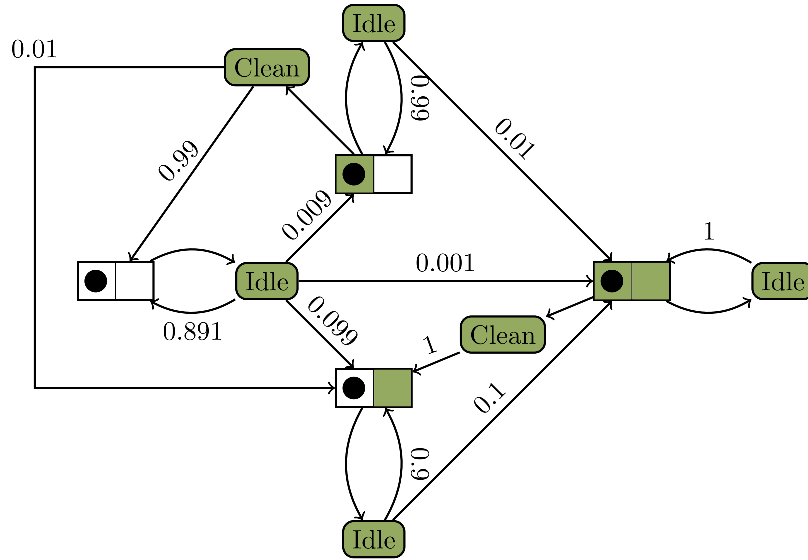


Figure 1.4: Example of a port of a robot MDP. Rectangles with rounded corners are actions, the other rectangles are state. Missing from this graph are actions moving the robot to the other room, and states with the robot on the right room.

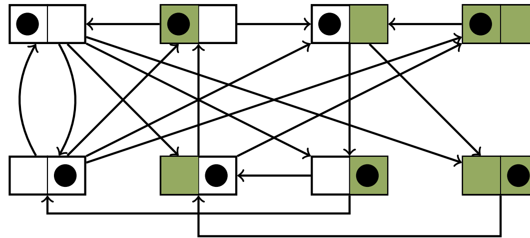


Figure 1.5: Markov chain obtained from Figure 1.4 when the robot always cleans the room it is in if the room is dirty and switches rooms otherwise; we have left out the probabilities to enhance readability.

Definition 1.5 (Markov chain) An MDP $\mathcal{M} = (\mathrm{S}_0, \mathrm{s}_0, \overline{A}, \mathrm{S}_p, p)$ is a *Markov chain* if there is no decision to make, i.e., in which $|\overline{A}(\mathrm{s})| = 1 \forall \mathrm{s} \in \mathrm{S}_0$. Instead of the above, we often write $\mathcal{M} = (\mathrm{S}_0, \mathrm{s}_0, p)$.

We are now going to pick an action for each state our little robot may be in. We note here that picking an action for each state is a very simple way of turning an MDP into a MC. We will investigate this more formally in Section 1.4

Example 1.5 Our little robot is very anxious to get to work, and even more anxious when it does not have anything to work on. It therefore cleans any dirt

in the room it currently is in, and if it finds no dirt in the common room, then it moves to the other. The resulting MC (without probabilities) is depicted in Figure 1.5.

Markov chains associate a probability with their finite runs. In the following, we will describe a probability space $\mathcal{P} := (\Omega, \mathcal{F}, \mu)$ over the set of infinite runs S_0^ω. Since infinite runs have probability zero in every non-trivial Markov chain. we cannot simply define $\mu(\mathrm{s}_0\,\mathrm{s}_1 \cdots) = \Pi_{i \in \mathbb{N}} p(\mathrm{s}_{i+1} \mid \mathrm{s}_i)$, as the following example shows.

Example 1.6 Let us continue with the MC in Figure 1.5. We will now look at the run that starts in the state in which both rooms are clean, and the robot is in the left room. The probability of going to the state in which both rooms are still clean, and the robot is in the right room is 0.891. The probability of going back to the initial state is again 0.891. So the probability of the finite loop we have just described is $0.891 \times 0.891 = 0.891^2$. The probability of taking the loop twice is 0.891^4. The probability of the infinite run consisting of only loops is therefore $\lim_{n \to \infty} 0.891^n = 0$.

Instead, we will follow common practice and use the probability of finite runs to define the probability of sets of infinite runs (so-called *cones*) with common prefix, and generate a probability space based on those.

Definition 1.6 (Probabilities of runs and cones [Put94]) Let $\rho \in \Omega_\mathrm{s}^*(\mathcal{M})$ be a finite run starting in some state $\mathrm{s} \in \mathrm{S}$. Then the probability of ρ is defined as $p(\rho) := \Pi_{0 \leq i < n} p(s_{i+1} | s_i)$. The cone $\nabla(\rho) \subseteq \mathrm{S}_0^\omega$ is defined as $\{\rho\rho' \mid \rho\rho' \in \Omega_{\rho_0}^\omega(\mathcal{M})\}$. The probability measure of a cone is the probability of the common prefix, i.e., $\mu(\{\nabla(\rho)\}) := p(\rho)$. The samples of $\mathcal{M}$ are the cones, and all sets of cones starting in a common state are measurable events, where $\mu(\{\nabla(\rho_0), \nabla(\rho_1), \ldots\}) = \sum_{i \geq 0} p(\rho_i)$ if $\rho_i \neq \rho_j \forall i \neq j \in \mathbb{N}$.

Example 1.7 Let us continue with the MC in Figure 1.5. We will start with the state in which both rooms are clean, and in which the robot is in the left room. As before, we now move to the state in which both rooms are still clean, and in which the robot is in the right room, and then back to the previous state. The probability of this finite run v is $1 \cdot 10^{-6}$. The cone $\nabla(v)$ describes the set of infinite runs that start from here.

To see why it makes sense to define the probability of a cone as the probability of its finite common prefix, consider what can happen after v, i.e., what is the possible extension of v. According to Figure 1.5 there are four possibilies: (1) v_1: both rooms stay clean (2) v_2: the left room becomes dirty (3) v_3 the right room becomes dirty (4) v_4 both rooms become dirty. Note that $\nabla(v_1), \ldots, \nabla(v_4)$ partition $\nabla(v)$, i.e., every infinite extensions of v is an extension of one of the v_i. Therefore, $p(\nabla(v)) = p(\bigcup_{i=1}^4 \nabla(v_i))$ should be $\sum_{i=1}^4 p(\nabla(v_i))$. This hold because of the way the probabilities of cones are defined. Vice versa, this is the only possible way to define the probability of cones that results in a properly defined probability space having this property.

Two player games. Sometimes, we are really not worried about how a system behaves on average, or we have no idea what the probabilities are that influence something beyond our control. Chess is one example where probabilities play no role, but where there is clearly an antagonistic opponent. Another example is when relatives come to visit and you want to know how long a room might be dirty in the worst case. To model situations like these we define two player games. In the following, when it is unambiguous, we will refer to two player games simply as games.

Definition 1.7 (Game) A game is a stochastic game $\mathcal{S} = (\mathrm{S}, \mathrm{S}_0, \mathrm{S}_1, \mathrm{S}_p, \mathrm{s}_0, \Delta, p)$ in which $\mathrm{S}_p = \emptyset$. When defining games, we often leave out S_p and p and write $\mathcal{S} = (\mathrm{S}, \mathrm{S}_0, \mathrm{S}_1, \mathrm{s}_0, \Delta)$ instead.

To illustrate these concepts, we will let an antagonist decide which room becomes dirty when.

Example 1.8 Consider Figure 1.4, but ignore the probabilities on the edges. Player 0 picks the action, while Player 1 picks the following state from the set of reachable states.

We can use this graph to ask, for example, for the maximum number of consecutive steps that a room can be dirty compared to the number of steps that it is clean, or for the ratio of cleaning steps per walking step of the robot.

Finite state machines. *Finite state machines* are systems that change their states depending on an input word that they read. They can be seen as games in which the players take turns.

Definition 1.8 (Finite state machine) Formally, a finite state machine $\mathcal{S} = (\mathrm{S}, \mathrm{S}_0, \mathrm{S}_1, \mathrm{s}_0, \Delta)$ is a game with two restrictions. These restrictions ensure that Player 1 acts as providing an input letter, while Player 0 changes the state of the machine according to the input letter. Firstly, states following a Player 0 state are Player 1 states, and vice versa. Formally $\mathrm{s} \in \mathrm{S}_0 \wedge (\mathrm{s}, \mathrm{s}') \in \Delta \implies \mathrm{s}' \in \mathrm{S}_1$ and that $\mathrm{s} \in \mathrm{S}_1 \wedge (\mathrm{s}, \mathrm{s}') \in \Delta \implies \mathrm{s}' \in \mathrm{S}_0$. Secondly, Player 1 starts the game, i.e., $\mathrm{s}_0 \in \mathrm{S}_1$.

Usually, finite state machines are defined as reading from an input alphabet Σ, and denoted by $(\mathrm{S}, \mathrm{s}_0, \Sigma, \Delta)$, where $\Delta \subseteq \mathrm{S} \times \Sigma \times \mathrm{S}$. This corresponds to the game $(\mathrm{S} \cup \mathrm{S}', \mathrm{S}', \mathrm{S}, \mathrm{s}_0, \Delta')$, in which $\mathrm{S}' = \mathrm{S} \times \Sigma$, $\Delta' = \{(\mathrm{s}, (\mathrm{s}, \sigma)) \mid \mathrm{s} \in \mathrm{S}, \sigma \in \Sigma\} \cup \{((\mathrm{s}, \sigma), \mathrm{s}') \mid (\mathrm{s}, \sigma, \mathrm{s}') \in \Delta\}$. Intuitively, Player 1 first decides what letter comes next. Then Player 0 decides what the next state is. If Player 0 has no choice in any of his states (i.e., if $|\{\mathrm{s}' \in \mathrm{S} \mid \exists \sigma : (\mathrm{s}, \sigma, \mathrm{s}') \in \Delta\}| = 1$ for all $\mathrm{s} \in \mathrm{S}_0$), then the machine is called *deterministic*, and *non-deterministic* otherwise. For deterministic machines, we define $\delta : \mathrm{S} \times \Sigma \to \mathrm{S}$ as $\delta(\mathrm{s}, \sigma) = \mathrm{s}'$ such that $(\mathrm{s}, \sigma, \mathrm{s}') \in \Delta$.

Sometimes, we want our machines to not only read and change their state, but answer us as well. Therefore we equip them with output functions. Such machines are sometimes called transducers.

Definition 1.9 (Moore/Mealy machines, transducers) Transducers are deterministic machines equipped with an output function γ over an output alphabet Ω, such that it can read and write letters. There are two possible cases for γ: (1) the output depends exclusively on the current state of the machine, i.e., $\gamma : \mathrm{S} \to \Omega$, or (2) the output depends on the current state of the machine and the letter that is read, i.e., $\gamma : \mathrm{S} \times \Sigma \to \Omega$. A machine with output function (1) is called a *Moore machine*, a machine with output function (2) is called a *Mealy machine*. For both, we often write $(\mathrm{S}, \mathrm{s}_0, \Sigma, \Omega, \delta, \gamma)$ instead of the stochastic game.

Moore and Mealy machines each define a unique output word for each input word. Let $w \in \Sigma^\omega \cup \Sigma^*$ be an input word. Then the run of a machine is the sequence $\rho = \rho_0 \rho_1 \cdots$, where $\rho_0 = \mathrm{s}_0$ and $\rho_i = \delta(\rho_{i-1}, w_{i-1})$ for $i > 0$. The *output word* on input w of a Mealy machine, denoted by $\gamma(w)$, is defined as $\gamma(\rho_0, w_0)\, \gamma(\rho_1, w_1) \cdots$, and that of a Moore machine is defined as $\gamma(\rho_0)\, \gamma(\rho_1) \cdots$.

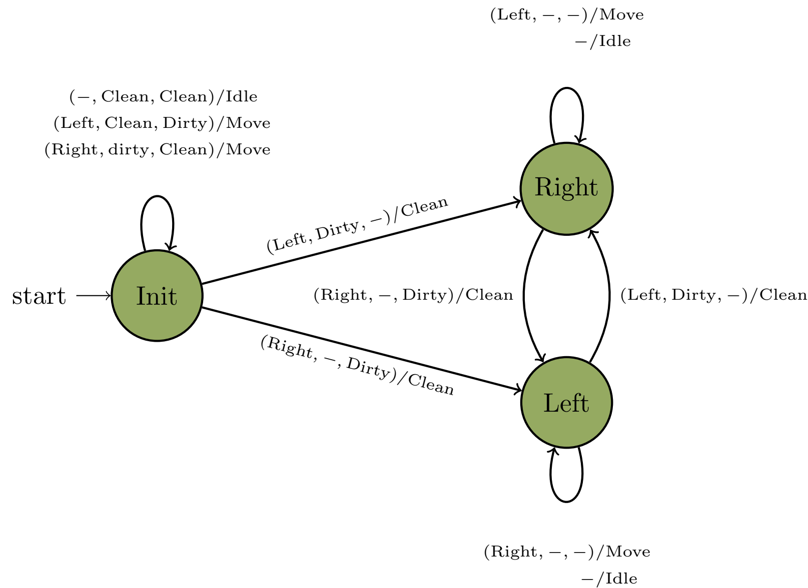


Figure 1.6: A machine for controlling the cleaning robot. The robot cleans the first room in which it discovers dirt, and then always the room it has not cleaned last. The labels on the edges of format In/Out describe what the machine reads as input and writes as output as reactions. Dashes (−) stand for the joker pattern, i.e., match anything not matched by anything else in the state.

Example 1.9 We will now build a machine to control our robot for the model in Figure 1.4. Our robot believes that it is better to clean the room that is has not cleaned last. We implement such a robot using a finite state machine with three states: (1) an initial state signifying that it has not cleaned any room yet, (2) a state Left signifying that the robot wants to clean the left room next (3) a state Right signifying that it wants to clean the right room next.

We therefore have S = {Init, Right, Left}. The robot reads from the set of states of the MDP, and it outputs actions, i.e., one of {Move, Idle, Clean}. We depict this machine in Figure 1.6. As you can see, the machine starts in state Init and stays there until the room the robot is in is dirty. The robot idles while no room is dirty. If the room the robot does not occupy becomes dirty first, then it will move to the other room and then clean the room in the next step. As soon as the left room becomes dirty and the robot is in the left room, the machine tells the robot to clean the room and moves to state

Right, signifying that it wants to clean the right room next. The analogous thing happens when the right room becomes dirty first.

Once in state Right, the robot first switches to the other room. It then idles there until the right room becomes dirty, in which case the robot cleans it and moves to state Left, and so on.

Strategies and objectives

Strategies. When two players play a game, they usually have a strategy beforehand, and sometimes make one up on the spot. A strategy can be either deterministic, i.e., in each state the player knows exactly what she wants to do, or she can leave the actual outcome to chance by, for example, flipping a coin. Further, a player's strategy can either depend on only the current state of the game, or it can depend on a finite history of the game, or it can depend on the complete history of the game.

Definition 1.10 (Strategy) Therefore, in its most general form, a *strategy* for Player $i \in \{0,1\}$ in a stochastic game $\mathcal{S} = (\mathrm{S}, \mathrm{S}_0, \mathrm{S}_1, \mathrm{S}_p, \mathrm{s}_0, \Delta, p)$ is a function $d : \mathrm{S}^* \, \mathrm{S}_i \to \mathrm{D}(\mathrm{S})$ such that $d(\rho\, \mathrm{s})(\mathrm{s}') > 0 \implies (\mathrm{s}, \mathrm{s}') \in \Delta$ for each run $\rho\, \mathrm{s}$ of $\mathcal{S}$.

A Player 0 strategy such that the co-domain of $d(\rho)$ is $\{0, 1\}$ for all runs $\rho \in \mathrm{S}^* \, \mathrm{s}$ with $\mathrm{s} \in \mathrm{S}_0$ is called *deterministic*. If a strategy can be implemented by a Moore machine, then it is called *finite memory strategy* or *finite-state strategy*. Formally, a strategy is a finite memory strategy if there is a Moore machine $\mathrm{M} = (S_{\mathcal{O}}, o_0, \Sigma, \Omega, \delta, \gamma)$ such that for all non-empty runs $\rho = \mathrm{s}_0\, \mathrm{s}_1 \cdots \mathrm{s}_n \in \Omega^*_{\mathrm{s}_0}(\mathcal{S})$ and for the output word $\omega_0\, \omega_1 \cdots \omega_n = \gamma(\rho)$ produced by M on ρ we have that $d(\rho) = \omega_n$.

We have already seen a finite, deterministic strategy in Example 1.9.

Qualitative properties. In this thesis, we call properties that a system or a run of a system does or does not have — i.e., properties that have no middle ground — *qualitative properties*. Examples of such properties include deadlock-freedom (can the system always continue, or does it get stuck) and safety (does the system always remain in a safe region of a state space). In this thesis, we will chiefly specify qualitative properties via linear temporal logic (LTL) [Pnu77]. We define qualitative properties over a finite set of *atomic propositions* $AP \neq \emptyset$.

Definition 1.11 (Linear temporal logic) *Linear temporal logic (LTL)* formulas are defined by the following grammar.

$$\varphi ::= p \in AP \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathsf{X}\,\varphi \mid \varphi\,\mathsf{U}\,\varphi$$

The semantics of this syntax is defined over infinite sequences of sets of atomic propositions, i.e., over $(2^{AP})^\omega$. A sequence $w \in (2^{AP})^\omega$ *fulfills* a property φ, written as $w \models \varphi$ if

- $\varphi = p \in AP$ and $p \in w_0$
- $\varphi = \neg\varphi'$ and not $w \models \varphi'$
- $\varphi = \varphi_1 \vee \varphi_2$ and $(w \models \varphi_1$ or $w \models \varphi_2)$
- $\varphi = \mathsf{X}\,\varphi'$ and $w_{>0} \models \varphi'$
- $\varphi = \varphi_1\,\mathsf{U}\,\varphi_2$ and there is an $i \in \mathbb{N}$ such that (1) for all $0 \leq j < i$ $w_{\geq j} \models \varphi_1$ and (2) $w_{\geq i} \models \varphi_2$

We denote by $\mathrm{L}(\varphi) = \{w \in (2^{AP})^\omega \mid w \models \varphi\}$ the *language of* φ, i.e., the set of sequences satisfying φ.

In addition to the grammar above, we define the following symbols for convenience.

- $\mathsf{true} = p \vee \neg p$ for some $p \in AP$
- $\mathsf{false} = \neg\mathsf{true}$
- $\varphi_1 \wedge \varphi_2 = \neg(\neg\varphi_1 \vee \neg\varphi_2)$
- $\mathsf{F}\,\varphi = \mathsf{true}\,\mathsf{U}\,\varphi$
- $\mathsf{G}\,\varphi = \neg\,\mathsf{F}\,\neg\varphi$

Example 1.10 Coming back to our robot example, one useful LTL property would be $\mathsf{G}(\mathrm{leftDirty} \rightarrow \mathsf{F}\,\mathrm{leftClean})$, i.e., whenever the left room is dirty, then it will be cleaned at some point. Another one we could find interesting is $\mathsf{G}(\mathrm{roboLeft} \wedge \mathsf{X}(\mathrm{roboRight}) \rightarrow \mathrm{rightDirty})$, i.e., we are only allowed to move from the left room to the right room when the right room is dirty.

For two words $w, w' \in (2^{AP})^\omega$ we define $w \cup w'$ by $(w \cup w')_i = w_i \cup w'_i$ for all $i \in \mathbb{N}$. We say that a Mealy machine $\mathrm{M} = (\mathrm{S}, \mathrm{s}_0, AP_I, AP_O, \delta, \gamma)$ fulfills a specification φ, written as $\mathrm{M} \models \varphi$ if for all words $w \in (2^{AP_I})^\omega$ we have that $w \cup \gamma(w) \in \mathrm{L}(\varphi)$. This definition works analogously for Moore machines.

The second qualitative property we will consider is the parity condition.

Definition 1.12 (Parity condition) Let $\lambda : 2^{AP} \to \mathbb{N}$ be a function with finite co-domain. This function defines a parity condition. We say that an infinite word $w \in (2^{AP})^\omega$ fulfills the parity condition if the highest number seen infinitely often in $\lambda(\rho_0) \cdot \lambda(\rho_1) \cdots$ is even.

We say that a Mealy machine fulfills a parity condition if for all words $w \in (2^{AP_I})^\omega$ we have that $w \cup \gamma(w)$ fulfills the parity condition.

Quantitative properties. By *quantitative properties* we mean properties that are not strictly true or false for a system or a word. Instead, a word or a system is associated with a number. For example, the time taken until an event happens is a quantitative property. Another is the energy used by a system.

We deal with two kinds of quantitative properties here. Firstly, we consider *probabilistic computation tree logic* (*PCTL*)[HJ94], which is a logic for probabilistic systems.

Definition 1.13 (PCTL) The grammar of PCTL is defined by the following.

$$\varphi ::= p \in AP \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathrm{P}_{\sim c}[\psi]$$
$$\psi ::= \varphi \,\mathsf{U}\, \varphi \mid \mathsf{X}\, \varphi,$$

where $c \in [0,1]$ is a probability, $\sim \in \{<, \leq, =, \geq, >\}$ is a relation and AP is a set of atomic propositions. In the above, a formula of shape φ is called a *state formula*, while a formula of shape ψ is called a *path formula*.

We define the semantics to the above syntax over a Markov chain $\mathcal{M} = (\mathrm{S}_0, \mathrm{s}_0, p)$ and a labelling $\lambda : \mathrm{S} \to 2^{AP}$. We say that a run ρ of $\mathcal{M}$ fulfills a path formula ψ, written as $\rho \models \psi$ if

- $\psi = \mathsf{X}\, \varphi$ and $\rho_1 \models \varphi$
- $\psi = \varphi_1 \,\mathsf{U}\, \varphi_2$ and there is an $i \in \mathbb{N}$ such that (1) for all $0 \leq j < i$ $\rho_{\geq j} \models \varphi_1$ and (2) $\rho_{\geq i} \models \varphi_2$,

where $\mathrm{s} \models \varphi$ means that state $\mathrm{s} \in \mathrm{S}$ fulfills state formula φ, which is the case if

- $\varphi = p \in AP$ and $p \in \lambda(\mathrm{s})$
- $\varphi = \neg\varphi'$ and not $s \models \varphi'$
- $\varphi = \varphi_1 \vee \varphi_2$ and ($\mathrm{s} \models \varphi_1$ or $\mathrm{s} \models \varphi_2$)
- $\varphi = \mathrm{P}_{\sim c}[\psi]$ if $\mathrm{P}(\{\rho \models \psi \mid \rho \in \Omega^\omega_{\mathrm{s}}(\mathcal{M})\}) \sim c$,

where $\mathrm{P}(\{\rho \models \psi \mid \rho \in \Omega^\omega_{\mathrm{s}}(\mathcal{M})\})$ is the probability measure of all runs starting in s that fulfill ψ, which is a measurable function according to [Var85].

Example 1.11 Recall the Markov chain in Figure 1.5, we could ask for the probability that both rooms will be dirty at some point, i.e., $\mathrm{P}[\mathsf{true}\,\mathsf{U}\,\mathrm{bothDirty}]$, which is equal to 1 in our case. Or we can ask for the probability that the left room will stay dirty forever, i.e., $\mathrm{P}[\mathsf{true}\,\mathsf{U}\,\mathrm{P}_{=0}[\mathsf{true}\,\mathsf{U}\,\mathrm{leftClean}]]$ The probability that both room become dirty are dirty in the next step is $\mathrm{P}[\mathsf{X}\,\mathrm{bothDirty}] = 0.001$ in any state in which both rooms are clean, and zero in any state in which both rooms are dirty already.

In addition to PCTL formulas, we also have reward-based properties.

Definition 1.14 (Reward function, accumulation function) A *reward function* $r : \mathrm{S} \to \mathbb{R}$ for a transition system $\mathrm{L} = (\mathrm{S}, \Delta, \mathrm{s}_0)$, is a function which assigns a reward to each state. In addition, we define *accumulation function*s $\alpha : \mathbb{R}^\omega \to \mathbb{R} \cup \{+\infty, -\infty\}$ or $\alpha : \mathbb{R}^\omega \times \mathbb{R}^\omega \to \mathbb{R} \cup \{+\infty, -\infty\}$ that accumulate rewards over paths. In this thesis, we will encounter the following reward functions.

- *Mean payoff* $\alpha(w) = \liminf_{n\to\infty} 1/n \sum_{i=0}^{n} w_i$, which yields the average reward seen along a path.
- *Minimum sum* $\alpha(w) = \min_{n\to\infty} \sum_{i=0}^{n} w_i$ which yields the minimal or maximal sum seen along a path. The maximum sum is defined analogously.
- *Discounted payoff* $\alpha(w) = \sum_{i\in\mathbb{N}} \lambda^i\, w_i$ for a $\lambda \in [0, 1)$, which considers rewards near the beginning of a path more important than those at the end of a path.
- *Ratio payoff* $\alpha(w^1, w^2) = \lim_{l\to\infty} \liminf_{u\to\infty} \frac{\sum_{i=l}^{u} w_i^1}{1+\sum_{i=l}^{u} w_i^2}$, which yields the ratio of two rewards.

Depending on what property we want to specify, we use different accumulation functions. We can use the mean payoff to calculate the average speed of a vehicle. The minimal or maximal sum of prefixes is useful when modeling a finite resources, such as battery charge in a robot. The discounted payoff is useful if we are uncertain of the future development of a model, but certain regarding the near future, for example for financial modelling. Lastly, the ratio of two rewards can be used for efficiency as we explore in Chapter 2, or to calculate the expected outcome of a repeated experiment.

The ratio uses two limits, one of them a limit inferior because the first limit allows us to ignore a finite prefix of the run, which ensures that we only consider the long-run behavior. We need the limit inferior here because the sequence of the limit might not converge. Consider a combination with states q and r, and the run $\rho = q^1 r^2 q^4 r^8 q^{16} \dots$, where q^k means that State q is visited k-times. Assume State q and State r have the following costs: $c(q) = 0$, $r(q) = 1$, $c(r) = 1$ and $r(r) = 1$. Then, the efficiency of $\rho_0 \dots \rho_i$ will alternate between $1/6$ and $1/3$ with increasing i and hence the sequence for $i \to \infty$ will not converge. The limit inferior of this sequence is $1/6$. The 1 in the denominator avoids division by 0 if the accumulated costs are 0 and has no effect if the accumulated costs are infinite. For similar reasons we use the maximum in the maximum sum (instead of just the infinite sum) and the limit inferior for the mean payoff.

Example 1.12 In the robot example we use several reward functions. For example, we can ask for the mean number of rooms cleaned per step, and compare with the mean number of dirty rooms. We could also add a battery to the robot and use the maximum sum reward to determine if it will always stay at an energy level greater than zero. To determine how efficient the robot is, we can ask how many rooms it cleans per movement step.

We have now defined rewards first for states or transitions and then for runs. We are now going to lift rewards from runs to systems.

Definition 1.15 (Rewards for systems) In the following we will abuse notation and lift r from states to sequences of states, i.e., we will write $r(w)$ to mean $r(w_0)r(w_1)\cdots \in \mathbb{R}^\omega$ for a sequence of rewards $w \in \mathbb{R}^\omega$. Let $\alpha : \mathbb{R}^\omega \to \mathbb{R} \cup \{+\infty, -\infty\}$ is an accumulation function. Then the reward of a run $\rho \in \Omega_\mathrm{s}^\omega(\mathrm{L})$ is defined by $(\alpha \circ r)(w)$.

We have different approaches for different systems to lift a *run reward function* $f : \alpha \circ r$ from single runs to sets of runs Ω^ω. They fall in one of two categories.

- Worst/Best case: we use the maximum or minimum over a set of infinite runs Ω^ω e.g., $\max_{\rho\in\Omega^\omega_{\mathrm{s}}(\mathrm{L})}(f(\rho))$ or $\min_{\rho\in\Omega^\omega_{\mathrm{s}}(\mathrm{L})}(f(\rho))$.
- Average case: we use the expected value $\mathbb{E}_{\mathcal{P}}[f]$, given by a probability space $\mathcal{P} := (\Omega^\omega_{\mathrm{s}}(\mathrm{L}), \mathcal{F}, \mu)$.

In general, when using structures entailing probabilities, then we use the average case, otherwise we will consider the worst/best-case.

Optimal strategies. Strategies are optimal for two player games and MDPs if a quantitative objective is optimized or if a qualitative objective is fulfilled. Formally, we start out with a stochastic game $\mathcal{S} = (\mathrm{S}, \mathrm{S}_0, \mathrm{S}_1, \mathrm{S}_p, \mathrm{s}_0, \Delta, p)$, and have to find a Player 0 strategy d_0 such that no matter what strategy d_1 Player 1 uses, Player 0 is playing optimal or according to a specification.

Definition 1.16 (Runs and probability space of strategies) Given $\mathcal{S}$ and d_0, d_1 as defined above, we define the set of runs $\mathrm{L}(\mathcal{S}, d_0, d_1) = \{\rho \in \Omega^\omega_{\mathrm{s}_0}(\mathcal{S}) \mid \forall i \in \{1, 2, \ldots\} \forall j \in \{0, 1\} : \rho_i \in \mathrm{S}_j \Rightarrow d_j(\rho_{\leq i})(\rho_{i+1}) > 0\}$. We define a probability space $\mathcal{P}(\mathcal{S}, d_0, d_1) = (\Omega^\omega_{\mathrm{s}_0}(\mathcal{S}), \mathcal{F}, \mu)$ over the cones of $\mathrm{L}(\mathcal{S}, d_0, d_1)$ as for Markov chains.

Depending on the structure and the objective (qualitative or quantitative), we then look at the worst case over all runs, the expected value of a reward function over $\mathcal{P}$ or the probability that a PCTL formula is fulfilled.

Definition 1.17 (Optimal strategy) In the following, we will use $d_\epsilon : \emptyset \to \mathcal{D}(\mathrm{S})$ to denote the strategy over the empty set, i.e., the strategy that does not need to decide anything. This strategy is used in cases where there is no player for whom to find a strategy, such as Player 1 in Markov decision processes.

- Given a game and an LTL formula φ, a Player 0 strategy d_0 is optimal if for all Player 1 strategies d_1 we have that $\mathrm{L}(\mathcal{S}, d_0, d_1) \subseteq \mathrm{L}(\varphi)$.
- Given a game and a parity condition $\lambda : \mathrm{S} \to \mathbb{N}$, a Player 0 strategy d_0 is optimal for all Player 1 strategies d_1 we have that all words in $\mathrm{L}(\mathcal{S}, d_0, d_1)$ fulfill the parity condition.

- Given a game and a function $f : \Omega^{\omega}_{s_0}(\mathcal{M}) \to \mathbb{R}$, a strategy d is optimal if d has the optimal worst case over all strategies d_1 of Player 1, i.e., a strategy d such that

$$\min_{d_1} \min_{\rho \in \mathrm{L}(\mathcal{S},d,d_1)} f(\rho) = \max_{d_0} \min_{d_1} \min_{\rho \in \mathrm{L}(\mathcal{S},d_0,d_1)} f(\rho)$$

.

- Given a Markov decision process $\mathcal{M} = (\mathrm{S}_0, \mathrm{s}_0, \overline{A}, \mathrm{S}_p, p)$ and a parity condition $\lambda : \mathrm{S}_0 \to \mathbb{N}$, we say that a strategy d_0 is optimal if the probability that a run fulfills the parity condition is maximal.

- Given a Markov decision process $\mathcal{M} = (\mathrm{S}_0, \mathrm{s}_0, \overline{A}, \mathrm{S}_p, p)$ and a measurable function $f : \Omega^{\omega}_{s_0}(\mathcal{M}) \to \mathbb{R}$, a strategy d_0 is optimal if $\mathbb{E}_{\mathcal{P}(\mathcal{S},d_0,d_\epsilon)}[f]$ is minimal or maximal over all strategies.

- Given a Markov decision process $\mathcal{M} = (\mathrm{S}_0, \mathrm{s}_0, \overline{A}, \mathrm{S}_p, p)$ and a PCTL path formula ψ, a strategy is optimal if $\mathrm{P}(\{\rho \models \psi \mid \rho \in \Omega^{\omega}_{s_0}(\mathcal{M})\})$ is minimal or maximal as measured in $\mathcal{P}(\mathcal{S}, d_0, d_\epsilon)$.

Verification and Synthesis. Verification means proving that a system fulfills a given property, no matter what an adversary or the environment does. For MDPs it means showing that the "optimal" strategy is above or below a certain bound. For example, we might want to prove that, no matter what the environment decides to do, the expected time a room is dirty is lower than 10 seconds. Synthesis, on the other hand, means finding a controller that would pass verification, i.e., a controller that is correct by construction. Coming back to the robot, we could ask for a controller such that, no matter what the environment does, the expected maximal time a room is dirty is lower than 10 seconds.

In the cases that we treat in this thesis, verification and synthesis both mean finding an optimal strategy[4], and can therefore be treated equally. The only difference is in what we do with the result. In verification we ask only for the existence of such a strategy, i.e., we ask if a specification is *realizable*. In synthesis, for a strategy that realizes an objective. We can then use this strategy to build a controller.

[4]That is, realizability and synthesis are the same problem from an algorithmic standpoint.

Combined objectives

In addition to regarding quantitative objectives and qualitative objectives in isolation, we can also study combinations of these two. In this thesis, we consider two kinds of combinations. Firstly, we combine different reward functions and use the same accumulation function. For example, we can consider the mean fuel usage and mean velocity, both of which we want to optimize. There is an obvious trade-off between these two. There are two ways of approaching the search for strategies for this kind of combination. We can either ask for a strategy such that all accumulated rewards are above or below a certain threshold, or we can ask for an approximation of all possible trade-offs, a so called Pareto curve. Secondly, we can consider combining quantitative and qualitative objectives.

Combined quantitative objectives. In the rest of this paragraph, we will use a vector of reward and accumulation functions $r = (\alpha \circ r_0, \alpha \circ r_1, \ldots, \alpha \circ r_n)$ for a stochastic game $\mathcal{S} = (\mathrm{S}, \mathrm{S}_0, \mathrm{S}_1, \mathrm{S}_p, \mathrm{s}_0, \Delta, p)$. For an infinite run $\rho \in \Omega^{\omega}_{\mathrm{s}_0}(\mathcal{S})$, we will denote by $r(\rho) = ((\alpha \circ r_0)(\rho), (\alpha \circ r_1)(\rho), \ldots, (\alpha \circ r_n)(\rho))$. We denote by $r^{\uparrow}(\mathcal{S}, d_0, d_1) \in \mathbb{R}^{n+1}$ the lifted reward function that defines the reward aggregated over all runs of the stochastic game when strategies d_0 and d_1 are applied. For example, in the case that the stochastic game is an MDP and α is the Mean function, $r^{\uparrow}(\mathcal{S}, d_0, d_1)$ denotes the vector of expected mean payoffs.

We have two ways of defining optimal strategies.

Definition 1.18 (Threshold optimal strategy) Given a vector $t \in R^{n+1}$, a *threshold optimal strategy* is a Player 0 strategy d such that for all Player 1 strategies d_1 we have $r^{\uparrow}(\mathcal{S}, d_0, d_1) \geq t$.

Definition 1.19 (Pareto optimal strategy) A strategy d is called *Pareto optimal* if it cannot be improved in any component of its reward without sacrificing another component. Formally, we call it this if there is no strategy d' such that $\max_{d_1} r^{\uparrow}(\mathcal{S}, d, d_1) < \max_{d_1} r^{\uparrow}(\mathcal{S}, d', d_1)$.

Combining quantitative and qualitative objectives. Now that we have defined quantitative and qualitative objectives, it is natural to desire to combine these two. We might want to ask for a strategy such that a qualitative objective is fulfilled and such that one or more quantitative objectives are optimized.

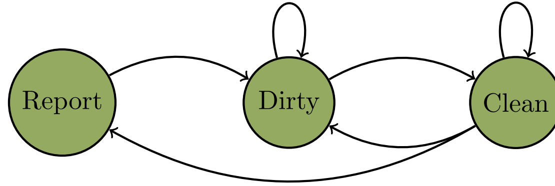


Figure 1.7: A two player game with only one player. The controller is supposed to fulfill $\mathsf{G}\,\mathsf{F}\,\text{Report}$. There further is a mean payoff function that gives a reward of 1 for every time that clean is visited, and 0 for the other states. The controller is supposed to achieve a mean payoff of one.

Example 1.13 We can ask for a strategy that, at the same time, fulfills the formula $\mathsf{G}(\text{LeftDirty} \to \mathsf{F}\,\text{LeftClean})$ and the analogous formula for the right room, and that minimizes the expected number of moves the robot has to take.

In fact, in literature we often consider something akin to the threshold objective: the task is to find a strategy such that a qualitative specification is fulfilled and such that one or more values are below or above a certain threshold.

Example 1.14 This example is adapted from [CHJ05]. Our robot now has only one room to take care of, but it also has to report its status from time to time. In Figure 1.7 we model this as a two player game. The three states model that the room is clean with state Clean, that it is dirty with state Dirty and that the robot reports its status with state Report. We then might want to fulfill, at the same time, property $\varphi = \mathsf{G}\,\mathsf{F}\,\text{Report}$, i.e., the robot will always report at some point in time, and we want to minimize the time the room is dirty. To further simplify the example, we assume that Player 1 leaves dirt in the room whenever it is clean. As payoff function we choose $r(\text{Clean}) = 1$ and $r(\text{Dirty}) = r(\text{Report}) = 0$ to reward the robot for clean rooms.

We are now looking for a strategy such that φ is fulfilled and such that the mean payoff of r is 1 (it does not matter if we pick the expected value or the minimum: they are the same). To that end, if we pick any finite memory strategy, then we necessarily will not reach mean payoff 1 if we visit Report from time to time. Any finite memory strategy has to have a postfix $v \in \mathrm{S}^\omega$ that repeats itself contiguously (recall that Player 1 has no choice here — the game is fully determined by Player 0). The mean payoff of this function equals the number of times Clean is visited divided by the length of v.

The only way to get payoff 1 is to visit state Clean longer and longer between visits to report. For example, we can play a strategy that visits Clean once, then goes to Report, then to Dirty. It then visits Clean twice, goes to Report and Dirty, and then visits Clean 3 times, and so on. This strategy provides a mean payoff of 0.

In the last example, we use a strategy with infinite memory (we need to count how often we visited Report and need to count down to know how long we have to stay in Clean. With the example before, we have just showed that when combining mean-payoff with parity objectives, the generated strategies might need infinite memory.

In Section 1.5 we will show what memory is required for what objective. We there distinguish along two axes. On the one axis, we distinguish between objectives that require randomized strategies for some structures and objectives for which deterministic strategies are enough. On the other axis, we examine the memory requirement. We distinguish between memoryless objectives, i.e., objectives where memoryless strategies are always sufficient; finite memory strategies, i.e., where strategies implementable by transducers are sufficient; and infinite memory objectives in which structures might require an infinite-memory strategy (like in the example above).

1.5 State of the art

In this section, we will investigate the known results and complexities for different quantitative and a few qualitative objectives for MDPs and two player games. This serves two purposes. On the one hand, we want to provide a context for the contribution of Chapter 2. On the other hand, we want to give an overview of what properties can be used for quantitative verification and synthesis, and how scalable these properties are. We will first look at purely qualitative objectives, and then at purely quantitative objectives. Then we will look at threshold optimal objectives, Pareto optimal objectives and finally at a mixture of quantitative and qualitative objectives.

This section glosses over huge fields of research that are related but not material to this thesis. On the one hand, the field of qualitative verification and synthesis has many more results. Refer to [BCJ14] for an overview. On the other hand, we do not consider the known results for stochastic and timed

games, because we use them as a unifying framework only, but do not depend on algorithms or results.

Qualitative objectives

LTL synthesis. LTL synthesis for games in general is two times exponentially hard in the size of the formula, i.e. $O(2^{2^{|\varphi|}})$. Research into efficiently synthesizable subsets of LTL have resulted in, for example, GR(1) specifications [BJP+12], we can synthesized in time polynomial in the size of the synthesized system. LTL synthesis for MDPs is solvable in $O(2^{2^{|\varphi|}})$ [dA97], and might require exponential memory.

Parity objective. The parity objective for games is an intriguing problem. Akin to the graph isomorphism problem it is one of the few problems which belong to NP (and co-NP), but for which it is not known whether it is in P or whether it is NP-complete[EJ88]. A pseudo-polynomial (polynomial in the size of the state space, but only pseudo-polynomial in the maximum weight) algorithm is known [McN93]. For parity games, pure strategies are sufficient[EJ88].

For MDPs, memoryless strategies are sufficient [dA97], and can be calculated in polynomial time.

Quantitative objectives

In the following we will summarize known complexity results for quantitative objectives. For each, we will indicate the best known strategy memory requirement for each objective, as well as the runtime complexity of finding an optimal strategy. Three questions marks (???) indicate open questions.

Single objective.

	Two player games		MDPs	
	Memory	Runtime	Memory	Runtime
Max	Pure [BFL+08]	P-Poly. [BFL+08]	Pure [CD11]	P-Poly. [CD11]
Discounted	Pure [ZP96]	P-Poly. [ZP96]	Pure [Put94]	Poly [Put94]
Mean	Pure [EM79]	P-Poly. [ZP96]	Pure [Put94]	Poly [Put94]
Ratio	Pure [BGHJ09]	P-Poly. [BGHJ09]	Pure [vEJ12]	Poly [vEJ12]

Table 1.1: Known complexity results for single objectives.

Mean-payoff games have been extensively studied starting with the works of Ehrenfeucht and Mycielski in [EM79] where they prove that memoryless optimal strategies exist if the other player is only allowed to use memoryless strategies as well.

No polynomial time algorithm is known for that problem. A pseudo polynomial time algorithm has been proposed by Zwick and Paterson in [ZP96], and [BCD+11] provided an improved algorithm. They also show a reduction to discounted games, and the bounds shown in the table. Max games and Mean games are equivalent according to [BFL+08]. Max MDPs can be reduced to Max games [CD11]. The ratio objective for games has been considered in [BGHJ09] to find robust strategies for games. In [vEJ12], on which Section 2 is based, we defined and analyzed the ratio objective for MDPs. For all other quantitative objectives in combination with MDPs, see [Put94].

Threshold.

	Two player games		MDPs	
	Memory	Runtime	Memory	Runtime
Max	Inf [VCD+12]	???	Random	Poly [FKN+11]
Discounted	???	???	Rand+Mem [CMH06]	Poly [CMH06]
Mean	Inf [VCD+12]	co-NP [VCD+12]	Rand+Mem [BBC+11]	Poly [BBC+11]
Ratio	???	???	Rand+Mem	Poly

Table 1.2: Known complexity results for threshold objectives.

In [VCD+12], the authors show that infinite memory optimal strategies exist for both max- and mean-payoff games, but that they might both require infinite memory. They show that finding an optimal strategy for mean-payoff games is co-NP complete. They further show that a player that if both players are restricted to finite memory strategies, then the problem is co-NP complete, and NP-complete for memoryless strategies.

For MDPs, [CMH06] first addressed finding randomized strategies for multiple discounted objectives. Later [FKN+11] extended these results to the max objective, while [BBC+11] contributed the same result for mean-payoff objectives. In all cases we have polynomial algorithms, and in all cases randomized finite memory strategies are sufficient.

Note that the results for MDPs with the discounted accumulation function require randomized strategies. Deciding if there is a deterministic strategy is NP-complete. The proof can be adapted to show that deciding if there is a deterministic strategy for the Total Sum accumulation function is NP-complete [CMH06].

In [FKN+11], the authors show that randomization and memory are needed for a variant of the discounted problem in which each reward components gets its own discount factor.

Pareto. The following table describes the time required to calculate an ϵ-approximation of the Pareto curve of an objective. For more information about Pareto curves, see Chapter 4. As for the threshold case, the ratio objective results are new and due to this thesis. That memory and randomization are required follows from the same result for the mean-payoff objective.

	MDPs	
	Memory	Runtime
Max	Random [FKN+11]	Poly [FKN+11]
Discounted	Random [CMH06]	Poly [CMH06]
Mean	Random+Mem [BBC+11]	Poly [BBC+11]
Ratio	Random+Mem	Poly

Table 1.3: Known complexity results for approximating the Pareto curve.

Typically, these approximations are achieved by reducing the problem to an equivalent linear program with multiple objectives. In Chapter 4 we show that the Pareto curve of all these objectives can be approximated via multiple optimizations of the single-payoff case.

Probabilistic qualtitative synthesis. Several authors have approached synthesizing controllers for MDPs from logic specifications. For example, [dA97] shows how to approach LTL synthesis in probabilistic environments by first transforming the LTL formula in a Rabin-automaton and then finding a controller in the product of the automaton with the MDP under reachability (equivalent to total sum objective).

[LAB11] shows how to find a controller satisfying a PCTL formula. Their solution does not always provide an optimal controller, nor does it always find a controller if one exists (i.e., it is incomplete). The runtime of their approach

is polynomial in the size of the MDP and linear in the formula.

[KP13] gives a good overview of the current state of the art and describes the approach of [dA97] in more detail.

Combining qualitative and quantitative objectives

When combining qualitative and quantitative objectives, strategies often require infinite memory where finite memory was sufficient for the single objectives (see Example 1.14).

For a combination of total sum rewards and reachability objectives (i.e., LTL formulas of the kind $\varphi = \mathsf{F}\, p$), the authors of [FKP12] show that combination of these is seamlessly possible, and that randomized strategies suffice. They also provide a polynomial runtime algorithm to solve the threshold and Pareto curve approximation problem.

For combining energy and parity or mean-payoff and parity objectives in games, the authors of [CHJ05, CRR12] show a single exponential lower and upper bound on memory for energy-parity games and mean-payoff-parity games is sufficient, if any such strategy exists. They also present an algorithm with single-exponential run-time.

One practical approach has been taken by [BBFR13], who combine LTL specifications with the mean payoff objective. They also show that the complexity of their algorithm is no worse than pure LTL synthesis, thereby providing both a lower and an upper bound.

The authors of [BFRR13] show how to combine game and MDP semantics by synthesizing strategies that fulfill a certain worst-case threshold in game-semantics and have optimal expected mean-payoff or total sum, among those strategies that fulfill the worst-case threshold. They show that finite strategies are not always sufficient and how to find a finite memory strategy if one exists.

For combining parity and max payoff objectives in MDPs, the authors of [CD11] show that finding an optimal strategy is in NP $\cap$ co-NP, and that exponential memory is required in the worst-case. For combining parity and mean-payoff objectives in MDPs, the authors of the same paper show that infinite memory is required in the worst-case, and that an optimal strategy can be found in polynomial run-time.

1.6 Tools

On the practical side of things, the best-known tools for quantitative verification are PRISM [KNP11] and MRMC [KZH+11]. PRISM (Probabilistic Symbolic Model checker) started out as a model checker for probabilistic logics for Markov decision processes and Markov chains, and has grown over the years to encompass reward-based properties and stochastic games. In contrast to other tools, PRISM concentrates on symbolic encoding via binary decision diagrams, which sometimes allow highly compressed storage and thereby faster algorithms. PRISM reads models from a custom format, which allows easy creation of new models. MRMC (Markov reward model checker) is a tool based on explicit storage and somewhat orthogonal in features to PRISM. Uppaal[BDL+06] is a tool for model checking of timed systems, i.e., systems consisting of a mix of continuous and discrete state space. Quasy [Cha11] supports finding strategies for games for a single mean-payoff objective or lexicographically ordered mean-payoff objectives. In addition, it finds strategies for mean-payoff objectives on MDPs. It also supports finding strategies for mean-payoff parity objectives of unichain MDPs (see Chapter 2 for a definition of unichain MDPs). As input it accepts games in a graph form.

On the quantitative synthesis side, Acacia+ [BBFR13] recently gained the ability to combine qualitative and quantitative specifications in a way that allows the synthesis of controllers fulfilling an LTL formula and optimizing a mean payoff.

2 Efficient Systems in Probabilistic Environments

In which we study the true meaning of efficiency and add to the big body of work around Markov Decision processes.

Résumé

Ce chapitre met au point la méthode de vérification et de synthèse utilisée pour les systèmes efficaces dans l'environment probabilistique. Nous commencerons par définition de l'efficacité, autrement dire, comment pouvons-nous obtenir la productivité optimale avec les éfforts minimisés. Pour cela nous déterminons le système comme éfficace, si il optimise le rapport entre le coüt et les efforts appliqués.

A la base de cet effet nous étudions les processus de décision markovien aves le ratio comme une foncionne objective. Ensuite nous prouvons, que ces stratégies déterministes sans mémoire sont suffisantes, et présentons trois algorithmes pour rechercher les stratégies optimales. Une de cette stratégie étant basée sur optimisation linéaire et l'autre sur optimisation linéaire fractionnaire. Ces trois algorithmes sont ensuite évaluiés sur une série des exemples. Finallement nous choisissons le plus efficace parmi ces algorithmes et le referons à la base de la diagramme de décision binaire de telle manière, qu'il se retrouve à la même echelle, comme un système de millions états.

2.1 Introduction

In this chapter we show how to automatically synthesize a system that has an "efficient" average-case behavior in a given environment. The efficiency of a system is a natural question to ask; it has also been observed by others,

e.g, Yue et al. [YBK10] used simulation to analyze energy-efficiency in a MAC (Media Access Control) Protocol. The oxford dictionary defines the adjective efficient as follows.

Definition 2.1 (Efficient (Oxford Dictionary)) Efficient (adjective): (of a system or machine) achieving maximum productivity with minimum wasted effort.

We analogously define efficiency as the ratio between a given *cost* model and a given *reward* model. To further motivate this choice, consider the following example: assume we want to implement an automatic gear-shifting unit (ACTS) that optimizes its behavior for a given driver profile. The goal of our implementation is to optimize the fuel consumption per kilometer (l/km), a commonly used unit to advertise efficiency. In order to be most efficient, our system has to maximize the speed (given in km/h) while minimizing the fuel consumption (measured in liters per hour, i.e., l/h) for the given driver profile. If we take the ratio between the fuel consumption (the "cost") and the speed (the "reward"), we obtain l/km, the desired measure.

Given an efficiency measure, we ask for a system with an optimal average-case behavior. The average-case behavior with respect to a quantitative specification is the expected value of the specification over all possible behaviors of the systems in a given probabilistic environment [CHJS10]. We describe the probabilistic environment using Markov Decision Processes (MDPs), which is a more general model than the one considered in [CHJS10]. It allows us to describe environments that react to the behavior of the system (like the driver profile).

Related Work

Related work can be divided into two categories: (1) work using MDPs for quantitative synthesis and (2) work on MDP reward structures.

From the first category we first consider [CHJS10]. We generalize this work in two directions: (i) we consider ratio objectives, a generalization of average-reward objectives and (ii) we introduce a more general environment model based on MDPs that allows the environment to change its behavior based on actions the system has taken. In the same category there is the work of Parr and Russell [PR97], who use MDPs with weights to present partially specified

machines in Reinforcement Learning. Our approach differs from this approach, as we allow the user to provide the environment, the specification, and the objective function separately and consider the expected ratio reward, instead of the expected discounted total reward, which allows us to ask for efficient systems. Finally, in [WBB+10], Wimmer et.al. introduce a semi-symbolic policy algorithm for MDPs with the average objective, while we present a semi-symbolic policy algorithm for MDPs with the ratio objective, subsuming the former.

Semi-MDPs [Put94] fall into the second category. Unlike work based on Semi-MDPs, we allow a reward of value 0. Furthermore, we provide an efficient policy iteration algorithm that works on our Ratio-MDPs as well as on Semi-MDPs. Approaches using the discounted reward payoff (cf. [Put94]) are also related but focus on immediate rewards instead of long-run rewards. Similarly related is the work of Cyrus Derman [Der62], who considered the payoff function obtained by dividing the expected costs by expected rewards. As shown later, we believe that our payoff function is more natural. Note that these two objective functions are in general not the same. Closest to our work is the work of de Alfaro [dA97]. In this work the author also allows rewards with value 0, and he defines the expected payoff over all runs that visit a reward with value greater than zero infinitely often. In our framework the payoff is defined for all runs. De Alfaro also provides a linear programming solution, which can be used to find the ratio value in an End-Component (see Section 6). We provide two alternative solutions for End-Components including an efficient policy iteration algorithm. Finally, we are the first to implement and compare these algorithms and use them to synthesize efficient controllers.

2.2 The system and its environment

In this section we will introduce the system, its environment and the quantitative monitor. We will show how they operate in lockstep and how their combination leads to a system with measurable performance. While doing so, we will introduce the necessary notation and definitions.

The system

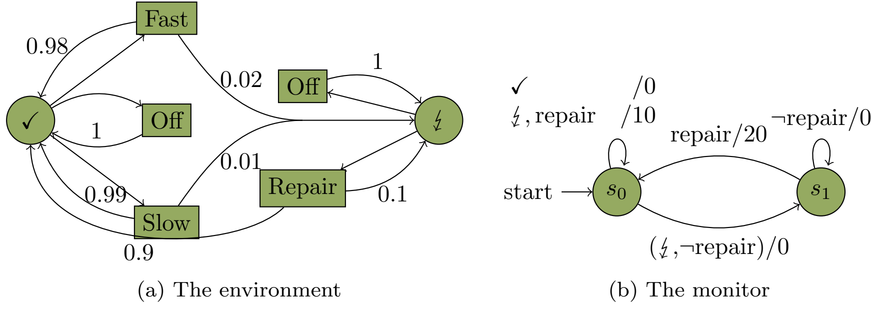
The systems we aim to synthesize are *reactive systems*. That is, systems that react infinitely to events from their environment. As usual, we model a reactive system as a *Moore Machine*, i.e., a machine that reads *letters* from an *alphabet* as *input* and writes letters in turn as *output* (see Definition 1.9 in Section 1.4).

Recall that a transducer is a tuple $T = (S, s_0, \Sigma, \Omega, \delta, \gamma)$, where S denotes the finite set of states of T, s_0 its initial state, Σ its finite input alphabet, Ω its finite output alphabet. Function $\delta : S \times \Sigma \to S$ is the transition function of T, defining how it moves from state to state in the course of reading its input. Finally, function $\gamma : S \times \Sigma \to \Omega$ is the output function of T, defining what output it writes, given the current state and the current input letter. If γ is constant in its second parameter (i.e., if $\forall s \in S\ \forall w_0, w_1 \in \Sigma : \gamma(s, w_0) = \gamma(s, w_1)$), then we call T a *Moore machine*, otherwise a *Mealy machine*. For Moore machines, we sometimes use $\gamma : S \to \Omega$ and $\gamma : S \times \Sigma \to \Omega$ equivalently.

As a running example we will synthesize a controller for a production plant. The plan consists of several production lines and we have several conflicting objectives. On the one hand, we want to maximize the number of units produced. On the other hand, driving the plant at full speed increases maintenance costs due to failing production lines. This is clearly a question of efficiency. A system (i.e., the plant controller) in this setting reads the state of the production lines, e.g., production line is broken or working. It then decides to turn specific lines on or off based on this state information.

We model the stream of events from the environment as an infinite stream of input letters, and the reactions of the system as an infinite stream of output letters. It is our goal to enable the probabilistic environment to react to the reactions of the system. To that end, we make the output of the system the input of the environment, thus forming a feedback loop. The system and the environment are stateful. Depending on the reactions of the system, the environment changes its state. We model such an environment as a MDP (Definition 1.4).

Recall that an MDP is defined by a tuple $\mathcal{M} = (M, m_0, A, \overline{A}, p)$, where M is the finite set of states of $\mathcal{M}$, $m_0 \in M$ is the initial state of the $\mathcal{M}$, A is its set of *actions*, $\overline{A} \subseteq M \times 2^A$ is the action activation relation and $p : M \times A \times M \to [0, 1]$ is the probability transition function, i.e., we demand that $\sum_{s' \in M} p(s, i, s') = 1$ for all states $s \in M$ and actions $i \in 2^A$. We demand that each state has at



(a) The environment

(b) The monitor

Figure 2.1: Environment model and quantitative specification of the production line example

least one activated action, i.e., that $\forall m \in M\ \exists a \in A :\ (m, a) \in \overline{A}$. When using an MDP to model the environment, then we assume without loss of generality that all actions are always activated, i.e., $\overline{A} = M \times 2^A$.

Recall further that a Markov chain (MC) (Definition 1.5) is a Markov decision process for which there exists exactly one action for each state, i.e., for which the cardinality of the set $\{a \in A \mid (m, a) \in \overline{A}\}$ is one for all states $m \in M$. For a Markov chain $\mathcal{M} = (M, m_0, A, \overline{A}, p)$ we sometimes write $\mathcal{M} = (M, m_0, p)$, and then we also write $p : M \times M \to [0, 1]$.

Instead of feeding the states of the MDP directly to the system, we use a labeling in between. The labeling intuitively allows us to decouple model states from inputs the model feeds the system. We could do without it, but it occasionally makes describing a model more pleasant.

Definition 2.2 (Labeling) Let Λ be a finite set. A *labeling* for $\mathcal{M}$ is a function $\lambda : M \to 2^\Lambda$ that is deterministic with respect to the transition function of $\mathcal{M}$, i.e., for all states $m, m', m'' \in M$ and every action $a \in A$ such that $p(m, a, m') > 0$ and $p(m, a, m'') > 0$ and $m' \neq m''$ we have $\lambda(m') \neq \lambda(m'')$.

The environment

Example 2.1 (Modelling a single production line) The model of a single production line is shown in Figure 2.1a. A line has two states: broken (↯) and ok (✓). In each of these states, the system can either turn a production line on to a slow mode with action Slow, turn it on to a fast mode with action Fast, switch it off with action Off, or repair it with action Repair. The failure of a production line is controlled by the environment. We assume

a failure probability of 1% when the production line is running slowly and 2% when the production line is running fast. If it is turned off, then a failure is impossible. Transitions in Figure 2.1a are labeled with actions and probabilities, e.g., the transition from state ✓ to ✓ labeled with action Slow and probability 0.99 means that we go from state ✓ with action Slow with probability 0.99 to state ✓. Note that the labels of the states (✓ and ↯) of this MDP correspond to decisions the environment can make. The actions of the MDP are the decisions the system can use to control the environment. The specification for n production lines is the synchronous product of n copies of the model in Figure 2.1a, i.e., the state space of the resulting MDP is the Cartesian product, and the transition probabilities are the product of the probabilities; for example, for two production lines, the probability to move from (✓, ✓) to (✓, ✓) on action (Slow, Slow) is 0.99^2.

The system and its environment now form a feedback loop, as depicted in Figure 2.2: First, $\mathcal{M}$ (the environment) signals its current (initial) state to T (the system). Then, T changes its state and provides an output letter, based on its own state and the state of $\mathcal{M}$. The environment $\mathcal{M}$ will read that letter, change its state probabilistically, and then provide the next output letter. The system reads this letter, changes its state, and provides the next letter. $\mathcal{M}$ reads this letter, makes a probabilistic choice based on it and its current state, and provides the next letter, and so on ad infinitum. This loop allows us to model control over the environment by the system.

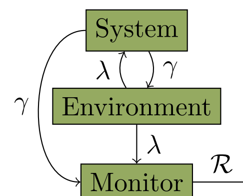


Figure 2.2: Overview

The monitor

The task of the monitor will be to measure the stream of states of the environment and the outputs of the system. We model the monitor as a transducer, but one whose output we fix to be two real numbers. These numbers model the *cost* and *reward* of the decisions of the system.

Definition 2.3 (Monitor) A monitor $\mathcal{O} = (S_\mathcal{O}, o_0, \Sigma_\mathcal{O}, \Omega_\mathcal{O}, \delta_\mathcal{O}, \gamma)$ for an MDP $\mathcal{M} = (M, m_0, A, \overline{A}, p)$ is a transducer that reads letters from $\Sigma_\mathcal{O} = M \times A$ as input and writes pairs of positive real values (i.e., $\Omega_\mathcal{O} = \mathbb{R}^{\geq 0} \times \mathbb{R}^{\geq 0}$) as output.

We sometimes write $c, r : S_{\mathcal{O}} \times \Sigma_{\mathcal{O}} \to \mathbb{R}^{\geq 0}$ for the first and second components of γ.

We use a monitor to evaluate a system with respect to a desired property. It reads words over the joint input/output alphabet and assigns a value to them. For example, the monitor for the production line controlling system reads pairs consisting of (i) a state of a production line (input of the system) and (ii) an action (output of the system). We obtain this transducer by composing transducers with a single cost function in various ways.

Example 2.2 (Monitor of a production line) In our example, we use for each production line two transducers with a single cost function to express the repair costs and the production due to this line. The transducer for the repair costs is shown in Figure 2.1b. It assigns repair costs of 10 for repairing a broken production line immediately and costs 20 for a delayed repair. If we add the numbers the transducer outputs, we obtain the repair costs of a run. For example, sequence (✓, Slow) (↯, Repair) (↯, Repair) has cost $0 + 10 + 10 = 20$. The amount of units depends on the speed of the production line. The transducer describing the number of units produced assigns value 2 if a production line is running on slow speed, 4 if it is running on fast speed, and 0 if the production line is turned off or broken.

We extend the specification to multiple production lines by building the synchronous product of copies of the transducer described above and compose the cost and reward functions in the following ways: we sum the rewards for the production and we take the maximum of repair costs of different production lines to express a discount for simultaneous repairs of more than one production line. The final specification transducer is the product of the production automaton and the repair cost automaton with (i) the repair cost as cost function and (ii) the measure of productivity as reward function.

In the current specification a system that keeps all production lines turned off has the (smallest possible) value zero, because lines that are turned off do not break down and repair is unnecessary. Therefore, we require that at least one of the lines is working. We can specify this requirement by using a qualitative specification described by a safety[1]automaton. This safety requirement can

[1] Our approach can also handle liveness specifications resulting in a Ratio-MDP with parity objective, which is then reduced to solving a sequence of MDP with mean-payoff parity objectives [CHJS10].

then be ensured by adapting the cost functions of the ratio objective [CHJS10, vEJ11]. For simplicity, we say here that any action in which all lines are turned off has an additional cost of 10.

Combining system, environment and monitor

In Section 2.2 we described how system and environment work together. Now, in addition, the system provides its output and the environment its state to the monitor. The monitor then provides two numbers in a tuple. These numbers model the cost and reward of the decision the system made in the current context. We describe this collaboration graphically in Figure 2.2. We now combine these three into one object as follows.

Definition 2.4 (Combination of system, environment and monitor) We define an *extended MDP* as the product of MDP and monitor, and the *combination* of MDP, monitor and system as follows.

1. We define the *extended MDP* of environment $\mathcal{M} = (M, m_0, A, \overline{A}, p)$ with and monitor $\mathcal{O} = (S_{\mathcal{O}}, o_0, \Sigma_{\mathcal{O}}, \Omega_{\mathcal{O}}, \delta_{\mathcal{O}}, \gamma)$ to be the MDP $\mathcal{M}' = (M', m_0', A', \overline{A}', p')$, where $M' = M \times S_{\mathcal{O}}$ is its set of states, $m_0' = (m_0, o_0)$ is its start state, $A' = A$ is its set of actions, $\overline{A}' = \overline{A}$ is its action activation function, and $p' : M' \times A' \times M'$ is its probabilistic transition function, where $p'((m, o), a, (m', o')) = p(m, a, m')$ if $o' = \delta_{\mathcal{O}}(o, (m, a))$ and zero otherwise.

   This combination also induces an output function $\gamma_{\mathcal{M}} : M' \times A \to \mathbb{R}^{\geq 0} \times \mathbb{R}^{\geq 0}$ of the MDP. The output function is defined as the output of the monitor in the same context, i.e.,$\gamma_{\mathcal{M}}((m, o), a) = \gamma(o, (m, a))$. As for monitors, we often use $c, r : C \to \mathbb{R}^{\geq 0}$ as shorthands for the cost and reward parts of $\gamma_{\mathcal{M}}$.

2. The *combination* of system $T = (S, s_0, \Sigma, \Omega, \delta, \gamma)$, environment $\mathcal{M} = (M, m_0, A, \overline{A}, p)$ with labelling λ and monitor $\mathcal{O} = (S_{\mathcal{O}}, o_0, \Sigma_{\mathcal{O}}, \Omega_{\mathcal{O}}, \delta_{\mathcal{O}}, \gamma)$ is defined as a *Markov chain*, i.e., as a tuple $\mathcal{C} = (C, c_0, p_{\mathcal{C}})$, where $C = S \times M \times S_{\mathcal{O}}$ is its set of states, $c_0 = (s_0, m_0, o_0)$ is its initial state and $p_{\mathcal{C}} : C \times C \to [0, 1]$ is its probabilistic transition function.

   The probabilistic transition function $p_{\mathcal{C}}$ models the progression of system, environment and monitor in lockstep, i.e., $p_{\mathcal{C}}((s, m, o), (s', m', o')) =$

$p(m, w, m')$ if $s' = \delta(s, \lambda(m))$ is the next state of T, based on its current state and the labeling of the state of the environment, and $o' = \delta_{\mathcal{O}}(o, (m, \gamma(s)))$ is the next state of the monitor, based on its current state, the state of the environment and the output of the system. Otherwise the value of $p_{\mathcal{C}}$ is zero.

We denote by $\overline{p_{\mathcal{C}}} : C^* \to [0, 1]$ the canonical extension of $p_{\mathcal{C}}$ to *finite runs*, i.e., $\overline{p_{\mathcal{C}}}(w_0 w_1 \ldots w_n) = \prod_{i=0}^{n-1} p_{\mathcal{C}}(w_i, w_{i+1})$, and $\overline{p_{\mathcal{C}}}(w) = 0$ for all other runs (i.e., runs that do not start in the initial state).

This combination also induces a output function $\gamma_{\mathcal{C}} : C \to \mathbb{R}^{\geq 0} \times \mathbb{R}^{\geq 0}$ on the Markov chain. The output function is defined as the output of the monitor in the same context, i.e., $\gamma_{\mathcal{C}}(s, m, o) = \gamma(o, (m, \gamma(s)))$. As for monitors, we often use $c, r : C \to \mathbb{R}^{\geq 0}$ as shorthands for the first and second part of $\gamma_{\mathcal{C}}$.

We sometimes interpret the probabilistic transition function as a matrix, i.e., we enumerate the state space from 1 to $n := |C|$, and interpret $p_{\mathcal{C}}$ as an $n \times n$ matrix, where the entry in row i and column j has value $p_{\mathcal{C}}(m_i, m_j)$. Analogously, we can interpret every function $f : C \to \mathbb{R}$ as a row or column vector of dimension n, where entry i has value $f(m_i)$.

Example 2.3 (State transition probabilities of lines) This combination provides us with a probability distribution over the development of system, environment and monitor over time. For instance, the probability of moving from $((\checkmark, \checkmark), (s_0, s_0))$ to $((\lightning, \lightning), (s_0, s_0))$ when choosing (Slow, Slow) is 0.01^2, while the probability of moving to $((\lightning, \lightning), (s_1, s_1))$ is 0 because we cannot move from s_0 to s_1 with this input.

Measuring efficiency

While we now have a way to measure local decisions, we are still lacking a means to measure the global, long-run quality[2] of the system. To that end, we will use the expected ratio payoff (see Definition 1.14 and Definition 1.15). Recall, that the ratio payoff of a run ρ is defined as

$$\mathcal{R}_{\frac{c}{r}}(\rho) = \lim_{l \to \infty} \liminf_{u \to \infty} \frac{\sum_{i=l}^{u} c(\rho_i)}{1 + \sum_{i=l}^{u} r(\rho_i)}$$

We often leave out c and r if they are clear from context.

[2] Pun intended

Intuitively, $\mathcal{R}$ computes the long-run ratio between the costs and rewards accumulated along a run. We divide costs by rewards, i.e., the higher the efficiency of a run the lower the ratio. Therefore, in the rest of this paper we try to minimize the ratio.

Definition 2.5 (Efficiency/Ratio of a combination) Given a combination $\mathcal{C}$, we define the efficiency or ratio of the combination as $\mathbb{E}_{\mathcal{C}}[\mathcal{R}]$.

Lemma 2.1 (Expected ratio exists) *The expected ratio* $\mathbb{E}_{\mathcal{C}}[\mathcal{R}]$ *exists since* $\mathcal{R}$ *is bounded from below by zero.*

We now can ask for an efficient system in a probabilistic environment. We model the system and the monitor as transducers and the environment as an MDP. We evaluate the performance of a system in this context as its expected efficiency, modeled by the expected ratio of costs and rewards. In the next section we will analyze the combination of the three components and show the theory necessary to find the optimal system for an environment and a monitor.

2.3 Analysis

In this section, we will lay the foundations of the algorithms in Section 2.4. We will first show that pure strategies are sufficient for the ratio objective. Thus we will make our search for the most efficient system simpler. We will further show how to calculate the expected ratio of pure strategies. On the basis of these results, we will look for algorithmic solutions to this search in Section 2.4.

Strategies and systems

To find a system T such that the combination of T, environment $\mathcal{M}$ and monitor $\mathcal{O}$ is optimal, we combine $\mathcal{M}$ and $\mathcal{O}$ to obtain a new MDP as defined in Definition 2.4 (1). We will then look for an optimal *strategy* in the resulting MDP.

Definition 2.6 (Strategy (Policy)) A *strategy* (or *policy*) for an MDP $\mathcal{M} = (M, m_0, A, \overline{A}, p)$ is a function $d : (M \times A)^* M \to \mathcal{D}(A)$ that assigns a probability distribution to all finite sequences in $(M \times A)^* M$ such that only active actions are chosen, i.e., for all sequences $w \in (M \times A)^*$, states $m \in M$ and actions $a \in A$ such that $d(wm)(a) > 0$ we have $(m, a) \in \overline{A}$.

A strategy such that the co-domain of $d(\rho)$ is $\{0,1\}$ for all $\rho \in (M \times A)^*M$ is called *deterministic*. A strategy that can be defined using domain M is called *memoryless*. A memoryless, deterministic strategy is called *pure*. We denote the set of pure strategies by $D(\mathcal{M})$.

Note that the previous definition of a strategy diverts slightly from that used in Chapter 1. A strategy in Chapter 1 was defined as $d : (M \cup A)^* M \to \mathcal{D}(M \cup A)$. Due to the way MDPs are defined as a subclass of stochastic games in Chapter 1 both definitions are equivalent.

Like transducers in Definition 2.4, strategies induce Markov chains.

Definition 2.7 (Induced Markov chain) Let $\mathcal{M}$ be an MDP and d be a pure strategy for $\mathcal{M}$. Then by $\mathcal{M}_d = (M, m_0, p_\mathcal{C})$ we denote the *induced Markov chain*, where $\mathcal{M}$ and $\mathcal{M}_d$ have the same set of states and same start state and the probability function is defined by d, i.e., $p_\mathcal{C}(m, m') = p(m, d(m), m')$ for all states $m, m' \in M$.

We are now going to prove that for every pure strategy (i.e., a function getting states as input) there is a transducer (i.e., a function getting sequences of labels as input), such that the two induce the same Markov chain and therefore the same expected ratio. This proof is required because a system reads labels from the Markov decision process as input, not its states.

Lemma 2.2 (Pure strategies are implementable by transducers) *Let $\mathcal{M}$ be an MDP and let $d : M \to A$ be a pure strategy for $\mathcal{M}$. Then for any ratio function $\mathcal{R}$ there is a transducer T such that for the combination $\mathcal{C} = (C, c_0, p_\mathcal{C})$ of T and $\mathcal{M}$ we have that $\mathbb{E}_\mathcal{C}[\mathcal{R}] = \mathbb{E}_{\mathcal{M}_d}[\mathcal{R}]$.*

Proof Let $\mathcal{C} = \mathcal{M}_d$. Let $\lambda : M \to 2^A$ be a labeling and $\lambda^* : M^* \to (2^A)^*$ be its canonical extension to words. We define $\lambda^{-1} : (2^A)^* \to M$ by $\lambda^{-1}(\lambda(m_0)) = m_0$ and $\lambda^{-1}(wi) = m$ for each word w and label i such that $wi = \lambda^*(\rho)$ for some run ρ of $\mathcal{M}_d$ with probability greater than zero, where m is the only state s.t. $p_\mathcal{C}(\lambda^{-1}(w), d(\lambda^{-1}(w)), m) > 0$ and $\lambda(m) = i$. For every other word, the definition of λ^{-1} is arbitrary. λ^{-1} thus identifies the last state of a run that a labeling sequence has come from. We will show later that there is indeed only one such state.

Let $d' = d \circ (\lambda^{-1})^*$. We now have to prove two claims: (1) $\lambda^{-1}(w)$ is well-defined and (2) d' is sufficient for the claim.

We prove (1) by induction over w. For $|w| = 1$, this follows from the first and last case of the definition. Assume the claim has been shown for w. We are now going to show that $\lambda^{-1}(wi)$ is well-defined. For that is it sufficient to show that only one such m as in the definition exists. Let $m = \lambda^{-1}(w)$. Assume that there exist $m' \neq m'' \in M$ such that $p(m, d(m), m') > 0 \wedge p(m, d(m), m'') > 0$ such that $\lambda(m') = \lambda(m'')$. This contradicts the definition of a labeling. Therefore, $m' = m''$ and λ^{-1} is well-defined.

For (2) it is sufficient to show that $d(\rho) = d'(\lambda^*(\rho))$ and that d' can be implemented by a transducer. Since we have that $d' \circ \lambda^* = d \circ (\lambda^{-1})^* \circ \lambda^*$, it is sufficient for the first to show that $(\lambda^{-1})^* \circ \lambda^*$ is the identity function on all runs with probability 0. This follows by induction over ρ. For the implementation of d' as a transducer it is sufficient to see that an implementation in the worst case has to keep track of the whole MDP to know exactly in which state the MDP currently is. ■

We are looking for an optimal pure strategy for the MDP constructed from the environment model and the monitor. In the next subsection we will show that there always exists an optimal pure strategy.

Pure strategies are sufficient

In [Gim07], Gimbert proves that in an MDP any function mapping sequences of states of that MDP to $\mathbb{R}$ that is *submixing* and *prefix independent* admits optimal pure strategies. Since our function $\mathcal{R}$ may also take the value ∞, we cannot apply the result immediately. However, since $\mathcal{R}$ maps only to non-negative values and the set of measurable functions is closed under addition, multiplication, limit inferior and superior and division, provided that the divisor is not equal to 0, the expected value of $\mathcal{R}$ is always defined and the theory presented in [Gim07] also applies in this case. Furthermore, to adapt the proof of [Gim07] to minimizing the function instead of maximizing it, one only needs to inverse the used inequalities and replace max by min. It remains to show that $\mathcal{R}$ fulfills the following two properties.

Lemma 2.3 ($\mathcal{R}$ is submixing and prefix independent) *Let $\mathcal{M} = (M, m_0, A, \overline{A}, p)$ be a MDP and ρ be a run.*

1. *For every $i \geq 0$ the prefix of ρ up to i does not matter, i.e., $\mathcal{R}(\rho) = \mathcal{R}(\rho_i \rho_{i+1} \dots)$.*

2. *For every sequence of non-empty runs* $u_0, v_0, u_1, v_1 \dots \in (A \times M)^+$ *such that* $\rho = u_0 v_0 u_1 v_1 \dots$ *we have that the function of the sequence is greater than or equal to the maximal ratio of sequences* $u_0 u_1 \dots$ *and* $v_0 v_1 \dots$, *i.e.,* $\mathcal{R}(\rho) \geq \min\{\mathcal{R}(u_0 u_1 \dots), \mathcal{R}(v_0 v_1 \dots)\}$.

Proof The first property follows immediately from the first limit in the definition of $\mathcal{R}$.

For the second property we partition $\mathbb{N}$ into U and V such that U contains the indexes of the parts of ρ that belong to a u_k for some $k \in \mathbb{N}$ and such that V contains the other indexes. Formally, we define $U := \bigcup_{i \in \mathbb{N}} U_i$ where $U_0 := \{k \in \mathbb{N} \mid 0 \leq k < |u_0|\}$ and $U_i := \{\max(U_{i-1}) + |v_{i-1}| + k \mid 1 \leq k \leq |u_i|\}$. Let $V := U \setminus \mathbb{N}$ be the other indexes.

Now we look at the value from m to l for some $m \leq l \in \mathbb{N}$, i.e. $\mathcal{R}_m^l := (\sum_{i=m\dots l} c(\rho_i))/(1 + \sum_{i=m\dots l} r(\rho_i))$. We can divide the sums into two parts, the one belonging to U and the one belonging to V and we get

$$\mathcal{R}_m^l = \frac{\left(\sum_{i \in \{m\dots l\} \cap U} c(\rho_i)\right) + \left(\sum_{i \in \{m\dots l\} \cap V} c(\rho_i)\right)}{1 + \left(\sum_{i \in \{m\dots l\} \cap U} r(\rho_i)\right) + \left(\sum_{i \in \{m\dots l\} \cap V} r(\rho_i)\right)}$$

We now define the sub-sums between the parentheses as $u_1 := \sum_{i \in \{m\dots l\} \cap U} c(\rho_i)$, $u_2 := \sum_{i \in \{m\dots l\} \cap U} r(\rho_i)$, $v_1 := \sum_{i \in \{m\dots l\} \cap V} c(\rho_i)$ and $v_2 := \sum_{i \in \{m\dots l\} \cap V} r(\rho_i)$. Then we obtain

$$\mathcal{R}_m^l = \frac{u_1 + v_1}{1 + u_2 + v_2}$$

We will now show

$$\mathcal{R}_m^l \geq \min\left\{\frac{u_1}{u_2 + 1}, \frac{v_1}{v_2 + 1}\right\}$$

Without loss of generality we can assume $u_1/(u_2 + 1) \geq v_1/(v_2 + 1)$, then we have to show that

$$\frac{u_1 + v_1}{1 + u_2 + v_2} \geq \frac{v_1}{v_2 + 1}.$$

This holds if and only if $(u_1 + v_1)(1 + v_2) = u_1 + v_1 + u_1 v_2 + v_1 v_2 \geq v_1 + v_1 u_2 + v_1 v_2$ holds. By subtracting v_1 and $v_1 v_2$ from both sides we obtain $u_1 + u_1 v_2 = u_1(1 + v_2) \geq u_2 v_1$. If u_2 is equal to 0 then this holds because u_1 and v_2 are greater than or equal to 0. Otherwise, this holds if and only if $u_1/u_2 \geq v_1/(1 + v_2)$ holds. In general, we have $u_1/u_2 \geq u_1/(u_2 + 1)$. From the assumption we have $u_1/(u_2 + 1) \geq v_1/(v_2 + 1)$ and hence $u_1/u_2 \geq v_1/(v_2 + 1)$.

The original claim follows because we have shown this for any pair of m and l. ■

Theorem 2.1 (There is always a pure optimal strategy) *For each MDP with the ratio function, there is a pure optimal strategy.*

PROOF See [Gim07] and the last lemma. ■

This theorem allows us to restrict the search for an optimal strategy (and therefore optimal system) to a finite set of possibilities. In the next subsection, we show how to calculate the expected ratio of a pure strategy. Then, in the next section, we will show algorithms that perform better than brute force search.

Expected ratio of pure strategies

To calculate the expected value of a pure strategy, we use the fact that an MDP with a pure strategy induces a Markov chain and that the runs of a Markov chain have a special property, which we can use to calculate the expected value. We will first show how to calculate the expected value on a *unichain* MC, and will then extend the result to any kind of Markov chain.

Definition 2.8 (Random variables of MCs [Put94]) Let $p^n_{\mathcal{C}}(m)$ be the probability of being in state m at step n and let $\pi(m) := lim_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n-1} p^i_{\mathcal{C}}(m)$. This is called the *steady state distribution* of $p^n_{\mathcal{C}}$. Let ν^n_m denote the *number of visits* to state m up to time n.

Definition 2.9 (Properties of MCs [Put94]) Let $\mathcal{C} = (M, m_0, p_{\mathcal{C}})$ be a Markov chain. A state $m \in M$ is called *transient*, if the probability of it occurring infinitely often in a run of $\mathcal{C}$ is equal to zero. Otherwise it is called *recurrent.*

A subset of states S of $\mathcal{C}$ is called *recurrence class* if all states can reach each other, all states are recurrent, and there is no such set of states S' such that $S \subset S'$.

We say that a Markov chain is *unichain* if it has at most one recurrence class. We call an MDP unichain if every strategy induces a unichain MC.

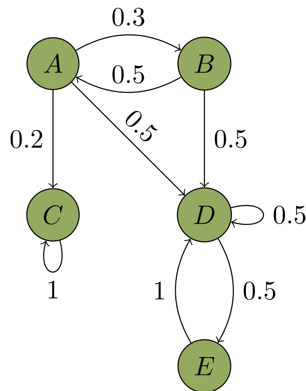


Figure 2.3: Markov chain with transient states A and B, and two recurrence classes $\{C\}$ and $\{D, E\}$.

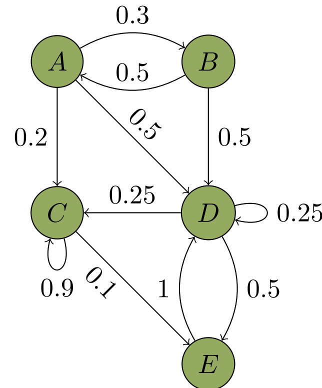


Figure 2.4: Markov chain with transient states A and B,

Example 2.4 Consider the Markov chain in Figure 2.3. An infinite run that starts in (A) or (B) can visit (A) infinitely often in principle, but it does so with probability 0, because at some point it will take the transition with to (C) or (D) with probability 1. On the other hand, once it reaches (C) it will visit state C infinitely often. Likewise, once it reaches (D), it will visit both (D) and (E) infinitely often. Hence, (C), (D) and (E) are recurrent. But they do not belong to the same recurrence class: it is impossible to reach (C) from (D) or (E), and vice versa.

Compare this to Figure 2.4. (A) and (B) are still transient states, and (C), (D) and(E) are still recurrent. But now (C), (D) and (E) can reach each other and are therefore in the same recurrence class.

We have the following lemma describing the long-run behavior of Markov chains [Tij03, Nor03].

Definition 2.10 (Well-behaved runs) Let ρ be an infinite run of a unichain Markov chain. Then we call this run *well-behaved* if $\lim_{l\to\infty} \frac{\nu_m^l}{l} = \pi(m)$.

Lemma 2.4 (Runs are well-behaved almost surely [Put94]) *A randomly selected run of a unichain MC is well-behaved almost surely, i.e.,* $\mathrm{P}(\textit{well-behaved}) = 1$.

This lemma guarantees that an infinite run will always visit the states in exactly the proportion as the expected number of visits, i.e., as the steady state

distribution proscribes. This is a non-trivial result, as the following example shows.

Example 2.5 Consider states (D) and (E) of Figure 2.3. A run just in these two states may take on many shapes. For example, it is possible that a run visits (D) infinitely often. Or that it always visits state (D) ten times and then visits state (E) one time. One might argue that each of these run has probability 0, but then very infinite run has probability 0.

When we calculate the expected ratio, we only need to consider well-behaved runs as shown in the following lemma.

Lemma 2.5 *Let $\mathcal{C} = (M, m_0, p_\mathcal{C})$ be a Markov chain, let $\mathcal{P} = (\Omega, \mathcal{F}, \mu)$ denote its induced probability space, and let N denote the set of runs that are not well-behaved. Then*

$$\mathbb{E}_\mathcal{C}[\mathcal{R}] = \int_{\Omega \setminus N} \mathcal{R}\ d\mu$$

Proof According to the definition of the expected value of a Markov chain on Page 7, the expected value is defined as $\mathbb{E}[\mathcal{R}] = \int_\Omega \mathcal{R}\ d\mu$. According to a well known property of Lebesgue integrals, we can ignore events having probability 0 when calculating the integral, i.e., $\int_\Omega \mathcal{R}\ d\mu = \int_{\Omega \setminus N} \mathcal{R}\ d\mu$ for any set of events N with $\mu(N) = 0$. From Lemma 2.4, it follows that the set of runs that are not well-behaved has probability zero. ■

For a well-behaved run, i.e., for every run that we need to consider when calculating the expected value, we can calculate the ratio in the following way.

Lemma 2.6 (Calculating the ratio of a well-behaved run) *Let ρ be a well-behaved run of a unichain Markov chain $\mathcal{C} = (M, m_0, p_\mathcal{C}, \gamma_\mathcal{C})$. Recall that we denote by c the first cost of $\gamma_\mathcal{C}$, and by r the reward.*

$$\mathcal{R}(\rho) = \frac{\sum_{m \in M} \pi(m) c(m)}{\lim_{l \to \infty} \frac{1}{l} + \sum_{m \in M} \pi(m) r(m)}$$

Proof By definition of $\mathcal{R}$ we have

$$\mathcal{R}(\rho) = \lim_{m \to \infty} \liminf_{l \to \infty} \frac{\sum_{i=l}^{m} c(\rho_i)}{1 + \sum_{i=l}^{m} r(\rho_i)}$$

To get rid off the outer limit, we are going to assume, without loss of generality, that there are no transient states. We can do this because every transient

state will not influence $\mathcal{R}(\rho)$ because ρ is well-behaved and because $\mathcal{R}$ is prefix independent.

$$\mathcal{R}(\rho) = \liminf_{l\to\infty} \frac{\sum_{i=0}^{l} c(\rho_i)}{1 + \sum_{i=0}^{l} r(\rho_i)}$$

We can calculate the sums in a different way: we take the sum over the states and count how often we visit one state, i.e.,

$$\frac{\sum_{i=0}^{l} c(\rho_i)}{1 + \sum_{i=0}^{l} r(\rho_i)} = \frac{\sum_{m\in M} c(m)\nu_m^l}{1 + \sum_{m\in M} r(m)\nu_m^l} = \frac{\sum_{m\in M} c(m)(\nu_m^l/l)}{1/l + \sum_{m\in M} r(m)(\nu_m^l/l)}$$

We will now show that the sequence converges for lim instead of lim inf. But if a sequence converges for lim, then it also converges to lim inf, and the two limits have the same value. Because both the numerator and the denominator are finite values we can safely draw the limit into the fraction, i.e.,

$$\begin{aligned}(\dagger)\lim_{l\to\infty}\left(\frac{\sum_{m\in M} c(m)(\nu_m^l/l)}{1/l + \sum_{m\in M} r(m)(\nu_m^l/l)}\right) &= \frac{\lim_{l\to\infty}\left(\sum_{m\in M} c(m)(\nu_m^l/l)\right)}{\lim_{l\to\infty}\left(1/l + \sum_{m\in M} r(m)(\nu_m^l/l)\right)}\\ &= \frac{\sum_{m\in M} c(m)\lim_{l\to\infty}(\nu_m^l/l)}{\lim_{l\to\infty}(1/l) + \sum_{m\in M} r(m)\lim_{l\to\infty}(\nu_m^l/l)}\\ &\overset{\ddagger}{=} \frac{\sum_{m\in M} c(m)\pi(m)}{\lim_{l\to\infty}(1/l) + \sum_{m\in M} r(m)\pi(m)}\end{aligned}$$

Equality ‡ holds because we have $\lim_{l\to\infty}\frac{\nu_m^l}{l} = \pi(m)$ by Lemma 2.4. The limit diverges to ∞ if and only if the rewards are all equal to zero and at least one cost is not. In this case the original definition of $\mathcal{R}$ diverges and hence $\mathcal{R}$ and the last expression are the same. Otherwise the last expression converges, so † converges, and so lim inf and lim of this sequence are the same. ■

Note that the previous lemma implies that the value of a well-behaved run is independent of the actual run. In other words, on the set of well-behaved runs of a unichain Markov chain the ratio function is constant. So the expected value of such a Markov chain is equal to the ratio of any of its well-behaved runs.

Theorem 2.2 (Expected ratio of a unichain MC) *Let $\mathcal{C} = (M, m_0, p_{\mathcal{C}})$ be a unichain MC and let π denote the Cesaro limit of $p_{\mathcal{C}}^n$ of the induced Markov chain. Then the expected ratio can be calculated as follows.*

$$\mathbb{E}_{\mathcal{C}}[\mathcal{R}] = \frac{\sum_{m\in M} c(m)\pi(m)}{\lim_{l\to\infty}(1/l) + \sum_{m\in M} r(m)\pi(m)}$$

As a special case, when $r(m) = 1$ for all states, we can compute the mean payoff [Put94] as follows.

$$\mathbb{E}_{\mathcal{C}}[\mathcal{P}] = \sum_{m \in M} c(m)\pi(m)$$

PROOF This follows from Lemma 2.6 and the fact that $\mathcal{R}$ is constant on a unichain Markov chain (i.e., independent from the actual run). ■

Note that this means that an expected value is ∞ if and only if the reward of every action in the recurrence class of the Markov chain is 0 and there is at least one cost that is not.

This provides us with an efficient method of calculating the expected ratio of a unichain MC. We can calculate π by solving the linear equation system $\pi(P - I) = 0$ [Put94], where P is the probability matrix of $\mathcal{C}$ (Definition 2.4).

Each run of a MC will almost surely end in one recurrence class (the probability of visiting only transient states is equal to zero). And since $\mathcal{R}$ is prefix-independent, the ratio of this run will be equal to the ratio of the run inside the recurrence class.

Theorem 2.3 (Expected ratio of a MC) *Let $\mathcal{C}$ be a MC. For each recurrence class $\mathcal{C}'$, let $\pi(\mathcal{C}')$ be the probability of reaching $\mathcal{C}'$.*

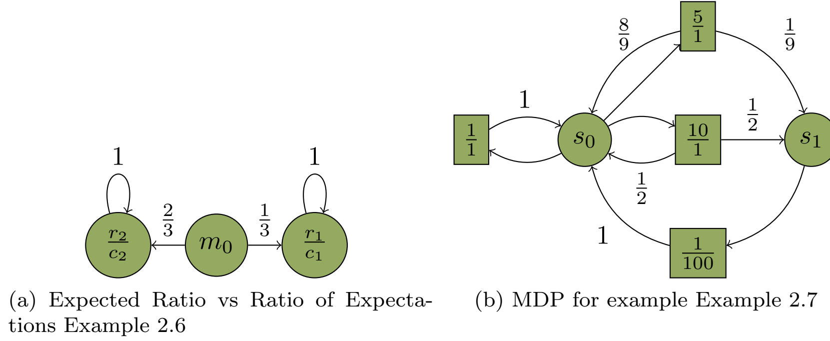
$$\mathbb{E}_{\mathcal{C}}[\mathcal{R}] = \sum_{\mathcal{C}' rec.\ class} \pi(\mathcal{C}')\mathbb{E}_{\mathcal{C}'}[\mathcal{R}],$$

where $\mathcal{C}'$ ranges over all recurrence classes of $\mathcal{C}$ and $\mathbb{E}_{\mathcal{C}'}[\mathcal{R}]$ denotes the expected ratio of the MC consisting only of recurrence class $\mathcal{C}'$.

Difference between ratio and mean payoff

Note that Theorem 2.3 also hints at the difference between the expected ratio and the ratio between expectations. The following example shows that straightforward reduction from MDPs with the ratio function to MDPs with the mean-payoff function is not possible.

Example 2.6 (Expected ratio vs ratio of expectations) In Figure 2.5a we have a Markov chain with three states. m_0 is the initial state, and the states labeled with $\frac{r_1}{c_1}$ and $\frac{r_2}{c_2}$ are reached with probability 1/3 and 2/3, respectively. The labels also define the rewards and costs of each state.



(a) Expected Ratio vs Ratio of Expectations Example 2.6

(b) MDP for example Example 2.7

Figure 2.5: Two examples showing that the Ratio objective cannot be easily reduced to the Mean objective

From the previous theorem it follows that we have $\mathbb{E}[\mathcal{R}] = 1/3 \cdot r_1/c_1 + 2/3 \cdot r_2/c_2$. Note that this is not the same as dividing the expected average cost by the expected average reward $\frac{\mathbb{E}[\mathcal{P}_c]}{\mathbb{E}[\mathcal{P}_r]} = \frac{1/3 \cdot c_1 + 2/3 \cdot c_2}{1/3 \cdot r_1 + 2/3 \cdot r_2}$ (i.e., the ratio of expected average rewards and costs) for appropriate r_1, r_2, c_1 and c_2.

It is also not possible to just subtract costs from rewards and obtain the same result. Recall the ACTS unit from Section 2.1. We want to optimize the relation of two measures: speed (km/h) and fuel consumption (l). When subtracting kilometers per hour from liters, the value of the optimal controller has no intuitive meaning. Furthermore, it can lead to non-optimal strategies, as shown by the following example.

Example 2.7 (Subtraction leads to different strategies) Consider an MDP with two states, s_0 and s_1, as depicted in Figure 2.5b. There is one action enabled in s_1. It has cost 1 and reward 100 and leads with probability 1 to s_0. There are three actions in s_0: Action a_0 has cost 5 and reward 1 and leads with probability 1/9 to s_1 and with 8/9 back to s_0. Action a_1 has cost 10 and reward 1 and leads with probability 1/2 to s_1 and with 1/2 to s_0. Action a_2 has cost and reward 1 and leads with probability 1 back to s_0. We will ignore this action for the remainder of this example.

The steady state distribution of the strategy choosing a_0 is $(9/10, 1/10)$, and so its ratio value is $(9/10 \cdot 5 + 1/10 \cdot 1)/(9/10 \cdot 1 + 1/10 \cdot 100) \approx 0.42$. For the strategy choosing a_1, the steady state distribution is $(2/3, 1/3)$ and the ratio value is $(2/3 \cdot 10 + 1/3 \cdot 1)/(2/3 \cdot 1 + 1/3 \cdot 100) \approx 0.634$, which is larger than the value for a_1. Hence choosing a_0 is the better strategy for the

ratio objective. If we now subtract the reward from the cost and interpret the result as a Mean-Payoff MDP, then we get rewards 4, 9, and -99 respectively. Choosing strategy a_0 gives us $9/10 \cdot 4 - 1/10 \cdot 99 = -6.3$, while choosing strategy a_1 gives us $2/3 \cdot 9 - 1/3 \cdot 99 = -27$. So, choosing a_1 is the better strategy for the average objective.

These two examples show that we cannot easily reduce the ratio payoff to mean-payoff.

2.4 Algorithms

In this section we discuss three algorithms calculating most efficient strategies for MDPs. In all of them, we first decompose the MDPs into strongly connected components (called end-components) and then calculate optimal strategies for each component. Finally we compose the resulting strategies into one optimal strategy for the complete MDP.

We will first discuss end-components. Then we will define the common structure for all algorithms. Afterwards we will discuss three ways to compute optimal strategies for end-components. Finally, we will evaluate the performances of all three algorithms and discuss their implication.

End-components

In [dA97], the author defines *end-components* as follows.

Definition 2.11 (End-component) Let $\mathcal{M} = (M, m_0, A, \overline{A}, p)$ be an MDP. A subset of its states $M' \subseteq M$ is called an *end-component* if

- for each pair of states $m, m' \in M'$ there is a strategy such that a run starting at m will reach m' with probability greater zero, and
- for each state $m \in M'$ there is an action $a \in A$ such that for all states $m' \in M$ with $p(m, a, m') > 0$ we have $m' \in M'$.

An end-component is called *maximal* if there is no other end-component that contains all its states.

Example 2.8 Figure 2.6 illustrates an MDP with two end-components (inside the boxes)). The left end-component consists of two states: s_0 and s_2. s_0

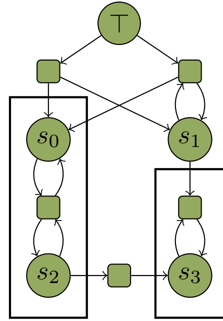


Figure 2.6: Illustration of maximal end-components. States s_0 and s_2 together form a maximal end-component, while state s_3 forms a maximal end-component by itself.

only has one possible choice: it has to go to the action below it, from which the next state is chosen probabilistically. However, s_2 has two possible choices: it can go up to the same action, or go right, from which the next state will be s_3. So both states in this end-component can reach each other with probability one. Both states have an action to stay inside the end-component. However, s_2 does not have to. There also is a strategy allowing it to only pass through this end-component, instead of remaining in it. So, while every run has to end in an end-component, a run that enters an end-component does not have to stay there. Note fruther that state s_1 is contained in no end-component, although it can reach itself by picking the action above itself. While there is an action s_1 can back that leads back to s_1, there is no strategy that enforces such a visit.

Lemma 2.7 (End-components allow optimal unichain strategies) *Let $\mathcal{M}$ be an end-component, and let d be a non-unichain strategy for $\mathcal{M}$. Then there is a unichain strategy d' with expected ratio that is as least as good as that of d.*

Proof Lemma 2.3 and Definition 2.11 allow us to construct a unchain strategy from an arbitrary pure strategy with the same or a better value: d' fixes the recurrent class M' with the minimal value induced by d; for states outside of M', d' plays a strategy to reach M' with probability 1. ■

```
    Input: MDP 𝓜, start state o_0
    Output: Value 𝔼[𝓡] and optimal strategy d
 1  ecSet ← decompose(𝓜);
 2  foreach i ← [0 ... |ecSet| − 1] do
 3  |  switch ecSet_i do
 4  |  |  case isZero :  λ_i ← 0; d_i ← zero-cost strategy;
 5  |  |  ;
 6  |  |  case isInfty :  λ_i ← ∞; d_i ← arbitrary; ;
 7  |  |  ;
 8  |  |  otherwise  : d_i ← solveEC(ecSet_i); ;
 9  |  endsw
10  end
11  d ← compose(𝓜, λ_0, ..., λ_{|ecSet|−1}, d_0, ..., d_{|ecSet|−1});
```

Algorithm 2.1: Finding optimal strategies for MDPs

```
    Input: MDP 𝓜, start state o_0
    Output: Set L of maximal end-components
 1  L ← {𝓜};
 2  while L cannot be changed anymore do
 3  |  𝓜' ← some element of L;
 4  |  Deactivate all actions that lead outside of 𝓜';
 5  |  Let 𝓜_1, ..., 𝓜_n be the strongly connected components of 𝓜';
    |  L ← L \ {𝓜'} ∪ {𝓜_1, ..., 𝓜_n};
 6  end
```

Algorithm 2.2: Decomposition into maximal end-components

General algorithm structure

As Lemma 2.7 shows, we can look for unichain strategies in the end-components and then compose these strategies into an optimal strategy for the whole MDP. The general shape of the algorithms is shown in Algorithm 2.1. In Line 1 we decompose the MDP into maximal end-components [dA97] (see Algorithm 2.2). Then we analyze each end-component separately: the predicates `isZero` and `isInfty` (Line 4 and 6, resp.) check if an end-component has value zero or infinity. This is necessary because the algorithms calculating optimal strategies for end-components (`solveEC`, Line 8) only work if a strategy with finite ratio exists and if the optimal strategy has ratio greater than zero. Finally, function `compose` (Line 11) takes values and strategies from all end-components and computes an optimal strategy for $\mathcal{M}$ using Lemma 2.10.

Decomposing MDPs

Decomposition into maximal end-components, due to [dA97], happens in a sequence of refinements of MDPs, until no further refinement is possible. We describe this formally in Algorithm 2.2.

Simple checks `isZero` and `isInfty`

Functions `isZero` and `isInfty` can be implemented efficiently as follows.

Lemma 2.8 *For every MDP $\mathcal{M} = (M, m_0, A, \overline{A}, p)$ such that M is an end-component of $\mathcal{M}$, we can check efficiently if the value of $\mathcal{M}$ is zero or infinity and construct corresponding strategies.*

PROOF $\mathcal{M}$ has value zero if there exists a strategy such that the expected average reward w.r.t. the cost function c is zero. We check this by removing all actions from states in $\mathcal{M}$ that have $c > 0$ and then recursively removing all actions that lead to a state without enabled actions. If the resulting MDP $\mathcal{M}'$ is non-empty, then there is a strategy with value 0 for the original end-component. It can be computed by building a strategy that moves to and stays in $\mathcal{M}'$.

$\mathcal{M}$ has value infinity iff (i) for every strategy the expected average reward w.r.t. cost function c is not zero, i.e., $\mathcal{M}$ has not value zero, and (ii) for all strategies the expected average reward w.r.t. the reward function r is zero. This can only be the case if for all actions in the end-component the value of cost function r is zero. In this case, any arbitrary strategy will give value infinity. ■

Algorithms for end-components

We will now discuss three algorithms for end-components. For all of them, we assume that there exists a strategy with a finite ratio value and that the optimal strategy does not have value zero. The first two solutions are based on reduction to linear programs. The last solution is a new algorithm based on strategy (or policy) iteration.

Fractional linear program

Using Theorem 2.3, we transform the MDP into a fractional linear program. This is done in the same way as is done for the expected average payoff case (cf. [Put94]). We define variables $x(m,a)$ for every state $m \in M$ and every available actions $a \in \overline{A}(m)$. This variable intuitively corresponds to the probability of being in state m and choosing action a at any time. Then we have for example $\pi(m) = \sum_{a\in\overline{A}(m)} x(m,a)$.

We need to restrict this set of variables. First of all, we always have to be in some state and choose some action, i.e., the sum over all $x(m,a)$ has to be one. The second set of restrictions ensures that we have a steady state distribution, i.e., the sum of the probabilities of going out of (i.e., being in) a state is equal to the sum of the probabilities of moving into this state.

Definition 2.12 (Fractional LP for MDP) Let $\mathcal{M}$ be a unichain MDP such that every Markov chain induced by any strategy contains at least one non-zero reward. Then we define the following fractional linear program for it.

$$\text{Minimize } \frac{\sum_{m\in M}\sum_{a\in\overline{A}(m)} x(m,a)c(m,a)}{\sum_{m\in M}\sum_{a\in\overline{A}(m)} x(m,a)r(m,a)}$$

subject to

$$\begin{aligned} \sum_{m\in M}\sum_{a\in\overline{A}(M)} x(m,a) &= 1 \\ \sum_{a\in\overline{A}(m)} x(m,a) &= \sum_{m'\in M}\sum_{a\in\overline{A}(m')} x(m',a)p(m',a,m) \quad \forall m \in M \end{aligned}$$

There is a correspondence between pure strategies and basic feasible solutions to the linear program[3]. That is, the linear program always has a solution because every positional strategy corresponds to a solution. See [Put94] for a detailed analysis of this in the expected average reward case that also applies here.

Once we have calculated a solution of the linear program, we can calculate the strategy as follows.

Definition 2.13 (Strategy from solution of LP) Let $x(m,a)$ be the solutions to the linear program. Let $M' = \{m \in M \mid \exists a \in A : x(m,a) > 0\}$. Then we define strategy d as $d(m) = a$ for all states $m \in M$ and the only possible

[3] A feasible solution is an assignment that fulfills the linear equations

$a \in A$ such that $x(m, a)$. For all other states, choose a strategy such that M' is reached with probability 1 (Lemma 2.7).

Note that this is well defined because for each state m there is at most one action a such that $x(m, a) > 0$ because of the bijection (modulo the actions of transient states) between basic feasible solutions and strategies and because the optimal strategy is always pure and memoryless.

Linear program

We can also use the following linear program proposed in [dA97] to calculate an optimal strategy. We are presenting it here for comparison to the other solutions later in this section.

Definition 2.14 (Linear program for MDP) Let $\mathcal{M}$ be an unichain MDP such that every Markov chain induced by any strategy contains at least one non-zero reward. Then we define the following fractional linear program for it.

$$\text{Minimize } \lambda$$

subject to

$$h_m \leq c_m - \lambda r_m + \sum_{m' \in M} p(m, a, m') h_{m'} \qquad \forall m \in M, a \in \overline{A}(m)$$

To calculate a strategy from a solution h_m to the LP we choose the actions for the states such that the constraints are fulfilled when we interpret them as equations.

Policy iteration

We will now design a policy iteration algorithm for $\mathcal{R}$ that is based on the policy iteration algorithm for $\mathcal{P}$, which we show in Algorithm 2.3. Recall that we use functions with finite domain and vectors interchangeably.

The goal of this algorithm is to find a strategy with minimal expected mean payoff. The algorithm consists of one loop in which we produce a sequence of strategies until no further improvement is possible (Line 17), i.e., until there is no strategy with smaller expected payoff. At the beginning of

Input: MDP $\mathcal{M} = (M, m_0, A, \overline{A}, p)$, mean payoff function $r : M \times A \to \mathbb{R}$

Output: Value $\mathbb{E}_{\mathcal{M}_d}[\mathcal{P}]$ and optimal strategy d

1. $n \leftarrow 0$, $d_0 \leftarrow$ arbitrary strategy;
2. **repeat**
3. Obtain vectors g_n, b_n that satisfy

$$\begin{aligned} (P_{d_n} - I)g_n &= 0 \\ r_{d_n} - g_n + (P_{d_n} - I)h_n &= 0 \\ P^*_{d_n} h_n &= 0 \end{aligned}$$

4. $\overline{A}'(m) \leftarrow \arg\min_{(m,a)\in\overline{A}} \sum_{m'\in M} p(m, a, m')g_n(m')$;
5. Choose d_{n+1} such that $d_{n+1}(m) \in \overline{A}'(m)$;
6. **foreach** $m \in M'$ **do if** $d_n(m) \in \overline{A}'(m)$ **then** $d_{n+1}(m) \leftarrow d_n(m)$;
7. ;
8. ;
9. **if** $d_n = d_{n+1}$ **then**
10. $\overline{A}'(m) \leftarrow \arg\min_{(m,a)\in\overline{A}} r(m) + \sum_{m'\in M} p(m, a, m')h_n(m')$;
11. Choose d_{n+1} such that $d_{n+1}(m) \in \overline{A}'(m)$;
12. **foreach** $m \in M'$ **do if** $d_n(m) \in \overline{A}'(m)$ **then** $d_{n+1}(m) \leftarrow d_n(m)$;
13. ;
14. ;
15. **end**
16. $n \leftarrow n + 1$;
17. **until** $d_{n-1} = d_n$;

Algorithm 2.3: Finding optimal strategies for MDPs with mean payoff [Put94]

the loop we solve a linear equation system (Line 3). In this system, we denote by P_d the probability matrix we obtain from combining $\mathcal{M}$ with d, i.e., $P_d(m, m') = p(m, d(m), m')$. Analogously, r_{d_n} denotes the reward vector induced by strategy d_n, i.e., $r_{d_n}(m_i) = r(m_i, d(m_i))$. Finally, by I we denote an identity matrix of appropriate size. The resulting vectors are gain g and bias h. Gain $g(m)$ is equal to the expected payoff of a run starting in m. The bias can be interpreted as the expected total difference between a reward obtained in a state and the expected reward of that state [Put94]. Its detailed semantics is not of material importance to this chapter. In Line 4 we collect all possible actions for each state that minimize the local expected gain. In line Line 5 we choose one strategy from the possible actions. To guarantee termination

of this algorithm we fix the chosen strategy (Line 8) such that we choose the same action as the old strategy whenever possible. If it was not possible to find an improved strategy in this way, then we perform the steps from line Line 4 to Line 8 with a different local target function (Line 9 to Line 14), based on reward and bias.

Theorem 2.4 (Algorithm 2.3 terminates and is correct) *Algorithm 2.3 always terminates and returns an optimal strategy.*

PROOF In each of the iterations of this algorithm we have one of the following cases [Put94]

- $d_n = d_{n+1}$: In this case there is no better strategy.
- $d_n \neq d_{n+1}$: We know that either $g_n < g_{n+1}$ or $g_n = g_{n+1}$ and $h_n < h_{n+1}$ (i.e., we have a lexicographic ordering).

In the first case we know that we found the best possible strategy. This implies the correctness. From the second case it follows that no two strategies can show up twice except for the first case. Since there are only finitely many strategies we know that the algorithm therefore terminates. ■

We are now going to reduce the search for an optimal ratio strategy for an MDP $\mathcal{M}$ with reward r and cost c to the search for an optimal mean cost strategy. According to Theorem 2.2, the ratio value of a unichain strategy d is $\lambda_d = \pi_d c_d / \pi_d r_d$, if we interpret c_d, r_d and π_d as vectors. Equivalently, $(c_d - \lambda_d r_d)\pi_d = 0$. If we now construct a mean payoff reward function $r' = c - \lambda r$, then d has therefore an expected mean payoff of zero. We call r' the *reward induced by* λ.

Definition 2.15 (Reward induced by $\boldsymbol{\lambda}$) Let c and r be cost and reward functions and let $\lambda \in \mathbb{R}$ be a constant. Then we define the *reward induced by* λ as $r'(m,a) = c(m,a) - \lambda r(m,a)$.

The correlation between functions r, c and r' go even further, as the following lemma shows.

Lemma 2.9 (Relation of ratio and mean payoff) *Let $\mathcal{M}$ be an MDP, let r and c be payoff functions, let d and d' two unichain strategies with expected ratio λ and λ', respectively, and let r' be the reward function induced by λ. Then the following three claims hold.*

1. *$\lambda' = \lambda$ if and only if the expected mean cost of d' in $\mathcal{M}$ with r' is zero i.e., $\mathbb{E}_{\mathcal{M}_{d'}}[\mathcal{P}_{r'}] = 0$.*

2. *$\lambda' < \lambda$ if and only if the value of d' in $\mathcal{M}$ with r' is smaller than zero, i.e., $\mathbb{E}_{\mathcal{M}_{d'}}[\mathcal{P}_{r'}] < 0 \iff \mathbb{E}_{\mathcal{M}_{d'}}[\mathcal{R}_{\frac{c}{r}}] < \mathbb{E}_{\mathcal{M}_{d}}[\mathcal{R}_{\frac{c}{r}}]$.*

3. *If λ is not optimal, then there exists a strategy with value smaller than zero for $\mathcal{M}$ and r'.*

Proof For *1.*, $\mathbb{E}_{\mathcal{M}_{d'}}[\mathcal{P}_{r'}] = 0$ if and only if $r'_{d'}\pi_{d'} = 0$ according to Theorem 2.2. By definition of r', this is equivalent to $(c_{d'} - \lambda r_{d'})\pi_{d'} = 0$. By vector arithmetic, this is equivalent to $c_{d'}\pi_{d'}/r_{d'}\pi_{d'} = \lambda$. According to Theorem 2.2, $c_{d'}\pi_{d'}/r_{d'}\pi_{d'} = \lambda'$. So we obtain $\lambda = \lambda'$.

For *2.*, assume that d' with r' has a value smaller than zero, i.e., $0 > \pi_{d'}r'_{d'} = \pi(c_{d'} - \lambda r_{d'})$ by the first claim, where $\pi_{d'}$ is the steady state distribution of d' in $\mathcal{M}$. Equivalently, $0 > \pi_{d'}c_{d'} - \pi_{d'}\lambda r_{d'}$ and $\lambda > \pi_{d'}c_{d'}/\pi_{d'}r_{d'} = \lambda'$, where the last equality follows from Theorem 2.3. Since all transformations are equivalent, the proof of this claim is finished.

For *3.*, assume that d^* is optimal in $\mathcal{M}$ with c and r and that its value is λ^*. Also assume that d, i.e., that $\lambda > \lambda^*$ is not optimal. We will now show that d^* has a value smaller than zero in the combination of $\mathcal{M}$ and r', i.e., prove the claim.

Let $v = \pi_{d^*}r'_{d^*} = \pi_{d^*}(c_{d^*} - \lambda r_{d^*})$ be the expected mean payoff value of d^* for r' and let $v^* = \pi_{d^*}(c_{d^*} - \lambda^* r_{d^*})$ be analogous for λ^*. From $\lambda^* < \lambda$ it follows that $v^* > v$. Since v^* is the value of d^* in the combination of $\mathcal{M}$ and the reward induced by λ^*, we know that v^* is zero from the first claim. From $0 = v^* > v$ we have that v is smaller than zero. But v is the value of d^* in the combination of $\mathcal{M}$ and r' by definition, i.e., d^* is a better strategy in the induced MDP. ■

Lemma 2.9 allows us to find an optimal strategy for $\mathcal{R}$ by starting with some strategy d with value $\lambda < \infty$. We can then look for a better strategy with Algorithm 2.3 in the combination of $\mathcal{M}$ and the function induced by λ. If we cannot find such a strategy, then d is optimal, according to the third claim of the last lemma. If we find such a strategy, then it has a ratio lower than λ according to the second claim of the last lemma. This leads us to Algorithm 2.4.

Input: End-component $\mathcal{M}$, unichain strategy d_0 (with $0 < \lambda_0 < \infty$)
Output: Optimal unichain strategy d_n

```
1 n ← 0;
2 repeat
3 |  λ_n ← E_{M_{d_n}}[R];
4 |  d_{n+1} ← improved unichain strategy for M_{λ_n} ;
5 |  n ← n + 1;
6 until d_{n-1} = d_n;
```

Algorithm 2.4: Policy iteration for $\mathcal{R}$

This algorithm is correct and terminates since the expected values we produce are always decreasing. From Lemma 2.9 follows that the algorithm will always find a correct strategy. Note that it is undefined how far we improve the strategy in Line 4. We can take the first strategy having an expected payoff smaller than zero or we can find an optimal strategy. As we will see in Section 6 there seems to be little difference between the two approaches. However, the following example shows that choosing the best strategy in the induced MDP is not always beneficial.

Example 2.9 Consider Figure 2.5b on Page 53. If we choose for state s_0 the action with cost 1 and reward 1, then we obtain 1 as expected ratio payoff of this MDP. In the MDP induced by 1 we have -6.3 as expected mean payoff for choosing the action with cost 5 and reward 1, according to Example 2.7. Analogously, we have -27 for choosing the other action. Therefore, if choose the optimal strategy in the induced MDP we will choose the latter action. But, as seen in Example 2.7, choosing the former is optimal. ■

Theorem 2.5 (Algorithm 2.4 terminates and is correct) *Algorithm 2.4 terminates and is correct.*

Proof Two strategies with different efficiencies cannot be the same. The ratios in Algorithm 2.4 are monotonically improving. There are only finitely many strategies. So termination follows. Correctness follows from Lemma 2.9. ■

Composing MDPs

Once we have calculated optimal strategies for end-components, we calculate a strategy that selects end-components and decisions to reach them optimally. To

that end, we employ algorithms calculating optimal strategies of mean-payoff MDPs. We presented one such algorithm in Algorithm 2.3.

We construct a new MDP in which each end-component is represented by one state, and each state not in an end-component is represented by itself. If it was possible to move from one state or end-component to another with a given action, then it will be possible to move from one representing state to the other in the new MDP. We will assign rewards such that staying in a state representing an end-component is rewarded by the expected payoff of that component. Moving from one component to another has no cost. An optimal strategy for this MDP defines an optimal strategy for states not in an end-component as well as movement between end-components.

Lemma 2.10 *Given an MDP $\mathcal{M}$ and an optimal pure strategy d_i for every maximal end-component $C_i, 1 \leq i \leq n$ in $\mathcal{M}$, we can compute the optimal value and construct an optimal strategy for $\mathcal{M}$.*

PROOF Let λ_i be the value obtained with d_i in the MDP induced by C_i. Without loss of generality, we assume that every action is enabled in exactly one state.

Let $\overline{\mathcal{M}} = (\overline{M}, \overline{m_0}, A, \overline{\overline{A}}, \overline{p})$ be the MDP of $\mathcal{M}$ defined by

- $\overline{M} = \{C_i \mid \forall 1 \leq i \leq n\} \cup \{m \in M \mid m \notin \bigcup C_i\}$
- $\overline{m_0} = \begin{cases} C_i & m_0 \in C_i \\ m_0 & \text{otherwise} \end{cases}$
- $\overline{A} = A$
- $\overline{\overline{A}} = \{(m, a) \subseteq \overline{A} \mid m \notin \bigcup C_i\} \cup \{(C_i, a) \mid \exists m \in C_i \wedge (m, a) \in \overline{A}, 1 \leq i \leq n\}$
- $\forall m, m' \in M \cap M' \forall a \in A : \overline{p}(m, a, m') = p(m, a, m')$, i.e., movement between states that do not lie in any end-component is like for $\mathcal{M}$
- $\forall 1 \leq i \leq n \forall m \in M \cap M' \forall a \in A : \overline{p}(m, C_i) = \sum_{m' \in C_i} p(m, a, m')$, i.e., the probability of moving from a state in no end-component to end-component C_i in $\mathcal{M}'$ is equal to the probability of moving from m to any of the states of C_i in $\mathcal{M}$

- $\forall 1 \leq i \leq n \forall m \in M \cap M' : \forall a \in A : \overline{p}(C_i, a, m) = \max_{m' \in C_i} p(m', a, m)$, i.e., the probability of moving from end-component C_i to a state m in no end-component with action a is equal to the probability of moving from the single state in which a is activated to m; it is zero if no such state exists

- $\forall 1 \leq i, j \leq n \forall a \in A : \overline{p}(C_i, a, C_j) = \max_{m \in C_i} \sum_{m' \in C_j} p(m, a, m')$, i.e., the probability of moving from end-component C_i to end-component C_j with action a is equal to the probability of moving from the single state in which a is activated to any of the states in C_j; it is zero if no such state exists

We modify $\overline{\mathcal{M}}$ to obtain an MDP $\mathcal{M}'$ by removing all actions for which there is a state $m \in \overline{M}$ such that $\overline{p}(m, a, m) = 1$. Furthermore, for all states m that are an end-component in $\mathcal{M}$ with value $\lambda_i < \infty$, we add a new action a_i with $p(m, a_i, m) = 1$ and costs λ_i; all other actions have cost 0. We now recursively remove states without enabled actions and actions leading to removed states. If the initial state m_0 is removed, the MDP has value infinity, because we cannot avoid reaching and staying in an end-component with value infinity.

Otherwise, let d' be an optimal strategy for $\mathcal{M}'$. We define d by $d(m) = d'(\overline{m})$ for all states $m \notin \bigcup C_i$. For $m \in C_i$, if $d'(\overline{m}) = a_i$, we set $d(m) = d_i(m)$. Otherwise, let a be the actions chosen in state $\overline{m}$, and let m' be the state in which a is enabled. Then, we set $d(m') = a$ and for all other states in C_i we choose d such that we reach m' with probability 1. We can choose the strategy arbitrarily in states that were removed from $\mathcal{M}'$, because these states will never be reached by construction of d.

Because of the way we constructed $\mathcal{M}'$, d and d' have the same value, and d is optimal because d' is optimal (Theorem 2.3). ■

Evaluation

The goal of this first evaluation is to find out which of the given implementations we should follow to try to scale to large systems. We therefore apply all three implementations to a series of production line configurations of increasing size. We also report on the synthesized strategies.

Synthesis results

We synthesized optimal controllers for systems with two to five production lines. They behave as follows: For a system with two production lines, the controller plays it safe. It turns one production line on in fast mode and leaves the other one turned off. If the production line breaks, then the other production line is turned on in slow mode and the first production line is repaired immediately. For three production lines, all three production lines are turned on in fast mode. As soon as one production line breaks, only one production line is turned on in fast mode, the other one is turned off. Using this strategy, the controller avoids the penalty of having no working production line with high probability. If two production lines are broken, then the last one is turned on in fast mode and the other two production lines are been repaired. In the case of four production lines, all production lines are turned on in fast mode if they are all working. If one production line breaks, then two production lines are turned on and the third working production line is turned off. The controller has one production line in reserve for the case that both used production lines break. If two production lines are broken, then only one production line is turned on, and the other one is kept in reserve. Only if three production lines are broken, the controller starts repairing the production lines. Using this strategy, the controller maximizes the discount for repairing multiple production lines simultaneously.

We also evaluated the ACTS described in Section 2.1. The model has two parts: a motor and a driver profile. The state of the motor consists of revolutions per minute (RPM) and a gear. The RPM range from 1000 to 6000, modeled as a number in the interval (10, 60), and we have three gears. The driver is meant to be a city driver, i.e., she changes between acceleration and deceleration frequently. The fuel consumption is calculated as polynomial function of degree three with the saddle point at 1800 rpm. The final model has 384 states and it takes less than a second to build the MDP. Finding the optimal strategy takes less than a second. The resulting expected fuel consumption is 0.15 l/km. The optimal strategy is as expected: the shifts occur as early as possible.

Experiments

We have implemented the algorithms presented here. Our first implementation is written in `Haskell`[4] and consists of 1500 lines of code. We use the Haskell package `hmatrix`[5] to solve the linear equation system and `glpk-hs`[6] to solve the linear programming problems. In order to make our work publicly available in a widely used tool and to have access to more case studies, we have implemented the best-performing algorithm within the explicit-state version of PRISM. It is an implementation of the strategy improvement algorithm and uses numeric approximations instead of solving the linear equation systems.

First, we will give mean running times of our `Haskell` implementation on the production line example, where we scale the number of production lines. The tests were done on a Quad-Xeon with 2.67GHz and 3GB of heap space. Table 2.1 shows our results. Column n denotes the number of production lines

n	$\|M\|$	$\|A\|$	LP		FLP		Opt		Imp.	
2	9	144	0.002	13	0.015	14	0.003	13	0.003	14
3	27	1728	0.043	14	0.642	20	0.027	13	0.009	14
4	81	20736	1.836	41	14.73	332	0.122	21	0.122	24
5	243	248832	67.77	505	n/a	n/a	1.647	162	1.377	166

Table 2.1: Experimental results table

we use, $|M|$ and $|A|$ denote the number of states and actions the final MDP has. Note that $|M| = 3^n$ and $|A| = 12^n$. The next columns contain the time (in seconds) and the amount of memory (in MB) the different algorithms used. LP denotes the linear program, FLP the fractional linear program. We have two versions of the policy iteration algorithm: one in which we improve the induced MDP to optimality (Column Opt.), and one where we only look for any improved strategy (Column Imp.). The policy iteration algorithms perform best, and Imp. is slightly faster than Opt but uses a little more memory. For $n = 5$, the results start to differ drastically. FLP ran out of memory, LP needed about a minute to solve the problem, and both Imp. and Opt. stay below two seconds.

Using our second implementation, we also tried our algorithm on some of the case studies presented on the PRISM website. For example, we used the

[4] `http://www.haskell.org`
[5] `http://code.haskell.org/hmatrix/`
[6] http://hackage.haskell.org/package/glpk-hs

IPv4 zeroconf protocol model. We asked for the minimal expected number of occurrences of action `send` divided by occurrences of action `time`. If we choose $K = 5$ and `reset = true`, then the resulting model has 1097 states and finding the optimal strategy takes 5 seconds. For $K = 2$ and `reset = false`, the model has about 90000 states and finding the best strategy takes 4 minutes on a 2.4GHz Core2Duo P8600 laptop.

2.5 Symbolic implementation

In this section, we will discuss a symbolic variant of the policy iteration algorithm, i.e., the structure of Algorithm 2.1 with Algorithm 2.4 implementing `solveEC`. Symbolic encoding via *binary decision diagrams* (*BDDs*) has enabled model checking and qualitative synthesis to address the state explosion problem in many cases [BCM$^+$92], i.e., the problem that the state space we need to analyze grows exponentially with the number of the components of a model.

Recently, Wimmer et al. developed a semi-symbolic (or, in their terms, *symblicit*) variant of Algorithm 2.3 [WBB$^+$10] with promising results. In this section, we develop an analogous algorithm for the case of Ratio-MDPs.

We call the algorithm semi-symbolic because it uses symbolically as well as explicitly encoded MDPs. BDDs are good for encoding large structures but they are not suitable when it comes to solving linear equation systems (see for example [HMPS96, KNP02]). Therefore we (like [WBB$^+$10]) encode MDPs and strategies symbolically but convert the induced MC into an explicit linear equation system (after having reduced the state space via bisimulation).

Symbolic encoding

In this subsection we will first describe BDDs and their extension *multiterminal BDDs* (*MTBDDs*). We will then encode MDPs, MCs, strategies and reward functions as BDDs or MTBDDs as necessary.

Binary decision diagrams [Bry86] encode Boolean functions as directed acyclic graphs as follows.

Definition 2.16 (Binary decision diagram [Bry86]) We use BDDs to encode Boolean functions $2^{\mathrm{V}} \to \mathbb{B}$, where V is a finite set of variables. For $\mathrm{V} = \{v, w\}$ and $f : 2^{\mathrm{V}} \to \mathbb{B}$, we write $f(\{v\})$ to denote the function value of

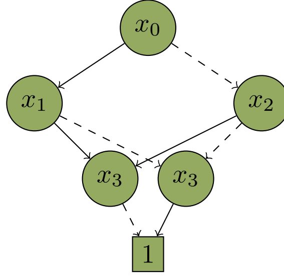


Figure 2.7: A BDD encoding of $x_0 \cdot (x_1 \leftrightarrow \neg x_3) \vee \neg x_0 \cdot (x_2 \leftrightarrow \neg x_3)$

the variable assignment $v = 1 \wedge w = 0$. Accordingly, $f(\emptyset)$ denotes the value of f for the assignment $v = 0 \wedge w = 0$.

We define a BDD by $(\mathrm{B}, \mathrm{b}_0, \overline{\mathrm{B}}, \overline{\mathrm{V}})$, where B is a finite set of nodes, $\mathrm{b}_0 \in \mathrm{B}$ is the root node, $\overline{\mathrm{B}} : \mathrm{B} \to (\mathrm{B} \cup \mathbb{B})^2$ encodes the edge relation, and $\overline{\mathrm{V}} : \mathrm{B} \to \mathrm{V}$ is a function assigning a variable to each node. That is, if $\overline{\mathrm{B}}(\mathrm{b}) = (\mathrm{b}', \mathrm{b}'')$, then there is an edge from b to b' and from b to b''. We demand the existence of a asymmetric ordering $<$ on V such that if there is a path from $\mathrm{b} \in \mathrm{B}$ to $\mathrm{b}' \in \mathrm{B}$, then $\overline{\mathrm{V}}(\mathrm{b}) < \overline{\mathrm{V}}(\mathrm{b}')$. We further demand that the graph be reduced, i.e., there may be no isomorphic sub-BDDs, i.e., no two sub-BDDs encoding the same Boolean function.

Example 2.10 (Binary decision diagram) In Figure 2.7 we show the BDD encoding the Boolean function, $f : 2^{\{x_0, x_1, x_2, x_3\}} \to \mathbb{B}, f = x_0 \cdot (x_1 \leftrightarrow \neg x_3) \vee \neg x_0 \cdot (x_2 \leftrightarrow \neg x_3)$. For example, $f(\{x_0, x_1\}) = f(\{x_0, x_1, x_2\}) = 1$ and $f(\{x_0, x_1, x_3\}) = f(\{x_0, x_1, x_2, x_3\}) = 0$. Each circle describes a node, each edge a connection between nodes. Solid edges denote positive assignments to variables, dashed edges negative assignments. For simplicity, we omit any edge leading to the leaf denoting zero.

For example, in Figure 2.7, assignment $\{x_0, x_1, x_3\}$ leads us to leaf 0 by first following the solid edge from x_0 to x_1, then the solid edge to x_3, and finally the solid edge from x_3, which is not depicted here, and therefore leads to leaf 0. On the other hand, assignment $\{x_0, x_1\}$ leads to leaf 1 by following the same path to x_3 and then taking the dashed edge.

BDDs support several logical operations very efficiently (linear in the number of nodes).

Definition 2.17 (Operations on BDDs) Let $\mathcal{B}$ and $\mathcal{B}'$ be BDDs, and let V be the set of variables. The following operations are efficiently supported on BDDs, corresponding to operations on Boolean functions.

- Negation $\neg\,\mathcal{B}$
- Conjunction $\mathcal{B} \wedge \mathcal{B}'$
- Disjunction $\mathcal{B} \vee \mathcal{B}'$
- Existential Quantification $\exists\, \mathrm{V}' \subseteq \mathrm{V} : \mathcal{B}$
- Universal Quantification $\forall\, \mathrm{V}' \subseteq \mathrm{V} : \mathcal{B}$

All operations have time complexity proportional to the size of the BDDs [Bry86].

For more information about how to implement these operations, see [Bry86].

In [FMY97] the authors introduced an extension of BDDs that encodes functions mapping from a finite set to $\mathbb{R}$. It was originally developed to encode matrices and operations on matrices efficiently. It has since been used successfully to encode MCs, MDPs, etc. [BCM+92].

Definition 2.18 (Multiterminal binary decision diagram) We use MTBDDs to encode functions $2^{\mathrm{V}} \to \mathbb{R}$, where V is a finite set of variables.

We define an MTBDD by $(\mathrm{B}, \mathrm{b}_0, \overline{\mathrm{B}}, \overline{\mathrm{V}})$, where B, b_0 and $\overline{\mathrm{V}}$ are encoded as for BDDs. The edge relation now supports $\mathbb{R}$ instead of $\mathbb{B}$ as leaves, i.e., $\overline{\mathrm{B}} : \mathrm{B} \to (\mathrm{B} \cup \mathbb{R})^2$. We still demand the existence of an ordering and the lack of isomorphic subgraphs as in Definition 2.16.

MTBDDs also support several operations very efficiently.

Definition 2.19 (Operations on MTBDDs) Let $\mathcal{B} = (\mathrm{B}, \mathrm{b}_0, \overline{\mathrm{B}}, \overline{\mathrm{V}})$ and $\mathcal{B}'$ be two MTBDDs. The following operations are efficiently supported on MTBDDs, corresponding to operations on functions.

- Negation $-\,\mathcal{B}$
- Addition $\mathcal{B} + \mathcal{B}'$
- Multiplication $\mathcal{B} \times \mathcal{B}'$

- Division $\mathcal{B} \,/\, \mathcal{B}'$
- Minimization $\min_{\mathrm{V}' \subseteq \mathrm{V}} : \mathcal{B}$
- Maximization $\max_{\mathrm{V}' \subseteq \mathrm{V}} : \mathcal{B}$
- Summation $\sum_{\mathrm{V}' \subseteq \mathrm{V}} : \mathcal{B}$
- Comparison with constant $c \in \mathbb{R}$: $\mathcal{B} < c$

All operations have time complexity polynomial in the size of the MTBDDs [FMY97].

These structures allow us to encode MDPs in the following way.

Definition 2.20 (Symbolic encoding of MDPs, MCs and strategies) Let $\mathcal{M} = (M, m_0, A, \overline{A}, p)$ be an MDP. Then we encode the MDP symbolically as follows.

- We assign a symbolic encoding to M using $\lceil \log_2(|M|) \rceil$ variables V_M, described as an injective function $\mathrm{enc}_M : M \to 2^{\mathrm{V}_M}$.
- We assign a second symbolic encoding to M using $\lceil \log_2(|M|) \rceil$ variables V'_M, described as an injective function $\mathrm{enc}'_M : M \to 2^{\mathrm{V}'_M}$.
- We assign a symbolic encoding to A using $\lceil \log_2(|A|) \rceil$ variables V_A described as an injective function $\mathrm{enc}_A : A \to 2^{\mathrm{V}_A}$.
- We encode the set of states as a function f_M such that $f_M(\mathrm{enc}_M(m)) = 1$ for all states $m \in M$, and 0 for everything else, i.e., if a variable assignment encodes a state, then f_M evaluates to 1, otherwise to 0. This is necessary because the set of assignments to V_M may be greater than the set of variables.
- We encode the action activation relation A as a function $f_{\overline{A}} : 2^{\mathrm{V}_M \cup \mathrm{V}_A} \to \mathbb{B}$ such that $f_{\overline{A}}(\mathrm{enc}_M(m) \cup \mathrm{enc}_A(a)) = 1$ if and only if $(m, a) \in \overline{A}$.
- We encode the transition function as a function $f_p : 2^{\mathrm{V}_M \cup \mathrm{V}_A \cup \mathrm{V}_{M'}} \to \mathbb{R}$ such that $f_p(\mathrm{enc}_M(m) \cup \mathrm{enc}_A(a) \cup \mathrm{enc}'_M(m')) = p(m, a, m')$ for all states $m, m' \in M$ and actions $a \in A$ and 0 for every other assignment.

- We encode a strategy $d : M \to 2^A$ as the corresponding Boolean function $f_d : 2^{\mathrm{V}_M \cup \mathrm{V}_A} \to \mathbb{B}$, i.e., $f_d(\mathrm{enc}_M(m) \cup \mathrm{enc}_A(a)) = 1$ if and only if $d(m) = a$.
- We encode costs and reward functions analogously as functions $f_c, f_r : 2^{\mathrm{V}_M \cup \mathrm{V}_A} \to \mathbb{R}$.

We encode MCs analogously by leaving out the actions

Our goal is to have a symbolic policy iteration algorithm. This algorithm uses linear equation systems to evaluate strategies (Algorithm 2.3). Because solving linear equation systems using MTBDDs is slow in general (see for example [HMPS96, KNP02]), we will show (based on [WBB+10]) how to construct a small equivalent linear equation system in the next subsection.

Bisimulation and symbolic bisimulation

We aim to solve the equation system in Algorithm 2.3 explicitly. The size of the equation system we solve is proportional to the number of states in the induced MC. Therefore, we aim to reduce the number of states in the Markov chain we have to analyze. To reduce the size of Markov chains, we will use bisimulation. Bisimulation aims to identify a *partition* of the set of states of a Markov chain that has certain properties, which we define later. We define a *signature* on partitions of states of Markov chains.

Definition 2.21 (Signature) Let $\mathcal{C} = (M, m_0, p_\mathcal{C})$ be a Markov chain and $\mathcal{B}$ a partition of M. For any state $m \in M$ and any block $\mathrm{B} \in \mathcal{B}$, we define $p_\mathcal{C}(m, \mathrm{B}) = \sum_{m' \in \mathrm{B}} p_\mathcal{C}(m, m')$ as the probability of going from m to any state in B. The signature of a partition $\mathcal{B}$ of M is defined as $\mathrm{sig}(\mathcal{B}) := m \mapsto \{(\mathrm{B}, p_\mathcal{C}(m, \mathrm{B})) \mid \mathrm{B} \in \mathcal{B}\}$, i.e., the signature of a bisimulation is a function mapping states of $\mathcal{C}$ to the probability of moving to blocks.

A *bisimulation* of a Markov chain is a partition of the set of states, such that any two states in the same block have 1) the same label according to a given labeling function and 2) have equal probability of transitioning to blocks.

Definition 2.22 (Bisimulation) For a MC $\mathcal{C} = (M, m_0, p_\mathcal{C})$ and a function $l : M \to L$ for some set L, a bisimulation is a partition of the states $\mathcal{B} \subseteq 2^M$ such that for all blocks $\mathrm{B} \in \mathcal{B}$ and for all states $m, m' \in \mathrm{B}$ therein we have that

$l(m) = l(m')$ and $p_\mathcal{C}(m, \mathrm{B}') = p_\mathcal{C}(m', \mathrm{B}')$ for all blocks $\mathrm{B}' \in \mathcal{B}$. A bisimulation is called *finer* than another bisimulation if it is finer when interpreted as a partition. A bisimulation $\mathcal{B}$ is called *maximal* if each other bisimulation is finer than $\mathcal{B}$.

A bisimulation defines a smaller, equivalent Markov chain in the following sense.

Definition 2.23 (Quotient MC) Let $\mathcal{C} = (M, m_0, p_\mathcal{C})$ be a Markov chain and let $\mathcal{B}$ be a bisimulation. The quotient MC is defined as $\mathcal{C}_\mathcal{B} = (\mathcal{B}, \mathrm{B}_0, p^\mathcal{B}_\mathcal{C})$, where $\mathrm{B}_0 \in \mathcal{B}$ is the block such that $m_0 \in \mathrm{B}_0$. The probabilistic transition function between blocks is defined as $p^\mathcal{B}_\mathcal{C}(\mathrm{B}, \mathrm{B}') = p_\mathcal{C}(m, \mathrm{B}') = \sum_{m' \in \mathrm{B}'} p_\mathcal{C}(m, m')$ for some[7] state $m \in \mathrm{B}$.

Obviously, $\mathcal{C}_\mathcal{B}$ is a well-defined Markov chain. It has the following property of interest to us.

Lemma 2.11 *Let $\mathcal{C} = (M, m_0, p_\mathcal{C})$ be a Markov chain, $l : M \to \mathbb{R}$ be a labeling function, $\mathcal{B}$ be a bisimulation of $\mathcal{C}$ and $\mathcal{C}_\mathcal{B}$ be the quotient MC defined by $\mathcal{C}$ and $\mathcal{B}$. Let $l_\mathcal{B}(\mathrm{B}) = l(m)$ for some $m \in \mathrm{B}$. Then $\mathbb{E}_\mathcal{C}[\mathcal{R}_l] = \mathbb{E}_{\mathcal{C}_\mathcal{B}}[\mathcal{R}_{l_\mathcal{B}}]$, i.e., instead of calculating the expected value of $\mathcal{R}$ for $\mathcal{C}$ we can calculate the expected value for its quotient MC $\mathcal{C}_\mathcal{B}$.*

In [WDH08], Wimmer et.al. propose a symbolic bisimulation algorithm, i.e., an algorithm calculating the maximal bisimulation of a Markov chain. We repeat this algorithm here.

The algorithm is based on the following insight. A partition $\mathcal{B}$ is a bisimulation if and only if for each pair of states m, m' in the same block we have $\mathrm{sig}(\mathcal{B})(m) = \mathrm{sig}(\mathcal{B})(m')$. Conversely, if $\mathcal{B}$ is not a bisimulation we can bring it "closer" to being one by refining $\mathcal{B}$ according to sig.

Definition 2.24 (Signature refinement algorithm) Let $\mathcal{C} = (M, m_0, p_\mathcal{C})$ be a Markov chain and let $\mathcal{B}_0$ of M be an initial partition of M according to a labeling function. Let $\mathcal{B}$ be a partition of M that is finer than $\mathcal{B}_0$. We refine $\mathcal{B}$ by putting all those states in one block that have the same signature and were equivalent in the initial partition, i.e.,

$$\mathrm{sigref}_{\mathcal{B}_0}(\mathcal{B}) = \{\{m' \in M \mid \mathrm{sig}(m, \mathcal{B}) = \mathrm{sig}(m', \mathcal{B}) \wedge m \equiv_{\mathcal{B}_0} m'\} \mid m \in M\}$$

[7] Note that it does not matter which state in B we choose because $p_\mathcal{C}(m, \mathrm{B}') = p_\mathcal{C}(m', \mathrm{B}')$ for any $m, m' \in \mathrm{B}$.

The signature refinement algorithm then consists of a fixpoint application of $\text{sigref}_{\mathcal{B}_0}$ to the initial partition.

Lemma 2.12 (Signature refinement is correct [WDH08]) *The signature refinement algorithm yields the maximal bisimulation of a Markov chain.*

To perform these operations symbolically, we will first need to define a symbolic encoding of partitions and signatures. Then we can define symbolic versions of sig and $\text{sigref}_{\mathcal{B}_0}$.

Definition 2.25 (Symbolic signature refinement [WDH08]) Let $\mathcal{M}$ be a Markov chain and let $\mathcal{B}$ be a partition of the state space of $\mathcal{M}$. Let $\mathrm{V}_{\mathcal{B}}$ be a set of $\lceil \log_2(|\,\mathcal{B}\,|) \rceil$ fresh variables. We identify each block of $\mathcal{B}$ with a unique encoding $\text{enc}_{\mathcal{B}}(\mathrm{B})$, for $\mathrm{B} \in \mathcal{B}$. For example, we can enumerate the blocks and use a binary encoding of the number of the block.

We can encode the blocks via a function $f_{\mathcal{B}} : 2^{\mathrm{V}'_M \cup \mathrm{V}_{\mathcal{B}}} \to \mathbb{B}$ such that $f_{\mathcal{B}}(\text{enc}'_M(m) \cup \text{enc}_{\mathcal{B}}(\mathrm{B})) = 1$ if and only if $m \in \mathrm{B}$, for states $m \in M$ and blocks $\mathrm{B} \in \mathcal{B}$.

We encode the signature of a state as the MTBDD of a function $f_{\text{sig}(\mathcal{B})} : 2^{\mathrm{V}_M \cup \mathrm{V}_{\mathcal{B}}} \to \mathbb{R}$, $f_{\text{sig}(\mathcal{B})}(\text{enc}_M(m) \cup \text{enc}_{\mathcal{B}}(\mathrm{B})) = r \in \mathbb{R}$ if and only if $(r, \mathrm{B}) \in \text{sig}(\mathcal{B})(m)$. For all other assignments, $f_{\text{sig}(\mathcal{B})}$ is zero. Given the encoding of a partition, we can compute this function by $f_{\text{sig}(\mathcal{B})} = \sum_{\mathrm{V}_{M'}} f_p \times f_{\mathcal{B}}$.

To take an initial partition $\mathcal{B}_0$ into account, we introduce a fresh variable V_0 and define the final signature function $f'_{\text{sig}(\mathcal{B})} = f_{\text{sig}(\mathcal{B})} + \mathrm{V}_0 \times f_{\mathcal{B}_0}$, where $f_{\mathcal{B}_0} : 2^{\mathrm{V}_M \cup \mathrm{V}_{\mathcal{B}}} \to \{0, 1\}$ encodes the initial partition.

A partial application of $f'_{\text{sig}(\mathcal{B})}$ to the encoding of two states m and m' will be the same if and only if they have the same signature according to $\mathcal{B}$ and lie in same block in $\mathcal{B}_0$: $f'_{\text{sig}(\mathcal{B})}(\text{enc}_M(m)) = f'_{\text{sig}(\mathcal{B})}(\text{enc}_M(m'))$ if and only if 1) $f_{\text{sig}(\mathcal{B})}(\text{enc}_M(m))(\text{enc}_{\mathcal{B}}(\mathrm{B})) = f_{\text{sig}(\mathcal{B})}(\text{enc}_M(m'))(\text{enc}_{\mathcal{B}}(\mathrm{B}))$ for every block $\mathrm{B} \in \mathcal{B}$ and 2) $f_{\mathcal{B}_0}(\text{enc}_M(m)) = f_{\mathcal{B}_0}(\text{enc}_M(m'))$.

We now show that the encoding of $f_{\text{sig}(\mathcal{B})}$ is correct.

Lemma 2.13 (The signature encoding is correct) *We have to show that* $f_{\text{sig}(\mathcal{B})}(\text{enc}_M(m) \cup \text{enc}_{\mathcal{B}}(\mathrm{B})) = r \in \mathbb{R}$ *if and only if* $(r, \mathrm{B}) \in \text{sig}(m, \mathcal{B})$.

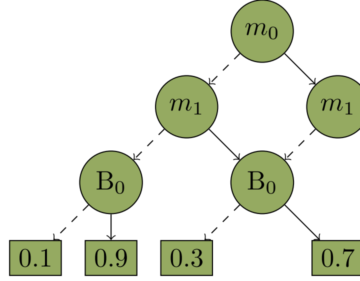


Figure 2.8: Symbolic signature

PROOF We defined $f_{\mathrm{sig}(\mathcal{B})} = \sum_{\mathrm{V}_{M'}} f_p \times f_{\mathcal{B}}$. If we now evaluate this function on state $m \in M$ and block $\mathrm{B} \in \mathcal{B}$, we receive

$$\begin{aligned} f_{\mathrm{sig}(\mathcal{B})}(\mathrm{enc}_M(m) \cup \mathrm{enc}_{\mathcal{B}}(\mathrm{B})) &= \sum_{\mathrm{V}_{M'}} f_p(\mathrm{enc}_M(m)) \times f_{\mathcal{B}}(\mathrm{enc}_{\mathcal{B}}(\mathrm{B})) \\ &= \sum_{m' \in \mathrm{B}} p(m, m') \\ &= p(m, \mathrm{B}) \\ &= r \iff (r, \mathrm{B}) \in \mathrm{sig}(\mathcal{B})(m) \end{aligned}$$

Example 2.11 (Signatures and blocks) To get a new symbolic partition from the symbolic signature, it is useful to recall that two states should end up in the same block if they have the same signature. Under the condition that variables encoding block numbers come after variables encoding states in the variable ordering (see Definition 2.18), we can get a simple algorithm achieving that. If we traverse the symbolic signature from the root until we find variables encoding the block number, all that remains is the encoding of that state's signature. For example, in Figure 2.8 the state encoded by $\emptyset$, i.e., where m_0 and m_1 are 0, leads to the block encoded by $\emptyset$ with probability 0.1, while it leads to the block encoded by $\{\mathrm{B}_0\}$ with probability 0.9. This also means that two states have the same signature if and only if the symbolic encoding of their signature is the same. Observe for example, in Figure 2.8, that the states encoded by $\{m_1\}$ and by $\{m_0\}$ lead to the block encoded by $\emptyset$ with probability 0.3, while they lead to the block encoded by $\{\mathrm{B}_0\}$ with probability 0.7. They have indeed the same signature, and should therefore end up in the same block in the refined partition. Therefore, all state encodings leading to the same node in $\mathrm{V}_{\mathcal{B}}$ have to have the same signature. We can therefore traverse the BDD backwards from subgraphs encoding signatures to get a symbolic encoding of

```
   Input: S, p, E
   Output: L
 1 L_0 ← [(S, E)];
 2 n ← 0;
 3 repeat
 4 |  L_{n+1} ← [];
 5 |  foreach (S', E') ← L_n do
 6 |  |  Let (S_1, E_1), ..., (S_l, E_l) be the SCCs of (S', E');
 7 |  |  E'_i ← E_i ∧ [∃a∀m' : p → S_i)] ;
 8 |  |  Add (S_i, E'_i) to L_{n+1};
 9 |  |  n ← n + 1;
10 |  end
11 until L_n = L_{n-1};
```

Algorithm 2.5: Symbolic end-component computation

all states in that block. This leads to an algorithm linear in the number of states of the BDD.

Symbolic policy iteration

After we have seen in Section 2.4 how to compute an optimal strategy for an explicitly encoded Ratio MDP, we will now develop a symbolic variant for the symbolic encoding we have just seen. To that end, we describe symbolic versions of all functions showing up in Algorithm 2.4.

Symbolic decomposition

We can adapt Algorithm 2.2 to symbolic structures. We will assume that there is a method of calculating SCCs symbolically.

First, we encode the MDP as a directed graph by replacing every leaf in p that has a value greater than 0 by 1. We further abstract existentially over all actions, i.e., $E = \exists_{\mathrm{V}_A} f_p$. We now have $E(\mathrm{enc}_M(m) \cup \mathrm{enc}_{M'}(m'))$ if and only if there is any action that makes it possible to move from m to m' in $\mathcal{M}$.

We present the algorithm in Algorithm 2.5. It works in exactly the same way as Algorithm 2.2. The only important line is Line 7, where we restrict the set of possible actions to those that stay inside an EC. Here we restrict the directed graph such that there is only an edge if there is any possible action that stays inside the end-component.

Symbolic `isZero` and `isInfty`

We use symbolic version of what we described in Lemma 2.8. For `isZero`, we restrict the set of states to those that have a cost of zero, i.e., $M \wedge \exists_{\mathrm{V}_{M'}} \exists_{\mathrm{V}_A} : f_c = 0$. If the resulting MDP has an end-component, then `isZero` will return true. For `isInfty` we check if $r = 0 \wedge [\neg M \vee (\exists_{\mathrm{V}_M} \exists_{\mathrm{V}_A} : c > 0)]$ is a tautology.

Symbolic policy iteration

```
   Input: MDP mdp consisting of a single end-component
   Output: strategy d and optimal ratio value λ of mdp
 1 d = initial(mdp) ;
 2 d_old = ⊥;
 3 while d ≠ d_old do
 4 |   λ = lambda(mdp, d);
 5 |   g, b = gainAndBias(mdp, d, λ);
 6 |   while g ≥ 0 and d_old ≠ d do
 7 |   |   d_old = d;
 8 |   |   d = next(mdp, d, λ, g, b);
 9 |   |   g, b = gainAndBias(mdp, d, λ);
10 |   end
11 end
12 return unichain(mdp, d), λ
```

Algorithm 2.6: Optimisation for a single end-component

In Algorithm 2.6 we present the algorithm that finds the optimal ratio value for an end-component. The algorithm first picks any strategy d that has a finite and strictly positive value (Line 1). We observed that the choice of this strategy has a strong influence on the performance of our algorithm.

Then, in Line 3 we enter a loop that produces in every iteration a new strategy that has the same or a better ratio value (λ) than the previous strategy. We exit the loop if the same strategy is produced twice, i.e., there is no strategy with a better ratio value for this MDP.

In the loop, we first compute the ratio value λ that can be obtained by a strategy generated from the strategy d (Line 4). This computation is done semi-symbolically. First, we compute the Markov chain C induced by strategy d. For C, we symbolically compute a bisimulation relation (Definition 2.24), which allows us to construct an equivalent smaller Markov chain C'. Then, we compute all recurrence classes (i.e., the strongly connected components) of C'. For each recurrence class, we build an explicit-state representation of the

sub-model and calculate the steady-state distribution, which in turn is used to calculate the ratio value of the recurrence class. We set λ to the value of the best recurrence class. This value is not necessarily the value of d but we can construct a strategy that has value λ. Furthermore, λ is at least as good as the actual value of d (see proof of Lemma 2.7).

In the rest of the algorithm, we perform computation on an MDP with average objective induced by the reward function $c - \lambda \times r$ (which we compute symbolically). For this induced MDP, we compute gain (g) and bias (b) (as in Algorithm 2.3). The computation of gain and bias is similar to the computation in Algorithm 2.3, i.e., we calculate gain and bias explicitly on an equivalent smaller Markov chain. We know that a state has a gain smaller than zero in the induced MDP if and only if its ratio value is smaller (i.e., better) than the ratio value from which the induced MDP was calculated (Lemma 2.9). Since the ratio value of strategy d is at most as good as λ, the gain of all states at this point is greater than or equal to zero.

We now enter the inner loop (Line 6), which runs while the strategy keeps changing and all entries of the gain vector are greater than or equal to zero. Equivalently, the loop runs until there is a recurrence class of the current strategy that has a value smaller than λ or until there is no better strategy anymore.

In the inner loop, we try to improve the strategy (Line 8) and calculate the new strategy's gain and bias (Line 9). Note that the choice of the next strategy and the way of computing the value λ differs from our description in Section 2.4. In the latter version, we demand a unichain strategy from the induced Mean MDP. Here, we demand just any kind of strategy. Forcing the algorithm to use a unichain strategy was a major bottleneck in our initial symbolic implementation, because it increased the number of blocks significantly by introducing irregularity into the (MT)BDDs.

Instead we work with arbitrary strategies now. To calculate the expected ratio of a strategy, we calculate the expected ratio of each recurrent class with Theorem 2.2 and take their minimum. The correctness of this approach follows trivially from the existence of a unichain Strategy with the same recurrence class as the optimal recurrence class, and therefore with the same value Lemma 2.7.

Symbolic composition

We can build a strategy for the whole MDP once we have built a strategy for the end-components. This could be done exactly as in the explicit variant, and by using a symbolic policy algorithm for mean-payoff MDPs, i.e., with a symbolic variant of Algorithm 2.3.

Instead, we reduce the problem of composing strategies to the problem of finding a *stochastic shortest path*.

Definition 2.26 (Stochastic shortest path problem) Let $\mathcal{M} = (M, m_0, A, \overline{A}, p)$ be an MDP, and $r : M \times A \to \mathbb{R}$ be a reward function. Then the *Total Reward* $\mathcal{F}_r : (M \times A)^\omega \to \mathbb{R}$ defined by r is defined as $\mathcal{F}_r(\rho) = \sum_{i=0}^{\infty} r(\rho_i)$. For a state $m \in M$, an optimal strategy for the stochastic shortest path problem is one for which the expected value is minimal, i.e., $\arg\min_{d \in D'} \mathbb{E}_{\mathcal{M}_d}[\mathcal{F}]$, where $D' \subseteq D(\mathcal{M})$ is the set of pure strategies reaching m almost surely.

We use the following definition to reduce strategy composition to a simple variant of the shortest stochastic path problem.

Definition 2.27 (Reduction to stochastic shortest path problem) Let $\mathcal{M} = (M, m_0, A, \overline{A}, p)$ be a MDP, $r : M \to \mathbb{R} \cup \{\bot\}$ be the optimal ratio calculated for each state so far, or $\bot$ if no reward has been calculated (because the state does not lie in any end component).

We construct a new MDP $\mathcal{M}' = (M', m'_0, A', \overline{A}', p')$ by adding a special state $m_\bot$ to the set of states and keeping the start state, i.e., $M' = M \cup \{\bot\}$, and $m'_0 = m_0$. Let $E \subseteq M$ be the set of states in any end-component of $\mathcal{M}$.

We extend the set of inputs by a fresh symbol $a_\bot$, i.e., $A' = A \cup \{a_\bot\}$, where $a_\bot$ is available in all states in any end-component, i.e., $\overline{A}' = \overline{A} \cup E \times \{\bot\}$. For any input $a \in A$ in $\mathcal{M}$, the transition function p' of $\mathcal{M}'$ is the same as in $\mathcal{M}$, i.e., $p'(m, a, m') = p(m, a, m')$ for all states $m, m' \in M$. For the new input $a_\bot$ we define $p'(m, a_\bot, \bot) = 1$ for all $m \in E \cup \{\bot\}$.

As reward function r' we assign $r'(m, \bot) = r(m)$ for every state $m \in E$, and 0 for all other states and actions.

We use Value Iteration to solve this problem.

Lemma 2.14 (Algorithm 2.7 terminates and is correct) *When given a symbolically encoded MDP with r the minimal reward optimal reward of all end-components, f_p the symbolic transition function and immediate reward function*

Input: min. Reward r, symb. transition f_p, immediate reward I
Output: Values V', Strategy d

1. $V' = r$;
2. $V'(\bot) = 0$;
3. $V = \infty$;
4. **while** $||V - V'|| \geq \epsilon$ **do**
5. $V = V'$;
6. $V' = \min_{\mathrm{V}_p} \sum_{\mathrm{V}_M} (f_p \cdot V'(\mathrm{V}_{M'} \leftarrow \mathrm{V}_M) + I)$
7. **end**

Algorithm 2.7: Solving the Stochastic Shortest Path Problem

$I : M \times A \to \mathbb{R}$, Algorithm 2.7 terminates and delivers an ϵ-optimal strategy and ϵ-optimal values, i.e., the value of its strategy is at most ϵ away from the optimal value in the $|| \cdot ||$-norm.

Proof See [Put94].

Using Algorithm 2.7 and Definition 2.27, we obtain the following theorem.

Theorem 2.6 (Composing strategies) *Let $\mathcal{M} = (M, m_0, A, \overline{A}, p)$ be an MDP and let $\mathcal{M}'$ be constructed from $\mathcal{M}$ as in Definition 2.27. Let $\mathcal{E} \subseteq 2^{\mathcal{M}}$ be the set of end-components of $\mathcal{M}$, d_E the optimal strategy for all end-components $E \in \mathcal{E}$, and let d' be the optimal strategy for $\mathcal{M}'$.*

We call an end-component E active if there is a state $m \in E$ such that $d'(m) = a_\bot$. Define $d(M) \in A$ for m as

- *$d(m) = d_E(m)$ if there is an active end-component $E \in \mathcal{E}$ such that $m \in E$*
- *$d(m) = d'(m)$ otherwise*

Then d is optimal for $\mathcal{M}$.

Proof First note that for every strategy d in $\mathcal{M}$ there is an analogous strategy d' in $\mathcal{M}'$, and vice versa. We define d based on d' as defined above. For the other direction, call and end-component active if the probability of visiting and staying in this end-component is greater 0. We define d' based on d by assigning $d'(m) = d(m)$ for all states m not in an active end-component. For all active end-components E and all states $m \in E$, we assign $d'(m) = a_\bot$. Finally, we assign $d'(\bot) = a_\bot$.

Name	#States	#Blocks	Time in sec	RAM in MB
*line*3	386	271	0.9	112
*line*4	1560	945	5.6	150
*line*5	5904	3089	20.5	236
*line*6	21394	9448	96.8	326
*rabin*3	27766	722	5.2	199
*rabin*4	668836	12165	104.6	537
zeroconf	89586	29427	2948.7	608
acts	1734	1734	1.6	159
*phil*6	917424	303	1.2	181
*phil*7	9043420	303	1.9	262
*phil*8	89144512	342	2.6	295
*phil*9	878732012	342	3.3	287
*phil*10	8662001936	389	4.3	303
*power*1	8904	72	0.415	89.9
*power*2	8904	n/a	n/a	85

Table 2.2: Experimental results table

Then, for every optimal strategy d for $\mathcal{M}$, d' is optimal for $\mathcal{M}'$, and vice versa.

To see why, note that $\mathbb{E}_{\mathcal{M}_d}[\mathcal{R}] = \sum_{E \in \mathcal{E}} p_d(E) r(E) = \sum_{E \in \mathcal{E}} p'_{d'}(E_\perp) r(E) = \mathbb{E}_{\mathcal{M}'_{d'}}[\mathcal{F}]$, where by $p_d(E)$ we denote the probability that a run in $\mathcal{M}$ will reach end-component E and stay in it, given that strategy d is used. Analogously, by $p'_{d'}(E_\perp)$ we denote the probability that a run in $\mathcal{M}'$ will take action $a_\perp$ from a state in E, given that strategy d is used.

We chose this construction for the symbolic encoding because no numerical computations (i.e., no equation solving) is involved. This is a great asset when dealing with MTBDDs.

Evaluation of the symbolic algorithm

Table 2.2 shows the results of our implementation on various benchmarks. The implementation can be downloaded from `http://www-verimag.imag.fr/~vonessen/ratio.html`. The first column shows the name of the example; column `#States` denotes the number of states the model has; `#Blocks` the maximum number of blocks of the partitions we construct while analyzing the model; `Time` the total time needed; `RAM` the amount of memory used (including

all memory used by PRISM and its Java Virtual Machine). Below, we briefly describe the examples and discuss the results.

Experiments.. Examples *line*3-6 model the assembly line system described in Section 2.1. We optimize the ratio between maintenance costs and number of units produced by several lines running in parallel. Example *zeroconf* is based on a model of the ZeroConf protocol [KNPS06]. We modify it to measure the best-case efficiency of the protocol, finding the expected time it takes to successfully acquire an IP address. We choose a model with two probes sent, two abstract clients and no reset. This model shows the limit of our technique when bisimulation produces many blocks. In experiments *phil*6-10, we use Lehmann's formulation of the dining philosophers problem [LR81]. Here we measure the amount of time a philosopher spends. This model is effectively a mean-payoff model because we have a cost of one for each step. We use this experiment to compare our implementation to [WBB+10]. We are several orders of magnitude faster. We attribute the increase in speed to good initial strategy. In *rabin*3 and *rabin*4, we measure the efficiency of Rabin's mutual exclusion protocol [Rab82]. We minimize the time of a process waiting for its entry into the critical section per entry into the critical section. Note that only the ratio objective allows us to measure exactly this property, because we grant a reward every time a process enters the section and a cost for every time a process has to wait for its entry. We also modeled an automatic clutch and transmission system (*acts*). Each state consists of a driver/traffic state (waiting in front of a traffic light, breaking because of a slower car, free lane), current gear (1-4) and current motor speed (100 - 500 RPM). We modeled the change of driver state probabilistically, and assumed that the driver wants to reach a given speed (50 km/h). Given this driver and traffic profile, the transmission rates and the fuel consumption based on motor speed, we synthesized the best points to shift up or down. In *power*1-2, we used the example from [NPK+05, FKN+11], which the authors use to analyze dynamic power management strategies. Our implementation allows solution of optimization problems that are not possible with either [NPK+05] or the multi-objective techniques in [FKN+11]. For example, in *power*1 we ask the question "What is the best average power consumption per served request". In *power*2, we ask for the best-case power-consumption per battery lifetime, i.e., we ask for how many hours a battery can last.

Observations.

The amount of time needed by the algorithms strongly depends on the amount of blocks it constructs. We observed that a higher number of blocks increases the time necessary to construct the partition. Each refinement step takes longer the more blocks we have. Analogously, the more blocks we have, the bigger the matrices we need to analyze. We observed an almost monotone increase in the number of blocks while policy iteration runs. Accordingly, it is beneficial to select an initial strategy with as few blocks as possible.

In the original policy iteration algorithm of Section 2.4, we constructed unichain strategies from multichain strategies several times throughout the algorithm. As it turns out, unichain strategies increase the amount of blocks dramatically. We therefore successfully modified our algorithm to avoid them, which drastically improved performance

The symbolic encoding as well as bisimulation are crucial to handle models of a size that the explicit implementation described in Section 2.4 could not handle (storing a model of the size of *phil10* was not feasible on our testing machine).

2.6 Conclusion

We have presented a framework for synthesizing efficient controllers. The framework is based on finding optimal strategies in Ratio-MDPs, including a novel closed-loop between system and environment. To compute optimal strategies we first presented three algorithms based on strategy improvement, fractional linear programming, and linear programming, respectively. We have compared performance characteristics of these algorithms and integrated the best algorithm into the probabilistic model checker PRISM. Based on these algorithms, we introduced a semi-symbolic policy iteration algorithm and reported on experiments with its integration into PRISM. This implementation proved that we can analyze large MDPs.

Future Work. There still remains work to do. Of interest are methods to scale the existing algorithms to even larger MDPs. For example, parallelization of the algorithms could be considered. In another direction, abstraction and decomposition of models to obtain smaller models are of interest. Work has been published [KKNP10, DAT10] in this area, but more research seems

necessary. A promising approach uses SMT solvers to lump Markov chains [DKP13].

Finally, a major bottle neck in our implementation is the decomposition into end-components. More research for a faster algorithm in this area would benefit our implementation.

3 Program repair without regret

In which we use the idea of quantitative synthesis to find better repairs for programs.

Résumé

Dans ce chapitre nous décrivons l'application des idées de synthèse quantitative pour réparation automatique des programme. Nous définirons la réparation des programmes au nouveau comme la recherche d'un programme modifié. Le programme d'origine n'est différent que dans les traces de spécificqtion contaminées. Ensuite nous prouvons, que cette définition est aussi strict en générale. Pour cette raison nous définissons une version simple, dans laquelle nous introduisons une deuxième specification pour caractériser quelle trace éxactement doit rester unmodifier. Nous montrons sous quelles conditions il est possible de réparer cette version simple. Si cela est possible, nous fournissons un algorithme de réparation. Finnallement nous decrivons une implimentation et les expériences finalles.

3.1 Introduction

Writing a program that satisfies a given specification usually involves several rounds of debugging. Debugging a program is often a difficult and tedious task: the programmer has to find the bug, localize the cause, and repair it. Model checking [CE81, QS82] has been successfully used to expose bugs in a program. There are several approaches [CGMZ95, ELLL01, RS04, ZH02, JRS04, GV03, BNR03, RR03] to automatically find the possible location of an error. We are interested in automatically repairing a program. Automatic program repair takes a program and a specification and searches for a correct program that satisfies the specification and is syntactically close to the origi-

nal program (cf. [BEGL99, JGB05, EKB05, GBC06, JM06, CMB08, SDE08, VYY09, CTBB11]). Existing approaches follow the same idea: first, introduce freedom into the program (e.g., by describing valid edits to the program), and then search for a way of resolving this freedom such that the modified program satisfies the specification or the given test cases. While these approaches have been shown very effective, they suffer from a common weakness: they give little or no guarantees on preserving correct behaviors (i.e., program behaviors that do not violate the specification). Therefore, a user of a repair procedure may later *regret* having applied a fix to a program because it introduced new bugs by modifying behaviors that are not explicitly specified or for which no test case is available. The approach presented by Chandra et al.[CTBB11] provides some guarantees by requiring that a valid repair needs to pass a set of positive test cases. Correct behaviors outside these test cases are left unconstrained and the repair can thus change them unpredictably.

We present the first repair approach that constructs repairs that are guaranteed to satisfy the specification and that are not only syntactically, but also semantically close to the original program. The key benefits of our approach are: (i) it maintains correct program behavior, (ii) it is robust w.r.t. generous program modifications, i.e., it does not produce degenerated programs if given too much freedom in modifying the program, (iii) it works well with incomplete specifications, because it considers the faulty program as part of the specification and preserves its core behavior, and finally (iv) it is easy to implement on top of existing technology. We believe that our framework will prove useful because it does not require a complete specification by taking the program as part of the specification. It therefore makes writing specifications for programs easier. Furthermore, specifications are often given as conjunctions of smaller specifications that are verified individually. In order to keep desired behaviors, classical repair approaches repair a program with respect to the entire specification. Our approach can provide meaningful repair suggestions while focusing only on parts of the specification.

3.2 On languages

In addition to the definitions from Section 1.4, we need the following definitions. Let AP be the finite set of *atomic propositions*, as in Section 1.4.

Definition 3.1 (Alphabet over letters) We define the *alphabet* over AP (denoted Σ_{AP}) as the set of all evaluations of AP, i.e., $\Sigma_{AP} = 2^{AP}$. If AP is clear from the context or not relevant, then we omit the subscript in Σ_{AP}.

We are now going to partition AP into sets I and O, where I are input symbols and O are output symbols. Given a word $w \in \Sigma_{AP}^{\omega}$ consisting of letters made of input and output symbols, we will define the restriction of w to the set of inputs by cutting away all output symbols. Analogously, given a word consisting of only input symbols we will define its extension as the set of words you can create by adding output symbols.

Definition 3.2 (Restricted and extended words) Given a set of propositions $\mathrm{I} \subseteq AP$, we define the I-*restriction* of a word $w \in \Sigma_{AP}^{\omega}$, denoted by $w{\downarrow}_{\mathrm{I}}$, as $w{\downarrow}_{\mathrm{I}} = l_0 l_1 \cdots \in \Sigma_{\mathrm{I}}^{\omega}$ with $l_i = (w_i \cap \mathrm{I})$ for all $i \geq 0$. Given a language $\mathrm{L} \subseteq \Sigma_{AP}^{\omega}$ and a set $\mathrm{I} \subseteq AP$, we define the I-*restriction* of L, denoted by $\mathrm{L}{\downarrow}_{\mathrm{I}}$, as the set of I-restrictions of all the words in L, i.e., $\mathrm{L}{\downarrow}_{\mathrm{I}} = \{w{\downarrow}_{\mathrm{I}} \mid w \in \mathrm{L}\}$. Given a word $w \in \Sigma_{\mathrm{I}}^{\omega}$ over a set of propositions $\mathrm{I} \subseteq AP$, we use $w{\uparrow}_{AP}$ to denote the *extension* of w to the alphabet Σ_{AP}, i.e., $w{\uparrow}_{AP} = \{w' \in \Sigma_{AP}^{\omega} \mid w'{\downarrow}_{I} = w\}$. Extension of a language $\mathrm{L} \subseteq \Sigma_{\mathrm{I}}^{\omega}$ is defined analogously, i.e., $\mathrm{L}{\uparrow}_{AP} = \{w{\uparrow}_{AP} \mid w \in \mathrm{L}\}$.

If a language contains for each input word w at most one word whose restriction is w, then we call this language input deterministic. In effect this means that the language allows at most one possible output for each input. If a language contains for each input word w at least one word whose restriction is w, then we call this language input complete. This means that the language admits at least one output to each input.

Definition 3.3 (Input completeness and output determinism) A language $\mathrm{L} \subseteq \Sigma_{AP}^{\omega}$ is called I-*deterministic* for some set $\mathrm{I} \subseteq AP$ if for each word $v \in \Sigma_{\mathrm{I}}^{\omega}$ there is at most one word $w \in \mathrm{L}$ such that $w{\downarrow}_{\mathrm{I}} = v$. A language L is called I-*complete* if for each input word $v \in \Sigma_{\mathrm{I}}^{\omega}$ there exists at least one word $w \in \mathrm{L}$ such that $w{\downarrow}_{I} = v$.

A Büchi automaton is an automaton in conjunction with a parity objective with co-domain $\{1, 2\}$.

Definition 3.4 A *Büchi automaton* is a tuple $\mathrm{A} = (\mathrm{S}, \mathrm{s}_0, \Sigma, \Delta, \mathrm{F})$ such that $(\mathrm{S}, \mathrm{s}_0, \Sigma, \Delta)$ is an automaton. $\mathrm{F} \subseteq \mathrm{S}$ is the set of *accepting states*. A word is

accepted by A if there exists a run $s_0\, s_1 \ldots$ such that $s_i \in F$ for infinitely many i. We denote by $L(A)$ the language of the Büchi automaton, i.e., the set of words accepted by A. A language that is accepted by a Büchi automaton is called ω-regular.

For every LTL formula φ (see Definition 1.11 on Page 20) one can construct a Büchi automaton A such that $L(A) = L(\varphi)$ [WVS83, LP85].

We will use the following lemma in Section 3.4. It follows directly from the definition (i.e., from the fact that $\delta_{\mathcal{O}}$ is a function).

Lemma 3.1 (Machine languages) *The language* $L(M)$ *of any machine* $M = (S_{\mathcal{O}}, o_0, \Sigma_I, \Sigma_O, \delta_{\mathcal{O}}, \Omega_{\mathcal{O}})$ *is I-deterministic (input deterministic) and I-complete (input complete).*

Realizability and synthesis problem.

For LTL formulas, there is an algorithm to decide realizability and to solve the synthesis problem.

Theorem 3.1 (Synthesis Algorithms [BL69, Rab69, PR89]) *There exists a deterministic algorithm that checks whether a given LTL-formula (or an ω-regular language) φ is* realizable*. If φ is realizable, then the algorithm constructs* M.

3.3 Example

In this section we give a simple example to motivate our definitions and highlight the differences to previous approaches such as [JGB05].

Example 3.1 (Traffic Light) Assume we want to develop a sensor-driven traffic light system for a crossing of two streets. For each street entering the crossing, the system has two sets of lights (called `light1` and `light2`) and two sensors (called `sensor1` and `sensor2`). By default both lights are red. If a sensor detects a car, then the corresponding lights should change from red to yellow to green and back to red. We are given the implementation shown in Figure 3.1 as starting point. It behaves as follows: for each red light, the system checks if the sensor is activated (Line 12 and 18). If yes, this light becomes yellow in the next step, followed by a green phase and a subsequent red phase. Assume we require that our implementation is safe, i.e., the two

```
1   typedef enum {RED, YELLOW, GREEN} traffic_light;
2   module Traffic (clock, sensor1, sensor2, light1, light2);
3     input clock, sensor1, sensor2;
4     output light1, light2;
5     traffic_light reg light1, light2;
6     initial begin
7       light1 = RED;
8       light2 = RED;
9     end
10    always @(posedge clock) begin
11      case (light1)
12        RED: if (sensor1) // Repair: if (sensor1 & !(light2==RED & sensor2))
13                light1 = YELLOW;
14        YELLOW: light1 = GREEN;
15        GREEN: light1 = RED;
16      endcase // case (light1)
17      case (light2)
18        RED: if (sensor2)
19                  light2 = YELLOW;
20        YELLOW: light2 = GREEN;
21        GREEN: light2 = RED;
22      endcase // case (light1)
23    end // always (@posedge clock)
24  endmodule // traffic
```

Figure 3.1: Implementation of a traffic light system and a repair

lights are never green at the same time. In LTL, this specification is written as $\varphi = \mathsf{G}(\texttt{light1} \neq \texttt{GREEN} \vee \texttt{light2} \neq \texttt{GREEN})$. The current implementation clearly does not satisfy this requirement: if both sensors detect a car initially, then the lights will simultaneously move from red to yellow and then to green, thus violating the specification.

Following the approach in [JGB05] we introduce a non-deterministic choice into the program and then use a synthesis procedure to select among these options in order to satisfy the specification. For instance, we replace Line 12 (in Figure 3.1) by `if(?)` and ask the synthesizer to construct a new expression for `?` using the input and state variables. The synthesizer aims to find a simple expression s.t. φ is satisfied. In this case one simple admissible expression is `false`. It ensures that the modified program satisfies specification φ. While this repair is correct, it is very unlikely to please the programmer because it repairs "too much": it modifies the behavior of the system on input traces on which the initial implementation was correct. We believe it is more desirable to follow the idea of Chandra et al. [CTBB11] saying that a repair is only allowed

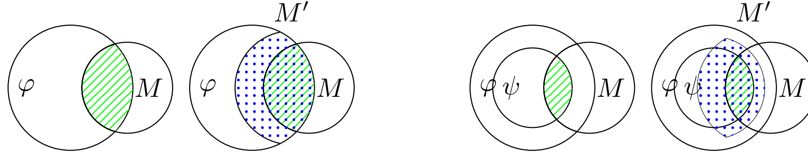


Figure 3.2: Graphical representation of Def. 3.5

Figure 3.3: Graphical representation of Def. 3.6

to change the behavior of incorrect executions. In our case, the repair suggested above would not be allowed because it changes the behavior on correct traces, as we will show in the next section.

3.4 Repair

In this section we first give a repair definition for reactive systems which follows the intuition that a repair can only change the behavior of incorrect executions. Then, we provide an algorithm to compute such repairs.

Definitions

Given a machine M and a specification φ, we say a machine M′ is an exact repair of M if (i) M′ behaves like M on traces satisfying φ and (ii) if M′ implements φ. Intuitively, the correct traces of M act as a *lower bound* for M′ because they must be included in L(M′). L(φ) acts as an *upper bound* for M′, i.e., it specifies the allowed traces.

Definition 3.5 (Exact Repair) A machine M′ is an exact repair of a machine M for a specification φ, if (i) all the correct traces of M are included in the language of M′, and (ii) if the language of M′ is included in the language of the specification φ, i.e.,

$$\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\varphi) \subseteq \mathrm{L}(\mathrm{M}') \subseteq \mathrm{L}(\varphi)$$

Note that the first inclusion defines the behavior of M′ on all input words to which M responds correctly according to φ. In other terms, M′ has only one choice forn inputs which M treat correctly. Figure 3.2 illustrates Definition 3.5: the two circles depict L(M) and L(φ). A repair has to (i) cover their intersection (first inclusion in Definition 3.5), which we depict with the striped area in the

picture, and (ii) lie within $L(\varphi)$ (second inclusion in Definition 3.5). One such repair is depicted by the dotted area on the right.

Example 3.2 (Traffic Light, cont.) The repair suggested in Example 3.1 (i.e., to replace `if (sensor1)` by `if (false)`) is not a valid repair according to Definition 3.5. The original implementation responds correctly, e.g., to the input trace in which `sensor1` is always high and `sensor2` is always low, but the repair produces different outputs. The initial implementation behaves correctly on any input trace on which `sensor1` and `sensor2` are never high simultaneously. Any correct repair should include these input/output traces. An exact repair (i.e, a repair according to Definition 3.5) replaces `if (sensor1)` by `if (sensor1 & !(light2 == RED & sensor2))`. This repair retains all correct traces while avoiding the mutual exclusion problem.

While Definition 3.5 excludes the undesired repair in our example, it is sometimes too restrictive and can make repair impossible, as the following example shows.

Example 3.3 (Definition 3.5 is too restrictive) Assume a machine M with input r and output g that always copies r to g, i.e., M satisfies $\mathsf{G}(r \leftrightarrow g)$. The specification requires that g is eventually high, i.e., $\varphi = \mathsf{F}\, g$. Definition 3.5 requires the repaired machine M' to behave like M on all traces on which M behaves correctly. M responds correctly to all input traces containing at least one r, i.e., $L(M) \cap L(\varphi) = \mathsf{F}(r \wedge g)$. Intuitively, M' has to mimic M as long as M still has a chance to satisfy φ (i.e., to produce a trace satisfying $\mathsf{F}(r \wedge g)$). Since M always has a chance to satisfy φ, M' has to behave like M in every step, therefore M' also violates φ, and cannot be repaired in this case.

In order to allow more repairs, we *relax* the restriction requiring that all correct traces are included in the following definition.

Definition 3.6 (Relaxed Repair) Let ψ define a language (by an LTL-formula or a Büchi automaton). We say M' is a *repair of* M *with respect to* ψ *and* φ if M' behaves like M on all traces satisfying ψ and M' implements φ. That is, M' is a repair constructed from M iff

$$L(M) \cap L(\psi) \subseteq L(M') \subseteq L(\varphi) \tag{3.1}$$

In Figure 3.3 we give a graphical representation of this definition. The two concentric circles depict φ and ψ. (The definition does not require that $\mathrm{L}(\psi) \subseteq \mathrm{L}(\varphi)$, but for simplicity we depict it like that.) The overlapping circle on the right represents M. The intersection between ψ and M (the striped area in Figure 3.3) is the set of traces M′ has to mimic. On the right of Figure 3.3, we show one possible repair (represented by the dotted area). The repair covers the intersection of L(M) and $\mathrm{L}(\psi)$, but not the intersection of $\mathrm{L}(\varphi)$ and L(M). The repair lies completely in $\mathrm{L}(\varphi)$. The choice of ψ influences the existence of a repair. In Section 3.5 we discuss several choices for ψ.

Example 3.4 (Example 3.3 continued) Example 3.3 shows that setting ψ to φ, i.e., $\mathsf{F}\, g$ in our example, can be too restrictive. If we relax ψ and require it only to include all traces in which g is true within the first n steps for some given n (i.e., $\psi = \bigvee_{0 \leq i \leq n} \mathsf{X}^n\, g$), then we can find a repair. A possible repair is a machine M′ that copies r to g in the first n steps and keeps track if g has been high within these steps. In this case, M′ continues mimicing M, otherwise it sets g to high in step $n+1$, independent of the behavior of M. This way M′ satisfies the specification ($\mathsf{F}\, g$) and mimics M for all traces satisfying ψ.

Reduction to Classical Synthesis

The following theorem shows that our repair problem can be reduced to the classical synthesis problem.

Theorem 3.2 *Let φ, ψ be two specifications and* M, M′ *be two machines with input signals I and output signal O. Machine* M′ *satisfies Formula 3.1* ($\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\psi) \overset{(a)}{\subseteq} \mathrm{L}(\mathrm{M}') \overset{(b)}{\subseteq} \mathrm{L}(\varphi)$) *if and only if* M′ *satisfies the following formula:*

$$\mathrm{L}(\mathrm{M}') \subseteq \underbrace{\big((\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\psi)){\downarrow_I}{\uparrow_{AP}} \rightarrow \mathrm{L}(\mathrm{M})\big)}_{(i)} \cap \underbrace{\mathrm{L}(\varphi)}_{(ii)} \tag{3.2}$$

For two languages A and B, $A \rightarrow B$ is an abbreviation for $(\Sigma^\omega \setminus A) \cup B$. Intuitively, Equation 3.2 requires that (i) M′ *behaves like* M *on all input words that* M *answers conforming to ψ and (ii)* M *satisfies specification φ.*

Proof From left to right: We have to show that $\mathrm{L}(\mathrm{M}')$ is included in (i) and (ii). Inclusion in (ii) follows trivially from (b). It remains to show $\mathrm{L}(\mathrm{M}') \subseteq \big(\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\psi)\big){\downarrow_I}{\uparrow_{AP}} \rightarrow \mathrm{L}(\mathrm{M})$. Let $w \in \mathrm{L}(\mathrm{M}')$. If $w \notin \big(\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\psi)\big){\downarrow_I}{\uparrow_{AP}}$, then the implication follows trivially. Otherwise we have to show that $w \in$

L(M). Since $w \in \big(\mathrm{L(M)} \cap \mathrm{L}(\psi)\big)\downarrow_I\uparrow_{AP}$, it follows that $w\downarrow_I \in \big(\mathrm{L(M)} \cap \mathrm{L}(\psi)\big)\downarrow_I$. From $w\downarrow_I \in \big(\mathrm{L(M)} \cap \mathrm{L}(\psi)\big)\downarrow_I$ and the fact that $L(M)$ is input deterministic, we know that $M(w\downarrow_I) \in \mathrm{L(M)} \cap \mathrm{L}(\psi) \subseteq \mathrm{L(M')}$ (due to (a)). Together with L(M′) being input deterministic, it follows that $M(w\downarrow_I) = M'(w\downarrow_I) = w$, and so $w \in \mathrm{L(M)}$ holds.

From right to left: We have to show (a) and (b). (b) follows trivially from $\mathrm{L(M')} \subseteq (ii)$. It remains to show (a), i.e., that $\mathrm{L(M)} \cap \mathrm{L}(\psi) \subseteq \mathrm{L(M')}$. Assume a word $w \in \mathrm{L(M)} \cap \mathrm{L}(\psi)$, we have to show that $w \in \mathrm{L(M')}$. Let $w' \in \mathrm{L(M')}$ be a word such that $w\downarrow_I = w'\downarrow_I$. Note that w' exists because L(M′) is input complete. We now show that $w = w'$, which implies that $w \in \mathrm{L(M')}$. Since $w \in \mathrm{L(M)} \cap \mathrm{L}(\psi)$, it follows that $w\downarrow_I$ $(=w'\downarrow_I) \in (\mathrm{L(M)} \cap \mathrm{L}(\psi))\downarrow_I$. Therefore, we know that $w' \in \big(\mathrm{L(M)} \cap \mathrm{L}(\psi)\big)\downarrow_I\uparrow_{AP}$. From $\mathrm{L(M')} \subseteq (i)$ and from $w' \in \mathrm{L(M')}$, it follows that $w' \in \mathrm{L(M)}$. Since L(M) is input deterministic, $w \in \mathrm{L(M)}$, $w' \in \mathrm{L(M)}$, and $w\downarrow_I = w'\downarrow_I$, it follows that $w = w'$.

This theorem leads together with [PR89] to the following corollary, which allows us to use classical synthesis algorithms to compute repairs.

Corollary 3.1 (Existence of repair) *A repair can be constructed from a machine* M *with respect to specifications* ψ *and* φ *if and only if the language*

$$\big((\mathrm{L(M)} \cap \mathrm{L}(\psi))\downarrow_I\uparrow_{AP} \rightarrow \mathrm{L(M)}\big) \cap \mathrm{L}(\varphi) \tag{3.3}$$

is realizable.

Algorithm

Corollary 3.1 gives an algorithm to construct repairs based on synthesis techniques (cf. [JGB05]). In order to compute the language defined by Formula 3.3, we can use standard automata-theoretic operations. More precisely, we construct a Büchi automaton A_φ recognizing φ and a Büchi automaton A_ψ recognizing ψ. Note that M is a Büchi automaton in which all states are accepting. Since Büchi automata are closed under conjunction, disjunction, projection, and complementation, we can construct an automaton for $\big(\overline{(\mathrm{M} \times \mathrm{A}_\psi)|_I} + \mathrm{M}\big) \times \mathrm{A}_\varphi$, where $A \times B$ denotes the conjunction, $A + B$ denotes the disjunction of automata A and B, $\bar{A}$ denotes the complementation of A, and $A|_I$ the projection of automaton A with respect to a set of proposition I. Once we have a

Büchi automaton for the language in Formula 3.3, we can use Theorem 3.1 to synthesize a repair.

This algorithm in unlikely to scale because the complementation of a Büchi automaton induces an exponential blow-up in the worst case [DH94]. Furthermore, the projection operator can introduce non-determinism that can complicate the application of a synthesis procedure due to the need of an additional determinization step, leading to another exponential blow-up [Pit07, Sch09]. In the following we show how to obtain an efficient algorithm by avoiding complementation (Lemma 3.2) and projection (Lemma 3.3).

Lemma 3.2 *Given a machine* M *with input signals I and output signals O and an LTL-formula φ over the atomic propositions $AP = I \cup O$, the following equalities hold:*

$$\Sigma_I^\omega \setminus \big(\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\varphi)\big){\downarrow_I} \quad = \quad \big(\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\neg\varphi)\big){\downarrow_I} \tag{3.4}$$

$$\Sigma_{AP}^\omega \setminus \big(\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\varphi)\big){\downarrow_I}{\uparrow_{AP}} \quad = \quad \big(\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\neg\varphi)\big){\downarrow_I}{\uparrow_{AP}} \tag{3.5}$$

Proof Intuitively, Equation 3.4 means that the set of input words on which M behaves correctly, i.e., satisfies φ, is the complement of the set of inputs on which M behaves incorrectly, i.e., violates φ and therefore satisfies $\neg\varphi$. Formally, we know from the semantics of LTL that $\mathrm{L}(\neg\varphi) = \Sigma^\omega \setminus \mathrm{L}(\varphi)$, which implies that

$$\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\neg\varphi) \overset{(a)}{=} \mathrm{L}(\mathrm{M}) \cap \big(\Sigma^\omega \setminus \mathrm{L}(\varphi)\big) \overset{(b)}{=} \mathrm{L}(\mathrm{M}) \setminus \mathrm{L}(\varphi). \tag{3.6}$$

Equality 3.6.b follows from simple set theory. Furthermore, since L(M) is input deterministic and input complete, we know that

$$\forall\, w, w' \in \mathrm{L}(\mathrm{M}) : (w{\downarrow_I} = w'{\downarrow_I}) \rightarrow w = w' \tag{3.7}$$

$$\forall\, w \in \Sigma_{AP}^\omega : \exists\, w' \in \mathrm{L}(\mathrm{M}) : w{\downarrow_I} = w'{\downarrow_I} \tag{3.8}$$

We use these facts to show that for all $A \subseteq \Sigma^\omega$, $\Sigma_I^\omega \setminus \big(\mathrm{L}(\mathrm{M}) \cap A\big){\downarrow_I} = \big(\mathrm{L}(\mathrm{M}) \setminus A\big){\downarrow_I}$ holds, which proves together with Equation 3.6 that Equation 3.4 is true:

$$
\begin{aligned}
v \in \big(\mathrm{L(M)} \setminus A\big)\!\downarrow_I &\iff \exists w \in \mathrm{L(M)} \setminus A : (w\!\downarrow_I = v) \\
&\iff \exists w \in \mathrm{L(M)} : (w\!\downarrow_I = v) \wedge w \notin A \\
&\overset{\text{Eq.3.7}}{\underset{\text{Eq.3.8}}{\iff}} \forall w \in \mathrm{L(M)} : (w\!\downarrow_I = v) \rightarrow w \notin A \\
&\iff \forall w \in \mathrm{L(M)} : w \in A \rightarrow (w\!\downarrow_I \neq v) \\
&\iff \forall w \in \mathrm{L(M)} \cap A : (w\!\downarrow_I \neq v) \\
&\iff \nexists w \in \mathrm{L(M)} \cap A : (w\!\downarrow_I = v) \quad \iff v \notin \big(\mathrm{L(M)} \cap A\big)\!\downarrow_I
\end{aligned}
$$

Equation 3.5 is a simple extension of Equation 3.4 to the alphabet Σ_{AP}. It follows from the fact that for any language $\mathrm{L} \subseteq \Sigma_I^\omega : (\Sigma I^\omega \setminus \mathrm{L})\!\uparrow_{AP} = \Sigma_I^\omega\!\uparrow_{AP} \setminus \mathrm{L}\!\uparrow_{AP}$ holds.

With the help of Lemma 3.2 we can simplify Formula 3.3 to

$$
\big((\mathrm{L(M)} \cap \mathrm{L}(\neg\psi))\!\downarrow_I\uparrow_{AP} \cup \mathrm{L(M)}\big) \cap \mathrm{L}(\varphi) \tag{3.9}
$$

This allows us to compute a repair using a synthesis procedure for the automaton $\big((\mathrm{M} \times \mathrm{A}_{\neg\psi})|_I + \mathrm{M}\big) \times \mathrm{A}_\varphi$, which is much simpler to construct.

Lemma 3.3 (Avoiding input projection) *Given a machine* M *and an LTL-formula* φ*, for every word* $w \in \Sigma^\omega$, $w \in (\mathrm{L(M)} \cap \mathrm{L}(\varphi))\!\downarrow_I\uparrow_{AP} \iff \mathrm{M}(w\!\downarrow_I) \in \mathrm{L}(\varphi)$ *holds.*

Proof

$$
\begin{aligned}
w \in (\mathrm{L(M)} \cap \mathrm{L}(\varphi))\!\downarrow_I\uparrow_{AP} &\iff w\!\downarrow_I \in (\mathrm{L(M)} \cap \mathrm{L}(\varphi))\!\downarrow_I \\
&\iff \exists w' \in \mathrm{L(M)} \cap \mathrm{L}(\varphi) : w'\!\downarrow_I = w\!\downarrow_I \\
&\iff \exists w' \in \mathrm{L(M)} : w'\!\downarrow_I = w\!\downarrow_I \wedge w' \in \mathrm{L}(\varphi) \\
&\iff \mathrm{M}(w\!\downarrow_I) \in \mathrm{L}(\varphi)
\end{aligned}
$$

Due to Lemma 3.3 we can check if a word produced by M′ lies in $(\mathrm{L(M)} \cap \mathrm{L}(\varphi))\!\downarrow_I\uparrow_{AP}$ by checking whether M treats the input projection of that word correctly. A synthesizer looking for a solution to Equation 3.9 can simulate M and check its output against $\neg\psi$ to decide

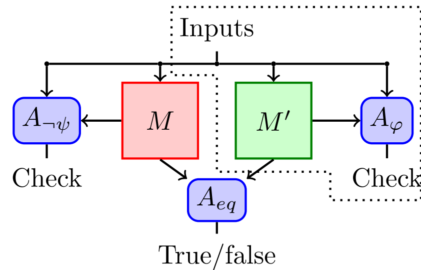


Figure 3.4: Efficient implementation

whether M′ is allowed to deviate from M.
This allows us to solve our repair prob-
lem using the simple setup we depict in Figure 3.4. It shows five automata running in parallel:

1. The original machine M.
2. The repair candidate M′, a copy of M that includes multiple options to modify M.
3. A specification automaton A_φ to check if the new machine M′ satisfies its objective.
4. A specification automaton $A_{\neg\psi}$ to check if the original machine M violates ψ.
5. A specification automaton A_{eq} that checks if the outputs of M and M′ coincide, i.e., $eq = \mathsf{G}(\bigwedge_{o \in O} o \leftrightarrow o')$, where O is the set of outputs of M and o' is the copy of output $o \in O$ in machine M′.

Theorem 3.3 *Given the setup depicted in Figure 3.4, a repair option in* M′ *is a valid repair according to Definition 3.6, if it satisfies the formula*

$$\varphi \wedge (\neg\psi \vee eq). \tag{3.10}$$

Proof Follows from Lemma 3.2 and Lemma 3.3.

Formula 3.10 forces M′ to (1) behave according to φ and (2) mimic the behavior of M, if M satisfies ψ. Note that all automata can be constructed separately because they can be connected through the winning (or acceptance) condition. We avoid the monolithic construction of a specification automaton and obtain the same complexity as for classical repair. E.g., if φ, $\neg\psi$, and eq are represented by Büchi automata, then we can check for $\varphi \wedge (\neg\psi \vee eq)$ by first merging the acceptance states of $\neg\psi$ and eq, and then solving for a generalized Büchi condition, which is quadratic in the size of the state space ($|A_{\neg\psi}| \times |M| \times |M'| \times |A_\varphi| \times 2$).

Implementation

Our prototype implementation is based on the following two ideas:

1. If a synthesis problem can be decided by looking at a finite set of possible repairs [1] (combinations of choices), then the choice of repair can be encoded using multiple initial states.

2. An initial state that does not lead to a counter example represents a correct repair. Any model checker can be adapted to return such an initial state, if one exists. By default a model checker returns the opposite, i.e., an initial state that leads to a counter-example but it is not difficult to change it. E.g., in BDD-based model-checkers some simple set operations suffice and in SAT-based checkers one can make use of unsat-core to eliminate failing initial states.

The main drawback of this approach is that the state space is multiplied by the number of considered repairs. However, the approach has several benefits which make it particularly interesting for program repair. First, it is easy to restrict the set of repairs to those that are simple and readable. In our prototype implementation we adapt the idea of Solar-Lezama et al. [SLRBE05] and search for a repair within a given set of user-defined expressions. In the examples, we derive these expressions manually from the operators used in the program (see Section 3.6 for more details). Furthermore, we assume a given fault location that will be replaced by one of the user-defined expressions (cf. [JGB05, JSGB12]). Expression generation and fault localization are interesting and active research directions (cf. Section 3.1) but are not addressed in this chapter. We focus on the problem of deciding what constitutes a good repair. The second main benefit is that we can adapt an arbitrary model checker to solve our repair problem. We believe (based on initial experiments) that at the current state, model checkers are significantly more mature than synthesis frameworks. In our implementation we used a version of NuSMV [CCG+02] that we slightly modified to return an initial state that does not lead to a counter example.

Note that using the sketch-like approach (i.e., using a set of expressions to choose from) is an implementation choice. It does not take away from the generality of our approach. An implementation generating stateful repairs as in [JGB05, JSGB12] is possible.

[1] Note that any synthesis problem with memoryless winning strategies satisfies this condition.

3.5 Discussion and limitations

In this section we discuss choices for ψ and analyze why a repair can fail.

Choices for ψ

We present several different choices for ψ and analyze their strengths and weaknesses:

1. $\psi = \varphi$
2. If $\varphi = f \rightarrow g$, then $\psi = f \wedge g$
3. $\psi = \emptyset$

In addition we suggest an approach based on Markov decision processes.

Exact. Choosing $\psi = \varphi$ is the most restrictive choice. It requires that M′ behaves like M on all words that are correct in M. While this is in general desirable, this choice can be too restrictive as Example 3.3 in Section 3.4 shows. One might think that the problem in Example 3.3 is that φ is a liveness specification. The following example shows that choosing $\psi = \varphi$ can also be too restrictive for safety specifications.

Example 3.5 Let M be a machine with input r and output g; M always outputs $\neg g$, i.e., M implements $\mathsf{G}(\neg g)$. Assume $\varphi = \mathsf{F}(\neg r) \rightarrow \mathsf{G}(g) = \mathsf{G}(r) \vee \mathsf{G}(g)$. Applying Formula 3.9, we obtain $(\mathsf{G}(\neg g) \wedge \neg(\mathsf{G}(r) \vee \mathsf{G}(g)))\downarrow_I\uparrow_{AP}$[2] $\wedge (\mathsf{G}(r) \vee \mathsf{G}(g)) = (\mathsf{F}(\neg r) \wedge \mathsf{G}(g)) \vee (\mathsf{G}(r) \wedge \mathsf{G}(\neg g))$. This formula is not realizable because a machine does not know if the environment will always send a request ($\mathsf{G}(r)$) or if the environment will eventually stop sending a request ($\mathsf{F}(\neg r)$). A correct machine has to respond differently in these two cases. So, M cannot be repaired if $\psi = \varphi$.

Assume-Guarantee. It is very common that the specification is of the form $f \rightarrow g$ (as in the previous example). Usually, f is an assumption on the environment and g is the guarantee the machine has to satisfy if the environment meets the assumption. Since we are only interested in the behavior of M if the assumption is satisfied, it is reasonable to ask the repair to mimic only traces on which the assumption and the guarantee is satisfied, i.e., choosing $\psi = f \wedge g$.

[2] LTL is not closed under projection. We use LTL only to describe the corresponding automata computations.

Example 3.6 (Example 3.5 continued) Recall Example 3.5, we decompose φ into assumption $\mathsf{F}\neg r$ and guarantee $\mathsf{G}\,g$. Now, we can see that M is only correct on words on which the assumption is violated, so the repair should not be required to mimic the behavior of M. If we set $\psi = \mathsf{F}\neg r \wedge \mathsf{G}\,g$, then $L(M) \cap L(\psi) = \emptyset$ and M′ is unrestricted on all input traces.

Unrestricted. If we choose $\psi = \emptyset$ the repair is unrestricted and the approach coincides with the work presented in [JGB05].

Reward based formulation. The first inclusion in Definition 3.6 strictly defines the set of traces of a machine M a repaired machine M′ has to include. We can relax this requirement using rewards. Using rewards allows us to ask for the machine that *agrees most of the time* with M and for the machine that *agrees on the most traces* with M. We will first present an example showing where relaxing ψ makes sense.

Letter-Optimal Solutions. One intuitive solution to the repair problem is a machine that "modifies the least possible output letters". If φ is a safety condition, then we are certain that we can win by staying in the safe region. Therefore, our task is twofold: (1) stay in the safe region and (2) minimize the number of times M′ chooses an output that differs from the output of M on average. This can be achieved using a mean payoff objective, as the following example illustrates.

Example 3.7 Let $\varphi = \mathsf{G}(r \rightarrow g)$, and let M fulfill $\mathsf{G}((g \leftrightarrow \mathsf{X}\neg g))$, i.e., the machine signals g in every second step. If we choose ψ to recognize the safe region of φ, then a repairing machine can output $\mathsf{G}(g)$, once M violates its specification, thus choosing an output that differs from that of M infinitely often.

Another option is to reward M′ whenever it copies the behavior of M. To calculate such a machine M′, we build a game from the safe region of φ and M. Player 0 wins the game if she wins the safety game. The game is played in rounds. First player 1 picks an input from Σ_{I}, and then player 0 picks an output form Σ_{O}. The game grants a reward whenever the output chosen at a state is equal to the output M chooses at this state.

In this example, an optimal strategy (i.e., the strategy that maximizes the grant) will give a grant in every second step and whenever r is signaled.

We can generalize this idea to reduce the repair problem to two-player games with mean-payoff and parity objectives. For a discussion and implementation of these games, see e.g., [BBFR13].

Definition 3.7 (Letter-optimal repair) Let $\mathcal{S} = ((\mathrm{S} \times \Sigma_{AP}), \mathrm{S}_0, \mathrm{S}_1, \mathrm{s}_0, \Delta)$ be a two player game with a parity objective $\lambda : \mathrm{S} \times \Sigma_{AP} \to \mathbb{N}$ such that player 0 controls the output symbols and player 1 controls the input symbols, i.e., such that $(\mathrm{s}, i \cup o) \in \mathrm{S}_0 \wedge ((\mathrm{s}, i \cup o), (\mathrm{s}', i' \cup o')) \in \Delta \implies i = i'$ and $(\mathrm{s}, i \cup o) \in \mathrm{S}_1 \wedge ((\mathrm{s}, i \cup o), (\mathrm{s}', i' \cup o')) \in \Delta \implies o = o'$, and such the game is played in turns (such a game is generated, for example, to synthesize an LTL-formula). Let further $\mathrm{M} = (M, m_0, \Sigma_\mathrm{I}, \Sigma_\mathrm{O}, \delta, \Omega)$ be a machine.

Then we define the *Letter-Optimal parity game* $\mathcal{S}'$ by combining $\mathcal{S}$ with M as follows. Let $\mathcal{S}' = (M \times (\mathrm{S} \times \Sigma_{AP}), M \times \mathrm{S}_0, M \times \mathrm{S}_1, (m_0, \mathrm{s}_0), \Delta')$ be a two player game, where $((m, (\mathrm{s}, i \cup o)), (m', (\mathrm{s}', i' \cup o'))) \in \Delta'$ if and only if $((\mathrm{s}, i \cup o), (\mathrm{s}', i' \cup o')) \in \Delta$ and

$$s' = \begin{cases} m & \text{if } (b, i \cup o) \in \mathrm{S}_1 \\ \delta(m, i) & \text{if } (b, i \cup o) \in \mathrm{S}_0 \end{cases}$$

As reward function, we define $r((m, (\mathrm{s}, i \cup o))) = 1$ if $\Omega(m, i) = o$ and $(\mathrm{s}, i) \in \mathrm{S}_1$, and 0 otherwise. That is, a reward is assigned if the output of the machine and the output of the last player 0 transition agree.

In addition, we define $\lambda' : M \times (\mathrm{S} \times \Sigma_{AP}) \to \mathbb{N}$ by $\lambda'(m, (\mathrm{s}, i \cup o)) = \lambda((\mathrm{s}, i \cup o))$, i.e., we copy the parity condition.

A letter-optimal repair is a winning and mean-payoff optimal strategy for the defined game and vice versa. The parity condition makes sure that the repair is qualitatively correct (e.g., fulfills an LTL-formula), while the quantitative (mean-payoff) condition makes sure that as few letters as possible are modified. Note that this might require infinite memory strategies in general, but ϵ-optimal finite memory strategies exist [BBFR13].

Trace-Optimal Solutions. Another intuitive solution to the repair problem is a machine that "modifies the least number of traces on average". We model the "on average" part using uniformly distributed inputs. Since this means that the input we receive is not adversarial but probabilistic, this approach does not work with every kind of qualitative specification, but only with safety specifications.

Example 3.8 Consider the following formula.

$$\varphi = (i \wedge \mathsf{X}\, i \rightarrow ((o \wedge \mathsf{X}\, o) \vee (\neg o \wedge \mathsf{X} \neg o))) \wedge (i \wedge \mathsf{X} \neg i \rightarrow (\neg o \wedge \mathsf{X} \neg o))$$

as specification over input alphabet $\Sigma_{\mathrm{I}} = \{i\}$ and output alphabet $\Sigma_{\mathrm{O}} = \{o\}$. It requires on input $ii\ldots$ either output $oo\ldots$ or output $\neg o \neg o\ldots$. On input $i\neg i$ it requires output $\neg o \neg o$. Consider further M writing o constantly as machine we need to repair. The machine is correct on input words starting with ii and words starting with $\neg i$. It is incorrect on all words starting with $i \neg i$.

Our goal is to minimize the number of traces that are modified. We therefore assign to each trace a payoff, either 1 or 0. An unchanged trace gets payoff 1, while a changed trace gets payoff 0. To "count" the number of changed traces in a set of traces, we take the average payoff of all traces in that set. If all traces are changed, then the payoff is 0; if no traces are changed, then the payoff is 1. If half of the traces have changed, then the payoff is 0.5.

The minimal number of traces a repaired system has to change is all traces that start with i, i.e., 50% of all possible traces.

The following definition provides an MDP whose optimal strategy provides a machine that is correct for a safety specification and has a minimal number of changed traces.

Definition 3.8 (Trace-optimal MDP) Let $\mathrm{A} = (\mathrm{S}, \mathrm{s}_0, \Sigma, \Delta, \mathrm{F})$ be a realizable, deterministic safety automaton, and let $\mathrm{M} = (M, m_0, \Sigma_{\mathrm{I}}, \Sigma_{\mathrm{O}}, \delta, \Omega)$ be a machine not fulfilling A.

We define the *Word-Optimal MDP* $\mathcal{M}$ as follows. Let $\mathcal{M} = ((M \cup \{\bot, \top\}) \times \mathrm{S} \times \Sigma_{\mathrm{I}}, (\top, \mathrm{s}_0, i_0), \Sigma_{\mathrm{O}}, \overline{A}, p)$ be an MDP, where i_0 is some letter in Σ_{I}, $\overline{A}(m, \mathrm{s}) = \{o \in \Sigma_{\mathrm{O}} \mid \exists \mathrm{s}' : (\mathrm{s}, i \cup o, \mathrm{s}') \in \Delta\}$, i.e., the set of outputs allowed by the safety automaton in state s for input i. Further, the probability function is defined as

$$p((m, \mathrm{s}, i), o)(m', \mathrm{s}', i') = \begin{cases} \frac{1}{|\Sigma_I|} & m = \top \wedge m' = m_0 \wedge \mathrm{s}' = \mathrm{s}_0 \\ \frac{1}{|\Sigma_I|} & m \neq \bot \wedge \Omega(m, i) = o \wedge \\ & m' = \delta(m, i) \wedge (\mathrm{s}, i \cup o, \mathrm{s}') \in \Delta \\ \frac{1}{|\Sigma_I|} & m \neq \bot \wedge \Omega(m, i) \neq o \wedge \\ & m' = \bot \wedge (\mathrm{s}, i \cup o, \mathrm{s}') \in \Delta \\ \frac{1}{|\Sigma_I|} & m = \bot \wedge m' = \bot \wedge (\mathrm{s}, i \cup o, \mathrm{s}') \in \Delta \\ 0 & \text{otherwise} \end{cases}$$

As reward function, we define $r((m, \mathrm{s}, i), o) = 1$ if $m \neq \bot$, and 0 otherwise.

The initial state (modeled by $\top$) defines the distribution of the first letter. Afterwards, the machine and the safety automaton move synchronously, depending on the random input letter. If the strategy for an MDP makes a choice that differs from the choice of M, then the first component of the state of the MDP goes to $\bot$ immediatly, signalling that we "left" the machine. On the other hand, if the strategy for an MDP never differs from the choice of M, then the first component of the states of a trace will always be an element of M. Therefore, the reward we chose will provide an average payoff of 1 if the behavior of M is never left, and a payoff of 0 if the behavior differs at least once. In that sense, the reward function "counts" changed and unchanged traces. An optimal, i.e., maximizing strategy for $\mathcal{M}$ therefore changes the minimal number of traces.

Reasons for Repair Failure

In the following we discuss why a repair attempt can fail. The first and simplest reason is that the specification is not realizable. In this case, there is no correct system implementing the specification and therefore also no repair. However, a machine can be unrepairable even with respect to a realizable specification. The existence of a repair is closely related to the question of realizability (Corollary 3.1). Rosner [Ros97] identified two reasons for a specification φ to be unrealizable.

1. **Input-Completeness:** if φ is not input-complete, then φ is not realizable. For instance, consider specification $\mathsf{G}(r)$ requiring that r is always

true. If r is an input to the system, the system cannot choose the value of r and therefore also not guarantee satisfaction of φ.

2. **Causality/Clairvoyance:** certain input-complete specifications can only be implemented by a clairvoyant system, i.e., a system that has knowledge about future inputs (a system that is non-causal). For instance, if the specification requires that the current output is equal to the next input, written as $\mathsf{G}(o \leftrightarrow \mathsf{X}\, i)$, then a correct system needs a look-ahead of size one to produce a correct output.

The following lemma shows that given an input-complete specification φ, input-completeness will not cause our repair algorithm to fail.

Lemma 3.4 (Input-completeness) *If φ is input-complete, then $\big((\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\psi))\downarrow_I \to \mathrm{L}(\mathrm{M})\big) \cap \mathrm{L}(\varphi)$ is input-complete.*

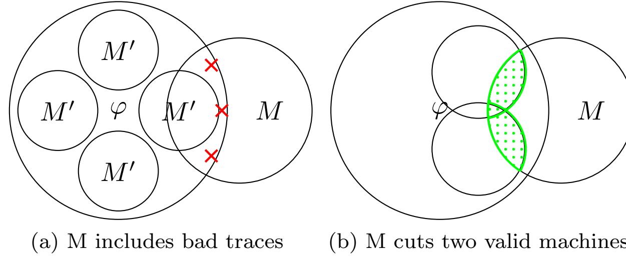
Proof Let $w_I \in {\Sigma_\mathrm{I}}^\omega$. If $w_I \in (\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\psi))\downarrow_I$, then there is a word $w \in \mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\psi)$ such that $w\downarrow_I = w_I$. Therefore we have found a word for w_I. If not, then a word for w_I exists because φ is input complete.

A failure due to missing causality can be split into two cases: the case in which the repair needs finite look-ahead (see Example 3.9 below) and the case in which it needs infinite look-ahead (see Example 3.10 below). The examples show that even if the specification is realizable (meaning implementable by a causal system), the repair might not be implementable by a causal system.

Example 3.9 Consider the realizable specification $\varphi = g \vee \mathsf{X}\, r$ and a machine M that keeps g low all the time, i.e., M satisfies $\mathsf{G}(\neg g)$. If input r is high in the second step, M satisfies φ. An exact repair (according to Definition 3.5) needs to set g to low in the first step if the input in the second step is *high*, because it has to mimic M in this case. On the other hand, it the input in the second step is *low*, g needs to be set to high in the first step. So, any exact repair has to have a look-ahead of at least one, in order to react correctly.

The following example shows a faulty machine and a (realizable) specification for which a correct repair needs infinite look-ahead.

Example 3.10 Consider a machine M with input r and output g that copies the input to the output. Assume we search for a repair such that the modified



(a) M includes bad traces (b) M cuts two valid machines

Figure 3.5: Two reasons for unrepairability

machine satisfies the specification $\varphi = \mathsf{G}\,\mathsf{F}\,g$ requiring that g is high infinitely often. Machine M violates the specification on all input sequences that keep r low from some point onwards, i.e., on all words fulfilling $\mathsf{F}(\mathsf{G}\,r)$. Recall that a repair M′ has to behave like M on all correct inputs. In this example, M′ has to behave like M on all finite inputs, because it does not know whether or not the input word lies in $\mathsf{F}(\mathsf{G}\,r)$ without seeing the word completely, i.e., without infinite look-ahead.

Theorem 3.4 (Possibility of repair) *Assume that we cannot repair machine* M *with respect to a realizable specification φ. Then a repairing machine needs either finite or infinite look-ahead.*

Proof Follows from [Ros97], Corollary 3.1, and Lemma 3.4.

Characterization based on possible machines. Another way to look at a failed repair attempt is from the perspective of possible machines. Recall, in Figure 3.3 we depict a correct repair M′ as a circle covering the set of words in the intersection of M and ψ. In Figure 3.5 we use the same graphical representations to explain two reasons for failure. Figure 3.5a depicts several machines M′ realizing φ. A repair of M has to be one of the machines realizing φ. As observed in [GBJV08], there are words satisfying φ that cannot be produced by any correct machine (depicted as red crosses in Figure 3.5a). E.g, recall the specification $\varphi = g \vee \mathsf{X}(r)$ in Example 3.9. The word in which g is low initially and r high in the second step satisfies φ but will not be produced by any correct (causal) machine because the machine cannot rely on the environment to raise r in the second step. If the machine we are aiming to repair includes such a trace, a repair attempt with $\psi = \varphi$ will fail. In this case,

we can replace φ (or ψ) by the strongest formula that is open-equivalent[3] to φ in order to obtain a solvable repair problem. However, even if φ is replaced by its strongest open-equivalent formula, the repair attempt might fail for the reason depicted in Figure 3.5b. We again depict several machines M′ realizing φ. M shares traces with several of these machines, but no machine covers the whole intersection of φ and M. In other words, an implementing machine would have to share the characteristics of two machines.

3.6 Empirical results

In this section we first describe the repair we synthesized for the traffic light example from Section 3.3. Then, we summarize the results on a set of example we analyzed. All experiments were run on a 2.4GHz Intel(R) Core(TM)2 Duo laptop with 4 GB of RAM.

Traffic Light Example. In the traffic light example, we gave the synthesizer the option to choose from 2^{50} expressions (all possible logical expression over combinations of light colors and signal states). NuSMV returns the expression $(s_2 \wedge s_1 \wedge (l_2 \neq \text{RED})) \vee (\neg s_2 \wedge s_1 \wedge l_2 \neq \text{GREEN})$, which is equivalent to $s_1 \wedge ((s_2 \wedge l_2 \neq \text{RED}) \vee (\neg s_2 \wedge l_2 \neq \text{GREEN}))$ in 0.2 seconds. The repair forbids the first light from turning yellow if the second light is already green. This is not the repair we suggested in Section 3.3 because the synthesizer has freedom to choose between the expressions that satisfy the new notion. Our new approach avoids the obvious but undesired repair of leaving the first light red, irrespective of an arriving car. This is the solution NuSMV provides (within 0.16s) if we use the previous repair notion [JGB05].

Experiment description

In order to empirically test the viability of our approach and to confirm our improved repair suggestions, we applied our approach to several examples. We will first describe the examples we considered, and then we will analyze the results.

Binary Search. This is an implementation of the binary search algorithm, which is famous for the mistakes people make when they implement it. The

[3]Two formulas φ and φ' are open-equivalent if any machine M implementing φ also implements φ' and vice-versa [GBJV08].

original implementation is as follows.

```
1  binary_search(array, needle) {
2    lower := 1
3    upper := len(array);
4    i := (upper − lower)/2;
5    while( array[i] != needle
6          && array[upper] > needle
7          && array[lower] < needle) {
8      if (array[i] < needle) {
9        lower = i;
10     } else {
11       upper = i;
12     }
13     i = (upper − lower)/2;
14     return i;
15 }
```

While the implementation looks reasonable, it contains a bug. This bug is revealed when checking against property

$$\varphi = \texttt{sorted(array)} \land \texttt{needle} \in \texttt{array} \implies \mathsf{F}\,\texttt{array[i] = needle}.$$

It goes into an infinite loop if the array has even length and the needle is in the rightmost element. This is due to a mistake in the assignment to `lower`. We free the assignment to `lower`, and give the synthesizer the option to replace it with `i`, `lower + 1`, `lower - 1`, `upper`, `i - 1` or `i + 1`.

Note the implication in φ. Synthesizing with $\psi = \varphi$ will not allow us to find a solution to this example. To see why, consider the array as input symbol and the returned value as output symbol. We then demand that the synthesizer finds a solution such that

1. The two implementations return the same result when the input was invalid
2. The two implementations return the same result when the input was valid and the broken implementation returns the correct result

It makes more sense to demand that the two implementations return the same result if the input is valid and the original implementation returns the

correct result (Section 3.5). In fact, given this specification, the synthesizer returns the correct result `lower + 1`.

PCI. This example models the PCI Bus protocol, and is taken from the NuSMV distribution. The arbiter has to give bus access to 6 elements, all of which can demand access at any time. The solution is to have 3 smaller 2-input arbiters that decide for priority between a pair of elements each, and one 3-input arbiter that takes the result of the 2-input arbiters as input. All of these can run either in a fixed-priority or in a round-robin mode.

For each bus element, there is a specification demanding that the element eventually can access the bus, if it demands it.

We introduced a bug in the round-robin mode of the 3-input arbiter, which gave access to a element 1 if element 2 requested it (a simple off-by-one error). This meant that, for example, the processor would never receive access to the bus, i.e., one of the specifications is violated. We freed the behavior of the offending round-robin scheduler, thus allowing the synthesizer a lot of choice.

Using the classical synthesis approach, we can guarantee access to the processor by giving it constant access. We have thus repaired the arbiter according to the violated property, but have introduced new bugs, violating other properties.

With our approach, the synthesizer finds the only correct implementation that guarantees access for all bus elements, although the specification defines only about one of them.

Read-Write Lock Example. A read-write lock can be implemented using a semaphore and a lock. In read-write locks there can be arbitrarily many readers to some datastructure. However, if a thread wants to write to the data-structure, then it tries to acquire a write-lock. Once it tries to acquire a write-lock, an implemenation can stop granting access to new readers. It then waits until all readers have left the data-structure, grants the write-lock, and only starts granting read- or write-locks, once the write-lock is released.

Consider the following implementation attempt.

```
1    struct rw_lock {semaphore sem(N_THREADS)};
2
3    write_lock(rw_lock) {
4      for i from 1 to N_THREADS {
5          sem−−;
```

```
6       }
7     }
8
9     read_lock(rw_lock) {
10      sem−−;
11    }
12
13    release_read_lock(rw_lock) {
14      sem++;
15    }
16
17    release_write_lock(rw_lock) {
18      for i from 1 to N_THREADS {
19          sem++;
20      }
21    }
22
23    THREAD_i {
24      while (∗) {
25        if (∗) {
26          read_lock(lock);
27          ....;
28          release_read_lock(lock);
29        } else {
30          write_lock(lock);
31          ....;
32          release_write_lock(lock);
33        }
34      }
35    }
```

Our specification demands that if whatever happens in ... is bounded, then there is no deadlock. The implementation fails this specification. Consider a run in which 2 threads simultaneously try to acquire the write-lock. The system can grant one half of the locks to one thread, the rest of the locks to the other thread. Since no thread has all locks, none can proceed and none of them releases all locks. Therefore the system is in a deadlock.

We now add an additional mutex, because we suspect that we have to

lock the writer locking function, but we do not know how and when exactly. Therefore we modify the implementation as follows.

```
1  struct rw_lock {semaphore sem(N_THREADS)};
2  mutex m;
3
4  write_lock(rw_lock) {
5    if (?) lock(m);
6    for i from 1 to N_THREADS {
7        sem−−;
8      }
9    if (?) unlock(m);
10 }
11
12 read_lock(rw_lock) {
13   if (?) lock(m);
14   sem−−;
15   if (?) unlock(m);
16 }
17
18 release_read_lock(rw_lock) {
19   if (?) lock(m);
20   sem++;
21   if (?) unlock(m);
22 }
23
24 release_write_lock(rw_lock) {
25   if (?) lock(m);
26   for i from 1 to N_THREADS {
27       sem++;
28     }
29   if (?) unlock(m);
30 }
```

We leave the actual condition when to acquire and release the lock free. Our repair method will only activate the condition in function `write_lock`. If any other lock is added, the change modifies runs that were implementing the specification before. Only the traces where two or more threads try to acquire the writer-lock are affected. This is not the case for the original repair

approach, which can enable locks everywhere.

Processor. Here we take a model processor from the VIS distribution. We introduce a bug that shows up when executing the `XOR` opcode, i.e., where the processor has to store the bit-wise xor of two words.

We have several different repair models for this example, varying in the number and structure of candidates. In the first model, we restrict repair to the faulty component. Here, classical synthesis and our new approach provide the same result.

In the other model, we have more freedom in repairs. In this case, classical synthesis will allow changes that are not relevant to the specification, thereby introducing new bugs. Our approach, on the other hand, forbids such repairs and finds the only repair not inducing new bugs.

Results

We report the results in Table 3.1; For each example, we report the number of choices for the synthesizer (Column #Repairs), the time and number of BDD variables to (1) verify the correctness of the repair that we obtain (Column Verification), (2) find a repair with our new approach (Column Repair), and (3) solve the classical repair problem (Column Classical Repair).

In order to synthesize a repair, we followed the approach described in Section 3.4 (Figure 3.4), i.e., we manually added freedom to the model and wrote formula for $\neg\psi$ and equality checking. For all but one of the examples (Processor (1)), the previous approach synthesizes degenerated repairs, while our approach leads to a correct program repair.

AG ($\rightarrow$) is Example 3.5 from Section 3.5. It uses the original specification for ψ, i.e., $\psi = \mathsf{F}(\neg r) \rightarrow \mathsf{G}(g)$. We let the synthesizer choose between all possible boolean combinations of g, r and a memory bit containing the previous value of g. Our approach fails to find a repair. AG (&) is Example 3.6 from Section 3.5 with $\psi = \mathsf{F}(\neg r) \wedge \mathsf{G}(g)$, using the same potential repairs. In this case, a valid repair is found. As in the Assume-Guarantee examples, we have two different choices for ψ in the Binary Search (BS) example. In the case that $\psi = \mathit{sorted} \rightarrow \mathit{correct}$, there is no repair available, while for $\psi = \mathit{sorted} \wedge \mathit{correct}$ we find the correct repair.

The RW-Lock example demonstrates that our approach can also be used to synthesize locks. The synthesizer can choose between 16 options (which

represent release/acquire actions of different locks at different locations). Our notion of repair forbids the acquisition of other locks (the repair we obtained in 27ms with the approach in [JGB05]), because this would imply for example that two threads asking for a read-lock at the same time have to wait for each other. Our notion of repair encodes that runs that were unobstructed before remain unobstructed in the new implementation as well, as long as they do not lead to a dead-lock. Our experiments show that our notion of repair urges the synthesizer to find the intended solution by forcing it to leave correct program runs unchanged. We therefore believe that our approach makes synthesis as a development methodology more practical.

The Processor examples demonstrate what happens in complex models when increasing the amount of freedom in a model. They also show how repairing partial specifications may lead to the introduction of new bugs. In Processor (1), the minimal amount of non-determinism is introduced, i.e., only as much freedom as strictly necessary to repair. Here, the classical approach and our new approach give the same result. In Processor (2), we introduce more freedom, which leads to incorrect repairs with the classical approach. In particular, the fault is in the ALU of the processor, and the degenerated repairs incorrectly execute the `AND` instruction, which is handled correctly in the original model. We allow replacing the faulty and the a correct instruction by either a `XOR, AND, OR, SUB` or `ADD` instruction. Finally, Processor (3) shows that the time necessary for synthesis grows sub-linearly with the number of repair options.

On average, synthesizing a repair takes 2.3 times more time than checking its correctness. Our new approach seems to be one order of magnitude slower than the classical approach. This is expected because finding degenerated repairs is usually much simpler. (This is comparable to finding trivial counter examples.).In order to find correct repairs with the approach of [JGB05], we would need to increase the size of the specification, which will significantly slow down the approach.

3.7 Future work and conclusions

Future Work. Investigation of ways to increase the computational power of a repaired machine seems interesting. Every machine M' repairing M has to

		Verification		Repair		Class. Repair	
	#Rep.	time	#Vars	time	#Vars	time	#Vars
AG (→)	2^{12}	n/a	n/a	0.038	16	0.012	14
AG (&)	2^{12}	0.015	14	0.025	14	0.012	12
BS (→)	5	n/a	n/a	0.78	27	0.1	21
BS (&)	5	0.232	27	0.56	27	0.1	21
RW-Lock	16	0.222	34	0.232	34	0.228	22
Traffic	2^{55}	0.183	68	0.8	68	0.155	63
PCI	27	0.3	56	0.8	56	0.5	53
Processor (1)	2	2m02s	135	2m41s	135	0.5	69
Processor (2)	4	4m28s	138	5m07s	138	0.5	69
Processor (3)	25	5m23s	140	18m05s	140	0.5	71

Table 3.1: Experimental results

behave like M until it concludes that M does not respond to the remaining input word correctly. As shown in Example 3.9, M′ might not know early enough if M will fail or succeed. Therefore, studying repairs with finite look-ahead is an interesting direction. To extend the applicability of our approach to infinite state programs, one could explore suitable program abstraction techniques (cf. [VYY10]). Finally, experiments with model checkers specialized in solving the sequential equivalence checking problem [KMKH01, KSHK07] seem interesting. We believe that such solvers perform well on our problem, because M′ and M have many similar structures.

Conclusion. When fixing programs, we usually fix bugs one by one; at the same time, we try to leave as many parts of the program unchanged as possible. In this chapter, we introduced a new notion of program repair that supports this method. The approach allows an automatic program repair tool to focus on the task at hand instead of having to look at the entire specification. It also facilitates finding repairs for programs with incomplete specifications, as they often show up in real word programs.

4 Quantitative verification and synthesis framework

In which we forge a framework out of the existing techniques of quantitative verification and synthesis, and apply it to save lives.

Résumé

Cette partie représente une implementation de base dans laquelle vérification et synthèse peuvent se produire dans une boucle. Nous soulignons l'importance des deux, c'est-à-dire ensemble de la vérification et de la synthèse dans une base commune. Ce n'est uniquement que dans ce cas-là, que nous pouvons verifier les modèle synthètisés. Nous utilisons Java comme la langue de programmation. Nous décrivons également, comment nous transférons les modèles écrits au Java dans les graphes représentés sous le forme d'une matrice éparse. De plus, nous généralisons et agrandissons le travail précédent sur l'approximation des Pareto courbes des processus de décision markovien.

4.1 Introduction

In this section, we present an implementation that is the basis for a common verification and synthesis framework. Our goals are the following.

- Provide a common framework for the implementation of and research on quantitative verification and synthesis.
- Show via examples that frameworks that do synthesis and verification in a feed-back loop are beneficial.
- Demonstrate that these algorithms can be used for examples not commonly used in formal methods.

- Show that Java can be used as a modeling language.

A common framework for verification and synthesis. Existing frameworks for quantitative verification or synthesis (for example PRSIM and Acacia+) focus on doing either one or the other, but seldom on both. We have a different goal for this framework. We want to support the analysis of controllers that we synthesize. The benefits of using the controller that we synthesize as input to the verification process are as follows.

- We can check the robustness of the controller against modified parameters. This is supposed to harden the synthesis process against model parameter errors or inaccuracies.
- We can check the controller in environments different from the one in which it was created. This can be used for example when synthesizing the controller with one obstacle, and then checking how it performs with two.
- We can replace parts of the previously probabilistic environment by an antagonistic player and check the performance of the synthesized controller. This can be used for a collision avoidance system that we synthesize with probabilistically behaving obstacles. The probabilistic behavior can be replaced by antagonistic behavior to check worst case performance.
- When using abstraction we can check the performance of the controller (which was synthesized in the abstracted model) in a high-detail model.

The architecture of our framework is summarized in Figure 4.1. The synthesis process consists of abstracting a concrete model, and using a synthesis technique on the abstract model. We then translate the controller, which was built on the abstract model, into the original, concrete model. This concrete controller can then be deployed.

We pursue two strategies to verify and validate the synthesized controller. First, we perform probabilistic model checking on an abstract model (not necessarily the one used for synthesis). Second, we perform stochastic model checking on the concrete model.

In the rest of this chapter, we will describe the design of the implementation in Section 4.2 in terms of its interfaces. Then we show how these interfaces can be used to implement value iteration algorithms.

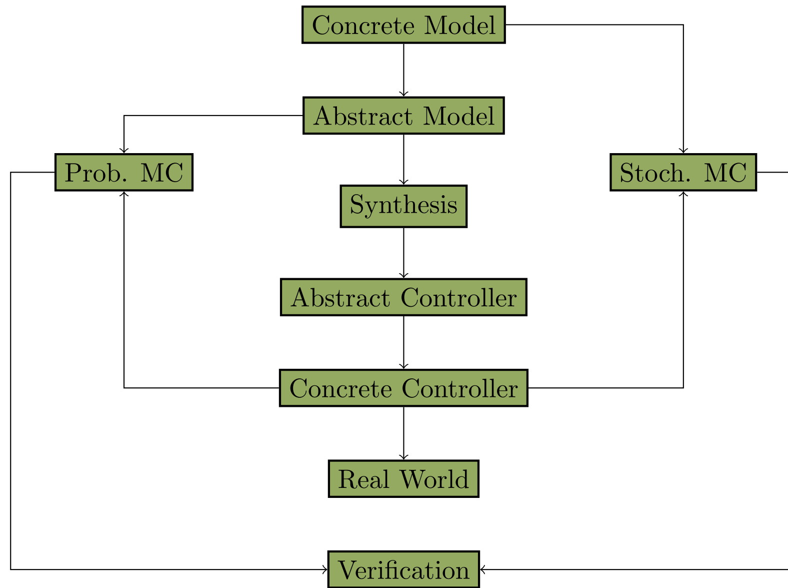


Figure 4.1: Framework Overview

In Section 4.3 we describe necessary preliminaries for Section 4.4, where we show one way to implement the interfaces with using Java as a modeling language. We use a Java EDSL (embedded domain specific language) to describe models in a continuous probabilistic environment. We will use the techniques described in the preliminaries to abstract the continuous models. In Section 4.5, we will describe a general algorithm to approximate Pareto curves of MDPs. Then, in Section 4.6, we will turn towards case studies. We will study emergency braking systems for cars, adaptive cruise control systems and get a car out of a ditch. The next chapter contains a more substantial case study of the currently proposed next generation Airborne Collision Avoidance System (ACAS X).

4.2 Implementation description

When designing this implementation, we had the following goals in mind.

- ACAS X had to fit into this framework
- Easy extensibility by researchers
- Multiple possible input formats for models

- Multithreaded implementation
- Storage and later reuse of synthesized strategies
- Memory efficiency

We defined a small set of interfaces to make the implementation easy to extend by *researchers*. Interface `Enumerator`, `EnumeratorFactory` and `Model` in Figure 4.2 define the interfaces a researcher has to implement if he wants to add a modeling language. They also define the interfaces that are used by all algorithms. So if a researcher wants to add capabilities to the framework, then these interfaces tell him what methods all models provide.

The framework also provides two modeling languages that implement `Model` (see Figure 4.2). A *user* of the framework uses these to model systems he wants to analyze or for which he wants to find strategies. Class `MRMCModel` can read MRMC[KZH$^+$11] models. class `HybridModel` provides a novel modeling language based on a Java *embedded domain specific language* (*EDSL*). We describe it and how we implement the `Model` interface in Section 4.4. Finally, class `CachedModel` can cache certain operations of a class implementing `Model`. It can be used to speed up `HybridModel` models by trading memory consumption for runtime.

In the rest of this section, we will describe the interfaces and how they can be used to implement various common algorithms. We will then turn to the new EDSL in Section 4.4 describe its semantics and how we use it to implement the interfaces. We will explore how we can use its unique features to easily validate the performance of strategies under different model assumptions. We further define the *transient matrix format* which allows us to avoid storing the transition function of an MDP in memory. We study the trade-off of this format in comparison to the sparse matrix format. To have the best of both worlds, our implementation also allows us to turn each transient matrix into a sparse matrix (via `CachedModel`).

Interfaces and algorithms

We have three interfaces at the core of all algorithms (see Figure 4.2 for two of these). First, the `Model` interface describes an MDP and declares a few further operations on it. Second, the `Enumerator` and `EnumeratorFactory`

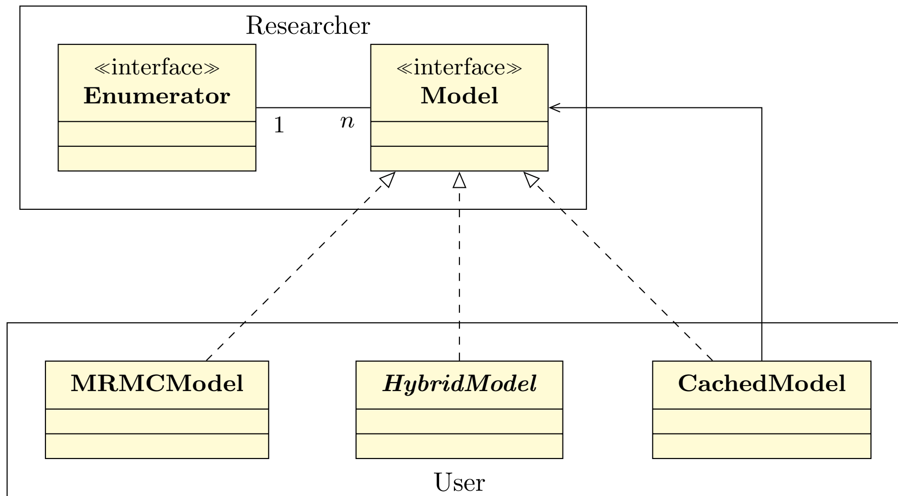


Figure 4.2: Class diagram of the framework.

interfaces are used to iterate in-place over a subset of the states of the model, (i.e,. by changing an instance of `Model`). Note that this is opposed to the usual `Iterator` interface of Java, which is expected to return a new instance on every call to `Iterator.next`, thereby creating a lot of work for the garbage collector. We chose to create a new interface instead of using `Iterator` to make this difference explicit.

These interfaces are of use to a researcher who wants to implement his own modeling, abstraction or algorithms. The framework does not require that a user knows these details.

The `Model` interface. An implementation of the `Model` interface (Figure 4.3) defines an MDP. A `Model` instance has the following tasks.

1. Describe the size of the model and the available actions.

2. Describe the rewards for each state and action.

3. Provide an enumerator over all states of the model.

4. Provide probabilities of next states and expected values.

The method signatures are designed such that a `Model` instance always contains an implicit state upon which its methods depend. For example, method `void fillDistribution(int a, Distribution<T> d)` in Line 36 defines the probabilistic transition function $p : M \times \overline{A} \to \mathcal{D}(M)$ of an MDP $\mathcal{M} =$

$(M, m_0, A, \overline{A}, p)$. Note that p gets a state and an action as parameters, while `fillDistribution` gets an action number and a collection of type `Distribution` to fill. The outcome of `fillDistribution` therefore depends on the state of the `Model` instance.

The interface requires no method to force an instance to take on a specific state. The algorithms we implemented never depend on specific states, but iterate over all states. Methods `next` in Line 18, `done` in Line 21 and `reset` in Line 24 together provide an enumeration interface that modifies the state of a `Model` instance. It is guaranteed that the sequence `reset(); while(!done()) next();` enumerates all possible states.

Each `Model` instance knows how many states there are (`nStates` in Line 3), how many actions are available to its implicit state (`nActions` in Line 6), and what happens if such an action is taken (`fillDistribution` in Line 36, `transitionProb` in Line 30 and `expectedValue` in Line 39 and Line 42).

Method `rewards` in Line 15 describes what rewards a state receives for an action. Since we may want to deal with more than one reward at a time, the reward functions return arrays instead of single values. The implementation promises to not modify the returned array, and also not to rely on the immutability of the returned array between calls to `rewards`. In addition to `rewards`, there is method `initialRewards` in Line 9, which has two purposes: on the one hand, it is used by some algorithm to get the reward of final states (i.e., states which have only self-loops); on the other hand, it is used by some algorithms as an initial guess on the final result of the algorithm (a good initial guess may speed up some algorithms immensely). Method `isFinal` in Line 27 indicates if a state is a final state.

Finally, each model has to define a perfect hashing method `index` in Line 46 that maps its implicit state to a number between zero and the number of states, and a method `get` in Line 49 to return a copy of the current state.

Enumeration. We define interfaces for enumerators and sub-enumerators to support parallelization.

Each enumerator is associated with an object that it modifies. Figure 4.4 presents the enumerator interface. Method `reset` resets the associated object to the start of the enumeration, while `next` modifies it to advance the enumeration by one step. No more calls to `next` are allowed when `done` returns `true`. Finally, method `set` sets another object to the same state as its associated

```
1  public interface Model<T extends Model<T>> {
2          /** @return Number of states in this model */
3          public int nStates();
4
5          /** @return Number of action available to current model */
6          public int nActions();
7
8          /** @return Initial rewards for current state */
9          public double[] initialRewards();
10
11         /**
12          * @param a action number − must be between 0 and {@link #nActions()} − 1
13          * @return Rewards returned when taking action a
14          */
15          public double[] rewards(int a);
16
17         /** Advance model's iteration */
18         public void next();
19
20         /** @return true if iteration is done */
21         public boolean done();
22
23         /** Reset iteration over model */
24         public void reset();
25
26         /** @return Is the current model a final state? */
27         public boolean isFinal();
28
29         /** Probability of transitioning to {@code state} */
30         public double transitionProb(int a, T state);
31
32         /**
33          * Fill a distribution for the given action.
34          * The method is expected not to call Distribution#reset().
35          */
36          public void fillDistribution(int a, Distribution<T> d);
37
38         /** @return expected value over @{code v} */
39         public double expectedValue(int a, double v[]);
40
41         /** Fill expected value of @{code v[i]} into @{code into [i]} */
42         public void expectedValue(int a, double v[][], double[] into);
43
44
45         /** @return index of current state */
46         public int index();
47
48         /** @return A new copy of the current model */
49         public T get();
50 }
```

Figure 4.3: Interface describing a model state

```
1 public interface Enumerator<T> {
2   public boolean done();
3   public void next();
4   public void reset();
5   public void set(T target);
6 }
```

Figure 4.4: The enumerator interface

```
1 public interface EnumFactory<T> {
2   Enumerator<T> get(T m);
3 }
```

Figure 4.5: The enumerator factory interface

object.

The enumerator factory (Figure 4.5) returns a new enumerator associated with an object. This allows easy implementation of parallel algorithms. Since almost all algorithms iterate over all states, we can implement an enumerator factory that successively returns enumerators that enumerate only parts of the state space. Each thread of a parallel implementation then accepts sub-enumerators and works on the part of the state space enumerated by the enumerator.

Supported algorithms

We have implemented algorithms for the following objectives:

- Total sum
- Discounted reward
- PCTL
- Bayesian Model checking

We will, by way of the discounted sum accumulation function, show how the above interface can be used to implement a value iteration algorithm. This will clarify which method corresponds to which concept. It will become clear how we can implement many of the algorithms in literature for each possible objective.

Value iteration is usually used to find the optimal strategy for the discounted sum accumulation function. The high-level description of the algorithm is displayed in Algorithm 4.1. We assign the vector of initial rewards to v in the loop starting with Line 4. Note how we use the combination of `reset()`, `next()` and `done()` to iterate over all states, and how `index()` is used to

```
  Input: Model m, discount double d
  Output: v
1 double v[] ← new double[m.nStates()];
2 double v2[] ← new double[m.nStates()];
3 m.reset();
4 while !m.done() do
5 |  v[m.index()] ← m.initialRewards();
6 |  m.next();
7 end
8 m.reset();
9 repeat
10 |  repeat
11 |  |  double max = Double.NEGATIVE_INFINITY;
12 |  |  foreach 0 <= a < m.nActions() do
13 |  |  |  double t ← m.reward(a) + d*m.expectedValue(a, v);
14 |  |  |  max ← Math.max(t, max);
15 |  |  end
16 |  |  v'[m.index()] ← max;
17 |  until !m.done();
18 |  v, v' ← v', v;
19 until |v' − v| < ε;
```

Algorithm 4.1: Discounted sum value iteration algorithm

index an array of values. We then calculate the optimal action for each state in the loop in Line 9. This part is easily parallelizable using the enumerator interfaces. We repeat this until the change of rewards between two iterations is small enough (Line 19).

Implementation via sparse matrices

One obvious implementation of these interfaces is based on sparse matrices, which is the same implementation MRMC [KZH+11] uses. We have implemented this and are in fact able to read MRMC models and perform model checking on them.

4.3 Discretization of spaces and distributions

In the next section, we will build an EDSL for defining probabilistic models with possibly continuous state space and normal distributions. To handle them with the MDP backend of this framework, we need means to turn continuous distributions into discrete distributions, and continuous state spaces into discrete state spaces. We chose already existing techniques compatible with the

ACAS X report[KC11] for this. This section will present and explain these techniques.

Sigma point sampling

Sigma point sampling [JU04] is a technique to approximate a continuous probability distribution by a deterministically chosen finite set of *sigma points*. In the following, given a matrix $\mathbf{A}$, $\sqrt{\mathbf{A}}$ denotes the square root matrix of $\mathbf{A}$.

Definition 4.1 (Sigma point sampling [JU04]) Given a Gaussian probability distribution $N(\overline{x}, \mathbf{R})$ with mean $\overline{x} \in \mathbb{R}^n$ and covariance matrix $\mathbf{R} \in \mathbb{R}^{n \times n}$ (i.e., $\mathbf{R}_{i,j}$ is the covariance between elements i and j of the random vector), and a tuning parameter $\kappa \in \mathbb{Z}$, we define the sigma points $\sigma_i \in \mathbb{R}^n$ and their weights $w_i \in \mathbb{R}$ for $0 \leq i \leq 2n$ as follows.

$$
\begin{aligned}
& w_0 = \kappa/(n+\kappa) \\
& \sigma_0 = \overline{x} \\
& \sigma_i = \overline{x} + \left(\sqrt{(n+\kappa)\mathbf{R}}\right)_i && \forall 1 \leq i \leq n \\
& \sigma_i = \overline{x} - \left(\sqrt{(n+\kappa)\mathbf{R}}\right)_i && \forall n+1 \leq i \leq 2n \\
& w_i = 1/(2(n+\kappa)) && \forall 1 \leq i \leq 2n
\end{aligned}
$$

Ideally, κ should be chosen such that $n + \kappa = 3$ [BH08].

Example 4.1 The sigma points of a single normally distributed random variable X with mean μ and standard deviation σ are μ, $\mu - \sqrt{3}\sigma$ and $\mu + \sqrt{3}\sigma$.

For two independent random variables X_1 and X_2 sampled from $N((\mu_1, \mu_2), \mathbf{R})$, with $\mathbf{R}_{1,1} = \sigma_1^2$ and $\mathbf{R}_{2,2} = \sigma_2^2$ (and $\mathbf{R}_{1,2} = \mathbf{R}_{2,1} = 0$ because of the independence), and with $\kappa = 3$, the sigma points are $(\mu_1, \mu_2), (\mu_1 - \sqrt{3}\sigma_1, \mu_2), (\mu_1 + \sqrt{3}\sigma_1, \mu_2), (\mu_1, \mu_2 - \sqrt{3}\sigma_2)$ and $(\mu_1, \mu_2 + \sqrt{3}\sigma_2)$.

The samples are chosen such that their mean and standard deviation are equal to that of the source distribution.

Lemma 4.1 (Properties of sigma points [JU04]) *Let $N(\overline{x}, \mathbf{R})$ be an n-dimensional Gaussian distribution, and let σ_i and w_i be sigma distribution derived as described in Definition 4.1. Then the mean mean of the sigma distribution is $\overline{x}$ and its covariance matrix of is $\mathbf{R}$.*

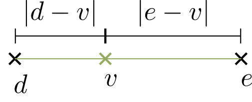


Figure 4.6: Example for interpolation of discretization in one dimension. Points d and e are discretization points, while point v is a continuous point. We have $\iota(d|v) = 1-|d-v|/\Delta$ and $\iota(e|v) = 1-|e-v|/\Delta$.

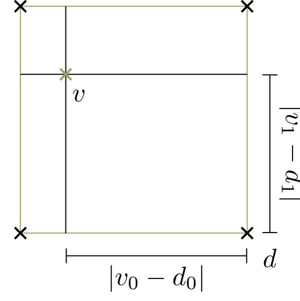


Figure 4.7: Example for interpolation of discretization. The four black crosses mark discretization points, while the green cross (v) marks a continuous point. The probability of d (the discretization point in the lower right corner) is $\iota(d \mid v) = (1-|d_0-v_0|)(1-|d_1-v_1|)$.

Linear interpolation and discretized state space

We discretize the continuous state space by picking points at fixed intervals.

Definition 4.2 (Discretized state space) We discretize a bounded continuous state space $\mathbf{C} \subseteq \mathbb{R}^n$ with bounds $[l_i, u_i]$ in dimension i by finite set of discrete points $\mathbf{D} = \mathbf{D}_1 \times \mathbf{D}_2 \times \ldots \times \mathbf{D}_n$, where the $\mathbf{D}_i$ describe a regular discretization of $\mathbf{C}$ with distances Δ_i, i.e., $\mathbf{D}_i = \{l_i, l_i + \Delta_i, l_i + 2\Delta_i, \ldots, u_i\}$. Note that $\mathbf{D}_i$ contains $(u_i - l_i)/\Delta_i$ points.

Given a continuous point $v \in \mathbf{C}$, we define a probability distribution over $\mathbf{D}$ such that a discrete point close to v has a higher probability than a point further away. If discrete point and continuous point are the same, then the probability of the discrete point should be one, and that of all other points should be zero. To that end, we will use *linear interpolation*.

Definition 4.3 (Linear interpolation) Given continuous state space $\mathbf{C}$ and discretized state space $\mathbf{D} \subseteq \mathbf{C}$, we define $\iota : \mathbf{C} \times \mathbf{D} \to [0, 1]$ as

$$\iota(v, d) = \begin{cases} \Pi_{1 \leq i \leq n}(1 - |v_i - d_i|/\Delta_i) & \text{if } \forall 0 \leq i \leq n : |v_i - d_i| < \Delta_i \\ 0 & \text{otherwise} \end{cases}$$

Instead of $\iota(v, d)$ we sometimes write $\iota(d \mid v)$.

Example 4.2 Figure 4.6 captures the idea for one dimension. Points d and e are discretization points, while point v is a continuous point. Since v is closer to d than to e, the probability of d should be higher than that of e. In addition, the probabilities of d and e have to add up to one. According to the previous definition, $\iota(d \mid v) = 1 - |d - v|/|d - e|$, i.e., one minus the distance between d and v divided by their maximum possible distance.

For an example in two dimensions, consider Figure 4.7. Here we have four different discretization points (the black crosses) and one continuous point (v). To calculate the probability of point d (lower right corner) given v, we measure the relative distance in both the first and the second component in the state space (i.e., $|v_0 - d_0|/\Delta_0$ and $|v_1 - d_1|/\Delta_1$). After measuring the relative distances, we have the probabilities of the two dimensions, i.e., $p_0 = 1 - |v_0 - d_0|/\Delta_0$ and $p_1 = 1 - |v_1 - d_1|/\Delta_1$. To get the probabilities of points v and d, we multiply the probabilities of the separate dimensions, i.e., $\iota(d \mid v) = p_0 \cdot p_1$.

That this function indeed defines a probability distribution is trivial.

Lemma 4.2 $\iota : \mathbf{C} \to \mathbf{D}$ *is a probability distribution.*

To ease notation later, we will now lift probability distribution from $\mathbf{C}$ to probability distributions over $\mathbf{D}$, for distributions with finite *support*

Definition 4.4 (Support) Let $p \in \mathcal{D}(A)$ be probability distribution over A. Then $\mathsf{Supp}(p) = \{a \in A \mid p(a) > 0\}$ is the *support* of p. We say that p has *finite support* if $\mathsf{Supp}(p)$ is finite.

Definition 4.5 (Interpolated probability distribution) Let $p : \mathbf{C} \to [0,1]$ be a probability distribution with finite support. The interpolated probability distribution $\mathrm{I}(p) : \mathbf{D} \to [0,1]$ is then defined as

$$\mathrm{I}(p)(d) = \sum_{v \in \mathsf{Supp}(p)} \iota(d \mid v) p(v).$$

Example 4.3 Consider a car moving in continuous space, as in Figure 4.8, and a discretization of the state space depicted by the crosses at the corners of the grid. In this figure, the car will move in one step along one of the arrows, to either point A or point B. In both cases, the car will end up between discretization points.

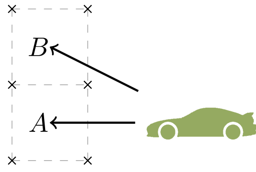


Figure 4.8: Illustration of car movement before interpolation.

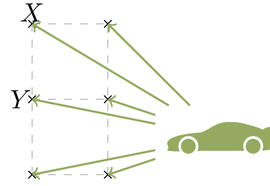


Figure 4.9: Illustration of car movement after interpolation.

Figure 4.9 shows the interpolated probability distribution. After interpolation, the car moves in one step to one of the discretization points. For example, the probability of moving to point X is the probability of moving to B times $\iota(X|B)$. The probability of moving to point Y is the probability of moving to A or B times the appropriate interpolation, i.e., $p(A) \cdot \iota(Y|A) + p(B) \cdot \iota(Y|B)$.

4.4 Specifying models in Java

In this section we present our EDSL for specifying continuous space models. It is based on implementing methods in the abstract class `HybridModel`, which we will describe later. We show how we turn continuous models into discrete state space models, and how we implement the `Model` interface automatically.

We will first present an adaptive cruise control (ACC) model as an example, point out the various parts that are used as sources for discretization and sigma point sampling, and we will then discuss the implementations that turn these sources and models into MDPs. We will then shortly discuss how we can interpolate the strategies that we generate from the discretized models and how we can use the original models for statistical model checking.

Figure 4.10 contains the code of the ACC model. A few features here are noteworthy. Continuous variables that are part of the model are annotated with `@CVAR`. Discrete variables (not present in the example) are annotated analogously with `@DVAR`. A variable should be part of a model if (1) it influences the probabilistic transition function and (2) it changes over time. In our case, variables `distance` and `velocity` are part of the model, while `desiredDistance` and `ticks` are not part of the model (because they are constant). Method `void next(double acceleration, ACC target)` defines a distribution over the next states, given the current state. We first sample a random acceleration for the other car with mean zero and an arbitrarily chosen standard deviation

```
1  public class ACC extends HybridModel<ACC> {
2      // Distance from car in front
3      @CVAR(min=0, max=100)
4      public double distance;
5
6      // Relative velocity
7      @CVAR(min=−14, max=14)
8      public double velocity;
9
10     // Distance we desire
11     public double desiredDistance = 50;
12
13     // Updates per second
14     public int ticks = 10;
15
16     @Override
17     public void next(double acceleration, ACC target) {
18         double random_acceleration = normal.sample(0, 4.0);
19         double nextVelocity =
20             velocity + (acceleration + random_acceleration) / ticks;
21         double nextDistance = distance −
22             (0.5 ∗ velocity + 0.5 ∗ nextVelocity) / ticks;
23         target.velocity = nextVelocity;
24         target.distance = nextDistance;
25     }
26
27     @Override
28     public double[] rewards(double acceleration) {
29         double[] rewards = new double[2];
30         rewards[0] = −Math.abs(desiredDistance − distance);
31         rewards[1] = −acceleration∗acceleration;
32         return rewards;
33     }
34 }
```

Figure 4.10: ACC example code

4 m/s^2. Next, we calculate the velocity of the next state, based on the current velocity, the random acceleration and the acceleration we got as input. Lastly, based on old velocity and distance and on the new velocity, we calculate the distance of the next state. Note that our framework supports both loops and branching, although they are not present in this example. Additionally, we define the rewards the controller gets for its decisions in method `double[] rewards(double acceleration)`. In the case of `ACC`, it receives a cost (negative reward) depending on how far the current distance is from the desired distance (`rewards[0]`), and a cost depending on how much it accelerates (`rewards[1]`). Note that these two define exactly the trade-off mentioned

before. We want to minimize both `rewards[0]` and `rewards[1]`, but applying less acceleration will lead to a greater deviation from our desired distance. Vice versa, being stricter about staying close to the desired distance requires more acceleration.

From hybrid to discrete state space

Implementations of class `HybridModel` describe probabilistic hybrid systems.

Definition 4.6 (Probabilistic hybrid system) A Probabilistic hybrid system is defined as $H = (\mathbf{C}, \mathrm{S}, \mathrm{s}_0, \overline{A}, A, p)$, where $\mathbf{C} \subseteq \mathbb{R}^n$ is a continuous state space, S is a finite set of states, $\mathrm{S}_0 \in \mathbf{C} \times \mathrm{S}$ is an initial state, A is a finite set of actions, $\overline{A} : \mathrm{S} \to 2^A$ is the action activation function, and $p : \mathbf{C} \times \mathrm{S} \times A \to \mathcal{D}(\mathbf{C} \times \mathrm{S})$ is the continuous probabilistic transition function.

Our algorithms work on MDPs, and so we abstract probabilistic hybrid systems to Markov decision processes as follows.

Definition 4.7 (Abstraction of hybrid systems) Let $H = (\mathbf{C}, \mathrm{S}, \mathrm{s}_0, \overline{A}, A, p)$ be a probabilistic hybrid system and let $\mathbf{D} \subseteq \mathbf{C}$ be a regular discretization of the continuous state space. If $p((v, \mathrm{s}), a)$ defines the probability distribution function over $\mathbf{C} \times \mathrm{S}$ for all $v \in \mathbf{C}$, $\mathrm{s} \in \mathrm{S}$ and $a \in A$, then we define the discretization of H as a Markov decision process $\mathcal{M} = (\mathbf{D} \times \mathrm{S}, \mathrm{s}'_0, \overline{A}, A, p')$ as follows. Let $p'' : p : \mathbf{C} \times \mathrm{S} \times A \to \mathcal{D}(\mathbf{C} \times \mathrm{S})$ be the probability distribution resulting from sigma point sampling of p. Then $p' : \mathbf{D} \times \mathrm{S} \times A \to \mathcal{D}(\mathbf{D} \times \mathrm{S})$ is defined as $p'((v, \mathrm{s}), a, (v', \mathrm{s}')) := \mathrm{I}(p''((v, \mathrm{s}), a))(v', \mathrm{s}')$. s'_0 is defined as the element of $\mathbf{D}$ that contains s_0.

Note that we make no claim to soundness or completeness. The choice of discretization was made with the goal of being compatible to the ACAS X report [KC11]. The framework is of course flexible enough to swap this discretization out for any other that generates MDP (for example that of [Hah12]).

Hybrid model base class

We will now present the base class that is used to implement the EDSL. We will then show how we can use the information provided by the EDSL to automatically implement the abstract of Definition 4.7.

```
1  abstract public class HybridModel<T extends HybridModel<T>>
2                              implements Model<T>, Enumerator<T> {
3          /** Samples a new state from {@code this} state. */
4          public abstract void next(int action, T target);
5
6          /** @return Number of actions available in current state. */
7          public abstract int nActions();
8
9          /** @return Initial rewards for current state */
10         public abstract double[] initialRewards();
11
12         /**
13          * @param a Action for which we need a reward
14          * @return Reward for action {@code a}
15          */
16          public abstract double[] rewards(int a);
17
18          /** @param Model configuration to use */
19          public void setModelConfig(ModelConfig config) {
20            // ...
21         }
22
23         /** Reset sample iteration */
24         public boolean resetSample() { ... }
25
26         /** Increase sample iteration */
27         public boolean nextSample() { ... }
28 }
```

Figure 4.11: Hybrid model base class used for EDSL specifications.

Hybrid models as the one described above inherit from a common base class `HybridModel`, parts of which we display in Figure 4.11. Note that `HybridModel` implements the `Model` interface. Methods `nActions`, `initialRewards`, and `rewards` implement the `Model` interface (Figure 4.3). Only methods `next`, `resetSample` and `nextSample` are new. `next` is the continuous stepping function, which fills another instance with a sampled next state, as in Figure 4.10. `resetSample` and `nextSample` will be described below.

Example 4.4 (Class `Hybrid`) By way of an example, we will use class `Hybrid` in Figure 4.12 to explain how `HybridModel` implements most of the interface of `Model`.

This class has two model variables: a continuous variable `x` and a discrete variable `counter`. It uses a discrete probability distribution `coin` over Boolean variables (Line 2). We define `coin` to return `true` with probability 0.3, and

```
1  public class Hybrid extends HybridModel<Hybrid> {
2          private DiscreteDistribution<Boolean> coin = new DiscreteDistribution<Boolean>();
3
4          @CVAR(min=0, max=10)
5          double x = 5;
6
7          @DVAR(min=0, max=1)
8          int counter = 0;
9
10         public Hybrid() {
11                 coin.add(true, 0.3); coin.add(false, 0.7);
12         }
13
14         @Override
15         public void next(int action, Hybrid target) {
16                 boolean flip = coin.sample();
17                 double t;
18                 if (flip && counter == 1) {
19                         t = normal.sample(−1, 1);
20                 } else {
21                         t = normal.sample(1, 1);
22                 }
23                 target.x = Math.min(Math.max(this.x + t, 0), 10);
24                 target.counter = flip ? 1 − counter : 0;
25         }
26         // ...
27 }
```

Figure 4.12: Hybrid example used to explain the inner workings of class `HybridModel`.

`false` with probability 0.7 (see Line 11).

In the stepping function, `Hybrid` first flips the `coin`. If the coin flip this turn and the coin returns `true` and `counter` is one, then the next value of `x` will be samples from $N(-1,1)$, otherwise from $N(1,1)$. Finally, if the flipped coin returned `false`, then the next value of `counter` will be 0, otherwise it will be one if the old value of `counter` was zero, and zero otherwise.

Automatic discretization, enumeration and perfect hashing

Class `HybridModel` describes a continuous (and therefore infinite) state space model. To implement `Model`, we have to discretize the state space and the continuous probability distributions. In this section we describe how we implement sigma point sampling and linear interpolation Section 4.3 to this end.

As noted before, state space variables are annotated with `@DVAR` and `@CVAR`

```
1  public void fillDistribution(int a, Distribution<T> d) {
2          T nextContinuous = get();
3
4          resetSampling();
5          do {
6                  next(a, nextContinuous);
7                  interpolate(nextContinuous, getProb(), d);
8          } while (nextSample());
9  }
```

Figure 4.13: Implementation of `fillDistribution`, part of class `HybridModel`

respectively. In addition, a model is given a model configuration that defines the number of discretization points for each variable annotated with `@CVAR`. Following Definition 4.2, a variable annotated with `@CVAR(min=`l`, max=`u`)` and configured with n discretization points is discretized with distance $\Delta = (l - u)/(n-1)$. To this end, class `HybridModel` gathers information about the annotated fields: for each field, the lower and upper bounds are retrieved via reflection upon the first time an instance of the class is created. Based on this information, we can implement the enumeration methods and `index` automatically. It also allows the implementation of partial enumerators that only enumerate over a subset of the available fields. We use this to allow parallelization. In addition to enumeration, this information also allows the implementation of interpolation as in Definition 4.5.

We will show how to implement method `fillDistribution` based on the information above. Recall that fillDistribution models the probabilistic transition function. Methods `expectedValue` and `transitionProb` can be implemented analogously. The implementation is based on one loop (see Figure 4.13), which makes use of instrumented functions to generate all possible samples from the sampling function, and of an interpolation function, both of which we will discuss next. As described before, method `next` fills a given state with a randomly sampled state. We instrumentalize `next` as described below to enumerate, in conjunction with `resetSampling` and `nextSample`, all possible samples. In addition, the instrumentalized version keeps track of the probability of the sample it most recently returned. This probability is accessible via method `getProb()`. After having filled `nextContinuous` with a continuous next state, we interpolate this state using linear interpolation, and fill distribution `d` with the result.

Example 4.5 For the class in Figure 4.12, the coin flip can either return `true` or `false`. Depending on the outcome of the flip, and on `counter`, the discretized gaussian distribution of `x` has either outcomes $\{-1, -1-\sqrt{3}, -1+\sqrt{3}\}$ or $\{1, 1-\sqrt{3}, 1+\sqrt{3}\}$. That is, if `counter` is `1`, then the loop in Figure 4.13 will generate, one after the other, all combinations of $\{\texttt{1}\} \times \{-1, -1-\sqrt{3}, -1+\sqrt{3}\} \cup \{\texttt{0}\} \times \{1, 1-\sqrt{3}, 1+\sqrt{3}\}$

Instrumented `next`. We will now describe how we modify an implementation of the continuous `next` method that the user provides. There are two intended effects. First, we want to replace continuous sampling statements (i.e., calls to `normal.sample`) by sigma point sampling (Definition 4.1). Second, we want to be able to produce the whole discretized probability distribution by a loop like the one in Figure 4.13.

We replace sampling calls by versions that return values deterministically. What values are returned depends on an internal sampling stack explained later. This is implemented such that repeated calls to `next` will always return the same values, unless either `nextSample` or `resetSampling` are called.

Sampling calls are either calls to instances of class `DiscreteDistribution` which allows user-defined discrete distributions, or from calls to `normal.sample(mean, sd)`, where `normal` is a field present in each instance of `HybridModel`. We instrument calls to these methods in `next` by replacing them with hidden versions that have different semantics and an additional parameter.

In addition to their original parameters, the hidden version of `normal.sample` and `DiscreteDistribution.sample` receive an ID (based on the position of the sampling statement and how often it was called in an invocation of `next`). We use this ID to get a handle on the sampling state, which we will explain next.

Instead of returning random samples, these instrumentalized versions maintain a *sampling stack*, and return samples from this stack. The sampling stack consists of a stack of frames, where each frame is a stack of sampling states. A sampling state is a list of samples and probabilities.

In the beginning (i.e., before a call to `next`), the sampling stack is empty. When an instrumented sampling method is invoked, it uses its ID to request its frame from the sampling stack. When it finds that no such frame exists, a new frame is created with all possible sampling points of the sampling statement. For example, for a discrete probability distribution that can take on values `true`

and `false` with probability 0.5, will fill an empty frame with [0.5 : `false`, 0.5 : `true`]. A continuous probability distribution, on the other hand, will fill the frame with appropriate sigma point samples. After possibly filling the sampling frame, a sampling statement returns the topmost element from its frame and multiplies an instance field with the probability of that element.

Note that repeated calls to the instrumentalized `next` method will create the same state each time (because the sampling state is not changed). To advance to the next state, method `nextSample` (Line 8) is used. This method will remove the topmost element of the topmost sampling frame. If this frame should be empty after removal, then it removes the frame completely and removes the topmost element of the now newly topmost frame, until either it finds a frame that is not empty after removal or until no frame remains. If there are still frames left after this operation, then `nextSample` returns `true` to indicate that there are samples remaining, otherwise it returns `false`.

Example 4.6 Let us take the code in Figure 4.12 as an example again. After a call to `resetSampling`, the sampling stack is empty, i.e., `[]`. We then run the instrumented `next` function. On the call to `coin.sample`, its implementation will create a sampling frame for this sampling statement, i.e., the sampling stack is now `[[true, false]]`. Then `coin.sample` returns `true` and execution of `next` continues. The next instrumented statement is the sampling statement in Line 19. Since this sampling statement has no sampling frame yet, it will create one, i.e., the sampling state is now `[[true, false],` $[-1, -1-\sqrt{3}, -1+\sqrt{3}]$`]]`. `normal.sample` will return the topmost of its elements, i.e., -1. Then the run of `next` continues until the method is finished.

Next follows a call to `nextSample`, which modifies the sampling stack by popping the first element of the topmost frame. After this operation, the sampling stack is now `[[true, false],` $[-1-\sqrt{3}, -1+\sqrt{3}]$`]]`. Therefore, the next returned sample is going to be `(true,` $-1-\sqrt{3}$`)`. After a call to `nextSample`, the sampling stack will be `[[true, false],` $[-1+\sqrt{3}]$`]]`, and the returned sample will be `(true,` $-1+\sqrt{3}$`)`. Now, the next call to `nextSample` will pop $-1+\sqrt{3}$ of the topmost frame. Since the frame is now empty, it will be removed completely. The sample stack is now `[[true, false]]`. Since we just removed a sample frame, we will also remove the topmost element of the topmost frame, which yields stack `[[false]]`.

On the next call to `next`, `coin.sample` returns `false`. By continuing the

execution of `next`, the next sampling statement we encounter is in Line 21. `normal.sample` will find that it has no corresponding sampling stack, and therefore create one. After this operation, the sampling state is `[[false],` `[`$1, 1-\sqrt{3}, 1+\sqrt{3}$`]]`. After two further calls to `next`, the sampling state will be `[[false], [`$1+\sqrt{3}$`]]`. A call to `nextSample` pops first the last element of the topmost frame, then the last element of the bottom most frame. After this, the sampling stack is empty again, and sampling therefore completed.

Strategy interpolation

We compute strategies on discretized models. To make these usable on the original (continuous) model, we need means of strategy interpolation. By default, our implementation provides two schemata. The framework is extensible, though, and researchers may implement their own interpolation schemes.

Linear interpolation of discretized values. If the set of actions available in each state is a discretized state space from an originally continuous action set, then we can use linear interpolation to get a continuous strategy.

Definition 4.8 Given a continuous action set A, continuous state space $\mathbf{C}$, discretized action set B and discretized state space $\mathbf{D}$, and strategy $d : \mathbf{D} \to B$, the linearly interpolated strategy $d' : \mathbf{C} \to A$ is defined as

$$d'(c) = \sum_{\mathrm{s} \in \mathbf{D}} \iota(\mathrm{s} \mid c) \cdot d(\mathrm{s})$$

Weighted voting. If the set of actions available is discrete and finite, then we can use a weighted voting scheme. This is the scheme used in the ACAS X case study. Intuitively, we ask each of the surrounding states of a discretized state how good it believes each action to be. We then use linear interpolation to judge how important the opinion of each discretized state is.

Definition 4.9 Let $\mathcal{M} = (M, m_0, A, \overline{A}, p)$ with reward function $r : M \to \mathbb{R}$ be the discretized MDP of a stochastic hybrid automaton. We create a table equivalent to the function

$$T : M \times A \to \mathbb{R}, T(m, a) = r(m, a) + \mathbb{E}_{p(m' \mid m, a)}[\max_{a' \in \overline{A}(s')} T(m', a')]$$

that yields for each pair of state and action of the discretized MDP the immediate received reward associated with the action and the expected reward of

the optimal action in each following state. Based on this, we create a strategy $d : \mathbf{C} \to \overline{A}$ for the continuous state space $\mathbf{C}$ by $d(c) = \arg\max_{a \in \overline{A}(c)} \sum_m \iota(m \mid c) T(m, a)$.

Bayesian statistical model checking

Modeling using Java has one additional advantage. It allows us to implement statistical model checking. There are many flavors of statistical model checking (e.g., [Var85, BHHK03, KNP07]). The one we chose to implement here is called Bayesian statistical model checking [ZPC13], which uses a Bayesian approach to confidence intervals.

This thesis touches only on the intuition behind Bayesian statistical model checking, since it does not contribute to its theory. The contribution is the application of this technique to the case studies in the rest of the thesis.

In Bayesian model checking, we use Bayesian statistics to establish the probability that a randomly selected run of a Markov chain fulfills a property. To that end, we collect finite runs of the Markov chain. Each run serves to update our current belief about the real probability.

Example 4.7 Assume that we want to establish the unknown probability that a possibly biased[1] coin comes up head. Initially, we have a prior belief about the coin's bias. Let us say that we are totally clueless, and so our prior belief is a uniform distribution of the interval $[0, 1]$, i.e., we consider each bias equally likely.

We then flip the coin for the first time. Based on the outcome, our belief will either shift towards 0 or 1. We now flip the coin a second time, and update our belief again, and so on. For details, see [ZPC13].

We continue creating runs until we are content with our belief. What content means depends on the application in general. The approach chosen for this implementation (and presented in [ZPC13]) is that similar to confidence intervals: we keep on generating runs until we are confident enough that the probability lies inside an interval with a certain width.

Example 4.8 Back to the coin flip example, we could keep generating runs until we can believe with 99% probability that the bias lies in an interval $[l, u]$

[1]The bias b of a coin is the probability that it comes up head

with $u - l = 0.01$, i.e., we know the probability up to an error of 0.5%. In general, the smaller the interval and the higher our desired confidence, the more information we need, i.e., the more traces we need to generate. For a coin with bias 0.3, we needed to generate about 50,000 samples, to say that $\mathrm{P}(b \in [0.296, 0.306] |$ traces $) \geq 0.99$.

Trade-offs of EDSL

Compared to a classic sparse matrix implementation, the Java EDSL saves memory by never storing the transition function. It does this by not storing the transition matrix, but the transition function. This is analogous to storing a formula or a table generating a formula. This was crucial in Chapter 5, because the size of the transition function would have exceeded the available memory by far. This is worthwhile whenever we use the probabilistic next function only relatively rarely. Our implementation does allow to trade speed for memory when desired. We have implemented a class that caches the transition function in a sparse matrix, thereby reaching speed comparable to MRMC.

Note further that a user can override any of the functions that are implemented using the scheme described above. In the example in Chapter 5, we use this ability to speed up some of the computations.

4.5 Approximating Pareto curves

In this section we describe how we approximate Pareto curves. We use a new variant of the so-called sandwich algorithms (see, e.g., [RvDdH11]) to approximate Pareto curves. A sandwich algorithm is called thus because it maintains a lower and an upper bound on Pareto curves. Upper bounds, lower bounds, and Pareto curves are convex polygons, which in turn are defined by hyperplanes.

Definition 4.10 (Hyperplane) In $\mathbb{R}^n$, a *Hyperplane* is defined by an $n + 1$ dimensional vector $w = (w_1, w_2, \ldots, w_{n+1}) \in \mathbb{R}^{n+1}$. The hyperplane is then the set of points $\{v \in \mathbb{R}^n \mid v \cdot (w_1, \ldots, w_n) = w_{n+1}\}$. The hyperplane cuts $\mathbb{R}^n$ into two half-spaces. The *lower halfspace* $L \subseteq \mathbb{R}^n$ is defined as $L = \{v \in \mathbb{R}^n \mid v \times (w_1, w_2, \ldots, w_n) \leq w_{n+1}\}$, whereas the *upper halfspace* U is defined as all other points, i.e. $U = \mathbb{R}^n \setminus L$.

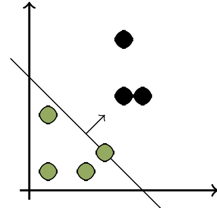


Figure 4.14: Figure demonstrating hyperplanes defining half-spaces. The line crossing both axes denotes a part of a hyperplane, while the space above and below the hyperplane are the upper and lower half-spaces. Green dots are in the lower half-space, while black dots are in the upper half space. The arrow protruding from the hyperplane denotes its normal.

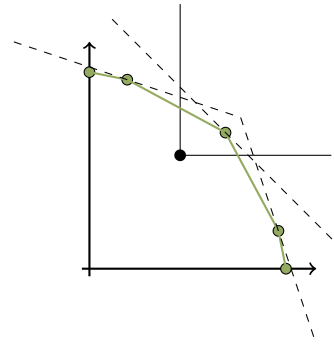


Figure 4.15: Figure demonstrating the sandwich algorithm. The black dot and the lines protruding from it denote the target; green dots denote Pareto optimal points, while the green lines connecting them denote the convex hull of the Pareto optimal points (i.e., the lower bound); the black dashed lines denote the hyperplanes going through the Pareto optimal points (i.e., the upper bound).

Alternatively, a hyperplane can be defined using two n dimensional vectors $v, w \in \mathbb{R}^n$, where the hyperplane is then defined as by vector $(w_1, \ldots, w_n, v \cdot w)$.

Example 4.9 Figure 4.14 demonstrates a hyperplane and its half-spaces in two dimensions. The hyperplane is the line cutting both axes. It is defined by its normal vector w (the arrow protruding from the hyperplane), and its distance from the origin d. That is, the hyperplane is defined as the set of points $\{v \in \mathbb{R}^2 \mid v \cdot w = d\}$. All points $v \in \mathbb{R}^2$ that are below the hyperplane, i.e., for which $v \cdot w \leq d$ holds, define the lower half-space of the hyperplane. Conversely, those points for which $v \cdot w > d$ holds make up the upper half-space. A hyperplane can equivalently be defined by two vectors $v, w \in \mathbb{R}$, where v is a reference point and w is the normal. The hyperplane defined by these two points is then the hyperplane with normal w and distance $v \cdot w$. In Figure 4.14, one such reference point is the green point lying on the hyperplane.

If we intersect several half-spaces and get a bounded space, then we call this space a convex polytope.

Definition 4.11 (Convex polytope) A *convex polytope* $\pentagon(v_1, v_2, \dots, v_m) \subseteq 2^{\mathbb{R}^n}$ is defined as the intersection of the m lower half-spaces defined by vectors $v_i \in \mathbb{R}^{n+1}$, i.e., $\pentagon(v_1, v_2, \dots, v_m) = \bigcup_{i=1}^{m} L_i$, where L_i is the lower half space defined by v_i, if this intersection is bounded.

A “side” of a polytope is called a facet.

Definition 4.12 (Facet) A face of a convex polytope $\pentagon(v_1, v_2, \dots, v_m) \subseteq 2^{\mathbb{R}^n}$ is every subset of $F \subset \pentagon(v_1, v_2, \dots, v_m) \subseteq 2^{\mathbb{R}^n}$ such that there is a hyperplane H such that $F = C \cup \pentagon(v_1, v_2, \dots, v_m) \subseteq 2^{\mathbb{R}^n}$ and such that all of the convex polytope lies in the lower half-space of the hyperplane.

For an n dimensional polytope, the $n-1$ dimensional faces are called facets.

The reward space of an MDP with multiple reward functions is the set of all achievable payoffs, i.e., the set of points in $\mathbb{R}^n$ that can be produced by a strategy.

Definition 4.13 (Reward space of an MDP) Let $\mathcal{M}$ be an MDP, let $r_1, \dots, r_n$ be n reward functions (i.e., $r_i : M \to \mathbb{R}$ forall $1 \leq i \leq n$), and let $\alpha : \mathbb{R}^* \to \mathbb{R}$ be an accumulation function. The *reward space* $\mathsf{rew}(\mathcal{M}, \alpha, (r_1, \dots, r_n))$ is the set of points reachable via any strategy, i.e., $\mathsf{rew}(\mathcal{M}, \alpha, (r_1, \dots, r_n)) = \{(\mathbb{E}_d[\alpha \circ r_1], \dots, \mathbb{E}_d[\alpha \circ r_n]) \in \mathbb{R}^n \mid d : (\mathrm{S} \times A)^* M \to A\}$

A member of the reward space is called Pareto optimal if there is no other member better than it.

Definition 4.14 (Pareto optimal points) A point $p \in \mathsf{rew}(\mathcal{M}, \alpha, r)$ is called *Pareto optimal* if there is no point $p' \in \mathsf{rew}(\mathcal{M}, \alpha, r)$ such that $p' > p$.

The Pareto curve is the set of Pareto optimal points.

Definition 4.15 (Pareto curve) The Pareto curve of $\mathsf{rew}(\mathcal{M}, \alpha, r)$ is the set of Pareto optimal points of the same.

Convex polytopes are interesting to us, because the reward space of MDPs with some accumulation functions are convex polytopes.

Lemma 4.3 ([CMH06]) *The reward space of an MDP with the discounted accumulation function is a convex polytope. Finite memory randomized strategies may be required.*

Lemma 4.4 ([FKN+11]) *The reward space of an MDP with the total sum accumulation function is a convex polytope, if it is bounded. Randomized and finite memory strategies may be required.*

Lemma 4.5 ([BBC+11]) *The reward space of an MDP with the mean accumulation function is a convex polytope. Randomized and finite memory strategies may be required..*

Often we are not interested in the full Pareto curve, but only in a part of it defined by a target point $t \in \mathbb{R}^n$. If we have such a point, then we are only interested in the set of Pareto optimal points p such that $p \geq t$.

Algorithm

Our algorithm works on the total sum accumulation function, the mean payoff accumulation function and the discounted payoff function, because they all fulfill the following property and have convex reward spaces.

Definition 4.16 (Linearizable) We call an accumulation function linearizable if it fulfills the following property.

$$\max_{d:(M\times A)^* M \to A} \mathbb{E}_d[\alpha \circ (r \cdot w)] < p \cdot w \implies p \notin \mathsf{rew}(\mathcal{M}, \alpha, r)$$

Lemma 4.6 *Maximum sum, mean-payoff and discounted sum are linearizable. Furthermore, their reward spaces are convex polygons.*

PROOF See [CMH06, FKP12] for discounted and maximum sum payoff. The linearizability for mean-payoff follows by a similar argument.

Using Definition 4.16 and the convexity of the reward spaces, we can approximate a Pareto curve by maintaining a lower bound, i.e., a subset of the space that definitively is a subset of the real reward space, and an upper bound, i.e., a set definitively containing all of the reward space.

If we have already found Pareto optimal points $X = \{p_1, \dots, p_n\}$, then we know that every convex combination of elements of X is achievable. Thus the convex hull of a set of points X forms a lower bound.

The upper bound is a result of the linearizability of the accumulation functions. To find a strategy and its payoff, we will linearize the reward vector $r \in (\mathrm{S} \to \mathbb{R})^n$ with a weight vector w and optimize the $\mathbb{E}[\alpha \circ (w \cdot r)]$. Because

```
1  X ← Initial();
2  while ¬Sufficient(X) do
3  |   F ← maxFacet(X);
4  |   w ← weightOf(F);
5  |   q ← findPoint(w);
6  |   if q ∈ F then
7  |   |   set error of F to 0;
8  |   else
9  |   |   X ← X ∪ {(q, w)};
10 |   end
11 end
```

Algorithm 4.2: General sandwich algorithm for Pareto curves

of the linearizability of the accumulation function, we then know that a point p cannot lie in the reward space if we have found a point q with weight w such that $q \cdot w < p \cdot w$.

Refer to Figure 4.15 for a graphical explanation of the bounds. In this figure we have already generated 5 points (in green). The convex hull connecting these points forms the lower bound of the reward space, i.e., all points inside this convex hull are part of the reward space. Note the dashed lines going through the points. A line going through a point q that was generated using weight w is the hyperplane with normal w and reference point q. Because of the linearizability, these lines form an upper bound on the reward space: the space above the lines (i.e., the union of all upper half-spaces of all hyperplanes) does not intersect with the reward space. Therefore, the whole reward space has to lie in the intersection of the lower half-spaces of the hyperplanes. So, In our picture, any remaining Pareto optimal points have to be between the green lines and the dashed lines.

These ideas already suggest algorithm Algorithm 4.2, the missing details of which we will now fill in. A more detailed discussion of each point follows later. In the algorithm we maintain a set of pairs $X \subseteq \mathbb{R}^n \times \mathbb{R}^n$ which contains a pair (q, w) if q is a Pareto optimal point and w the weight that has been used to generate q. As discussed before, X defines a lower and an upper bound of the Pareto curve. We initialize this set in Line 1 using function Initial, such that we have at least n different linearly independent Pareto optimal points (i.e., such that we have at least one facet). Then we will keep adding points to the set of points until we deem the approximation sufficient (Line 2). If the approximation is insufficient, then we pick a facet with maximal error

(see below for a discussion of error) in Line 3. We will refine this facet, by retrieving its weight w (Line 4), and calculating a Pareto optimal point q using w (findPoint in Line 5). If we have found a new Pareto optimal point, i.e., if q is not a member of facet F, then we will at (q, w) to X. Otherwise we know that the facet is already perfect, i.e., it defines both lower and upper bound.

Finding initial points (Initial). Finding the set of initial points is interesting in itself, especially if we are interested in only a subset of points. Many approaches are possible. For our implementation, we have chosen the following. First, we optimize the MDP with weights w_i for $1 \leq i \leq n$, where $(w_i)_i = 1$ and $(w_i)_j = 0$ for all $1 \leq i \neq j \leq n$. This is equivalent to finding the best possible value for each dimension. Sometimes (for example, if two dimensions are linearly dependent), this will not give us n different points. In that case, we use the separating hyperplane theorem to add new points.

Lemma 4.7 (Separating hyperplane theorem) *Given two non-intersecting convex shapes there is a hyperplane such that the one convex shape lies in its upper half-space and the other convex shape lies in its lower half-space.*

There are two possible ways to go about finding a separating hyperplane. Either we find a separating hyperplane between the points generated so far and a utopia point, or between these points and a target point, if we have it. The utopia point is $((q_1)_1, (q_2)_2, \ldots, (q_n)_n)$ if the q_i were generated using the w_i above. In both cases, we use the weight of the separating hyperplane to generate a new point. We repeat this process until enough points have been found.

Sufficient approximation (Sufficient). Sufficiency is defined depending on our goals. It can mean continuing until the distance between upper and lower bounds is small enough. It can mean continuing until we have proven that a target t is unreachable or until we have found a Pareto optimal point $p \geq t$, either as a combination of generated points, or just as a generated point (note that the complexity of these two objectives is different). Alternatively, it can mean continuing until we have approximated the Pareto curve above target point t to a satisfactory degree.

Finding the next facet (maxFacet). The next facet is always the facet with the maximal error. What exactly the maximal error is depends on our goals. In our case, we define the error of a facet as the maximal distance between the

facet and a point inside the upper bound. To that end, we calculate the error of a facet by solving the following linear programming problem for a facet with normal w and reference point q, in which p is the variable vector.

$$\max p \cdot w - q \cdot w$$

such that

$$p \cdot w > q \cdot w \tag{4.1}$$

$$p \cdot w' \leq q' \cdot w' \qquad \forall (q', w') \in X \tag{4.2}$$

In this LP, Equation 4.1 makes sure that p is outside the facet. Equation 4.2 makes sure that p is inside the upper bound we have defined so far. If we have a target t (i.e., if we are interested in only a subsection of the Pareto curve), then we add condition $p \geq t$.

Finding a point given a weight (findPoint). Given weight $w \in \mathbb{R}^n$ this means maximizing $\mathbb{E}[\alpha \circ (w \cdot r)]$, which can be done in polynomial time for all accumulations that we define.

Complexity of finding deterministic controllers

For all objectives discussed in this section, both finding a controller fulfilling a threshold and approximating a Pareto curve have algorithms with worst-case polynomial runtime-complexity. The search for a deterministic controller meeting a threshold criterion is a different matter. In [CMH06], the author shows that finding a deterministic controller for the discounted objective is NP-complete, and the proof can easily be adapted to the max sum and mean payoff objective.

4.6 Case studies

Automatic deceleration for cars

This case study considers an automatic collision avoidance system for cars. See [GKO+08] for a project report on automatic emergency braking systems prepared for the European commission. Among other technical difficulties, reports show that the system needs to react to unexpected driver behavior, unexpected road conditions, etc. Hence, these systems require substantial modeling effort.

This case study demonstrates how a (simple version of) a collision avoidance system can be modeled and analyzed.

In the setup we assume that the car we control is equipped with a device that detects the relative velocity to an object in front. It is the responsibility of the controller to brake the car as softly as possible until relative velocity reaches zero. Here we face a trade-off. Paramount is, of course, security: we do want to avoid collision, if possible, and reduce the velocity with which we collide, if collision is unavoidable. On the other hand, we do not want to engage the brakes harder or earlier than necessary.

Specification.

We use the following class to model this case study.

```
1  public class CCModel extends HybridModel<CCModel> {
2          // Distance to obstacle in meters
3          @CVAR(min=−10, max=200)
4          public double distance = 200;
5
6          // Relative velocity in meters per second
7          @CVAR(min=0, max=55)
8          public double velocity = 55;
9
10         // Update Frequency in Hz
11         public static int updateFrequency = 10;
12
13         // Used to discretize continuous action space
14          protected Discretized accelVar =
15                 new Discretized("Acceleration", 0, 6.5, 100);
16
17         public int nActions() {
18                 return accelVar.nBuckets();
19         }
20
21         public boolean isFinal() {
22                 return distance < 0 || velocity == 0;
23         }
24
25         public void next(int action, CCModel next) {
26                 double acceleration = accelVar.fromDiscretized(action);
```

```
27            double nextVelocity = velocity − acceleration / updateFrequency;
28
29            nextVelocity = Math.min(Math.max(nextVelocity, 0), velocity);
30
31            double nextDistance = distance −
32                (0.5 ∗ velocity + 0.5 ∗ nextVelocity + otherVel) /
33                  (double) (updateFrequency);
34            nextDistance = Math.max(nextDistance, −10);
35            nextDistance = Math.min(distance, nextDistance);
36
37            next.distance = nextDistance;
38            next.velocity = nextVelocity;
39        }
40
41        private double[] rewards = new double[2];
42
43        public double[] initialRewards() {
44            if (distance <= 0 && velocity > 0)
45                rewards[0] = −1 − velocity / 55;
46            else
47                rewards[0] = 0;
48
49            rewards[1] = 0;
50            return rewards;
51        }
52
53        public double[] rewards(int a) {
54            double accel = accelVar.fromDiscretized(a) / 6.5;
55            rewards[0] = 0;
56            rewards[1] = −(accel ∗ accel);
57            return rewards;
58        }
59    }
```

Our model has two model variables: `distance` measured in meters, and `velocity` measured in meters per second. The actions the controller can choose are varying degrees of braking strength (measured in meters per square second). Since we only allow finitely many actions, we use an auxiliary class

`Discretized` which takes care of linearly discretizing a variable in a certain range.

In this model, the state space forms a directed acyclic graph, because distance and relative velocity can only decrease. The model stops (i.e., has final states) when the car collides with the object in front, or if velocity reaches zero.

Note in the `next` method that this model uses a purely deterministic update function. In fact, we first de-discretize the acceleration, and then calculate the velocity of the next state based only on the velocity of this state and the braking force applied. The next distance to the obstacle is calculated based on the distance of this state and the average velocity of this state and the next state. Finally, we make sure that all state variables stay inside the described bounds.

Our rewards have two components. First, we incur a cost whenever we collide with the obstacle. The height of the cost depends on the velocity at the time of collision. Second, we incur a cost each time we apply force. The height of the cost is determined by the square of the applied force.

Controller generation. We use the total sum accumulation function to find an optimal controller. This seems to be a good choice because each run encounters a final state in a finite number of steps almost surely. In addition, we have a half-order on the states. By executing the value iteration steps indicated by this half-order, the value of each state has to be updated only once. This insight dramatically decreases the runtime of value iteration. We used 200 sample points for distance, and 50 sample points for velocity.

In Figure 4.16 we display the approximation of the Pareto curve generated for the braking system. On the x-axis, we see the expected velocity with which collision occurs, where we average over all states with maximal distance. On the y-axis, we see the average squared deceleration. There is a clear trade-off. Since it seemed to us that avoiding collisions was the most important aspect of a controller, we picked a weight that produces a reward very far on the left of the plot: $[0.988; 0.0116]$.

Analysis. We are first going to discuss the shape of the discrete controller and the interpolated controller. In Figure 4.18a, we display the action chosen for each state. Note that synthesis returns intuitively correct results. If the car is still far away from the obstacle, and if the relative velocity is low, then the braking force applied is low as well. If, on the other hand, relative speed is

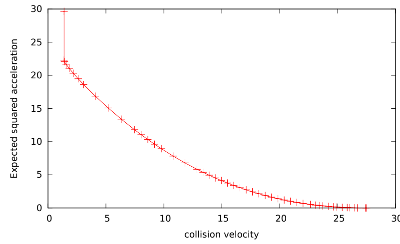


Figure 4.16: Pareto curve of collision avoidance system.

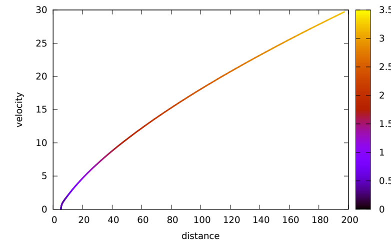
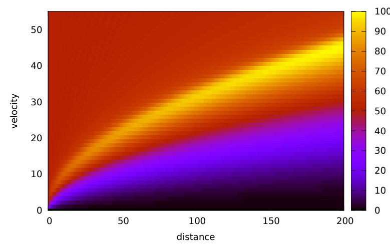
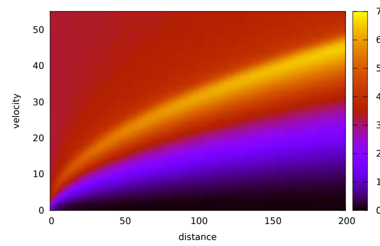


Figure 4.17: Trace starting at distance 50 meters and velocity 30 m/s. Plot color indicates applied breaking force.

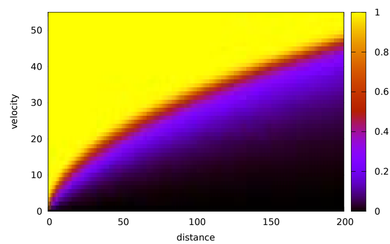


(a) Chosen actions

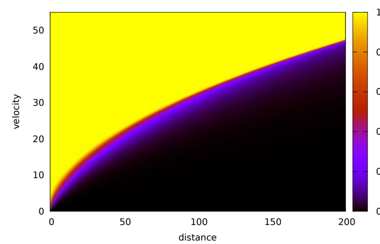


(b) Interpolated controller

Figure 4.18: Comparison between chosen actions and interpolated controller.



(a) 200 sample points for distance, 50 for velocity



(b) 500 sample points for distance, 200 for velocity

Figure 4.19: Probability of collision with obstacle in different resolutions

high, or if we are close, then the braking force applied is higher. Figure 4.18b shows the linearly interpolated controller (see Definition 4.8). The interpolation scheme provides a smoothed controller. Note that visual inspection of this plot alone is not sufficient: the plot does not tell us anything about the dynamics of the system. It would, for example, be possible that the controller will “stutter”, i.e., vary the applied braking force quickly.

To analyze stuttering, we now turn to a single trace in Figure 4.17. On the x-axis we have the distance to the obstacle, while we have the relative velocity on the y-axis. The color of the line depicts the current braking force the controller applies. This trace was generated by starting in a continuous state with distance 200 meters and relative velocity 30 meters per second. We then used the continuous transition function to generate this plot. To follow the trace in time, we start in the upper right corner and then follow the line by going left until velocity reaches 0. In the beginning the controller applies a braking force of $3.5m/s^2$. It then slowly and smoothly reduces the braking force until relative velocity reaches zero shortly before colliding with the object. Note that no stuttering occurs.

We are now going to look at the probability of colliding with the obstacle. We analyzed this property using both PCTL model checking and Bayesian model checking. We first evaluated the controller in its own environment, and then in an environment with reduced braking efficiency. The probability of a collision is plotted as heat-maps in Figure 4.19a and Figure 4.19b. The difference between the two figures is in the discretization resolution we used to check the property. In Figure 4.19a we used 200 points for distance and 50 points for relative velocity (we used the same resolution for controller generation). The yellow area above a certain threshold is not surprising. This is the area in which a collision happens even if maximal braking force is applied. The surprising part is the big area of uncertainty (in read, violet and blue) below the threshold. In this area, we are not sure if a collision is going to happen, although the transition function is deterministic. This uncertainty is a result of the probabilities introduced when we discretize the continuous transition function. As evidence we present Figure 4.19b, which checks the same property with a higher resolution (500 sample points for distance, 500 for distance). Both plots have the same shape, but the shape with higher resolution presents a much smaller area of uncertainty.

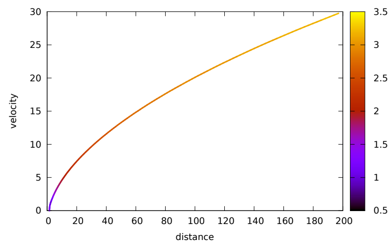


Figure 4.20: Trace with reduced braking effectiveness

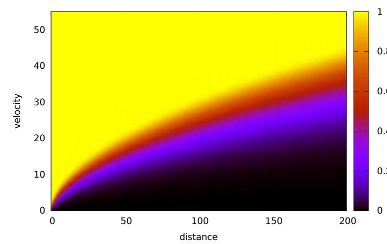


Figure 4.21: Probability of collision with reduced braking effectiveness.

To judge the robustness of the controller against unexpected reductions in braking strength, we first calculated the probability of collision in a model in which braking strength is reduced by 20%. We have plotted the generated trace in Figure 4.20 and the heat map in Figure 4.21

When comparing the original trace and the trace with reduced effectiveness, it becomes clear that the controller comes to a stop later when braking force is reduced. It also adapts to the new situation: while braking force does not rise, high braking force is maintained longer than in the trace with full braking effectiveness.

When comparing the two heat maps that depict probability of a collision, we firstly see that the area of certain collision grows. This is to be expected with lower braking strength. We also see that the area in which the probability of collision is neither zero nor one grows.

Stochastic model checking reveals the following results. At a distance of 200 meters, the synthesized controller is effective for a velocity of up to 47.25 meters per second, a figure that is reflected by Figure 4.19b. When initial velocity is treated as uniform over all possible values, then the probability of collision lies in the interval $[0.035, 0.0545]$ with probability 95%. With 20% reduced effectiveness of braking, the controller is successful at least until a velocity of 37.5 is reached. The probability of collision lies in the interval $[0.234, 0.254]$ with probability 95%.

Possible extensions of the model. This case study is fairly simplistic. We will now discuss how the model can be extended to get closer to current systems.

Real world models are usually three-stage systems. Firstly, the braking system is pre-charged such that the reaction time of the system will be miti-

gated. The second stage consists of warning the driver visibly or audibly about an imminent collision. Only in the last stage, and only in very recently deployed systems, do controllers actually initiate braking themselves. It would be fairly easy to extend the model to encompass these features, though research is required as to how drivers react to the warning signals.

Real world models also have to consider failing sensors or a reduction in braking strength. These could be included in the model, but it that they should be handled by a higher-instance, which detects sensor failures or conflicting sensor results and handles accordingly. Slight reductions of braking strength are handled by the controller we synthesize, as shown by the analysis. In the event of a catastrophic failure of the braking system, even the best controller is unable to prevent collision and we consider it therefore best to not include this possibility in the model.

Finally, it would be possible to extend this model to make it more robust with regard to reduced braking efficiency. One possibility is a feed-back loop informing the controller of the reduced braking efficiency. Another is to synthesize different controllers for different braking strengths, and let a higher-order system select the appropriate controller. A third possibility is to add a higher-order system that amplifies the controllers braking strength as necessary.

Adaptive cruise control

This case studies considers an adaptive cruise control system (ACC), which we described already in Section 4.4. ACCs are now built into luxury cars and are responsible for automatically maintaining a fixed distance to the car in front. Such a system senses (1) the current distance between the car it equips and the car in front, and (2) their relative velocity, i.e,. by how much the distance shrinks or grows per second. On the one hand, the goal of this system is to reach and maintain the desired distance quickly. On the other hand, the controller is also responsible for pleasant driving. That is, it should not unnecessarily or suddenly accelerate or jerk (where jerk is the change of acceleration over time). There is a trade-off between these two criteria, and our framework allows study of trade-offs like these. An additional concern is that the relative velocity is not exclusively under the control of the system. Instead, since we do not know and cannot predict the other driver's intentions, we assume that she is going to behave randomly. See [VE03] for an overview of research on collision avoidance

and adaptive cruise control for cars. [LDCd06] studies a synthesis approach similar to ours for cooperative adaptive cruise control. This approach assumes that cars communicate via compatible cruise control systems.

Specification. This case study uses the following class as model.

```
1  public class ACC extends Model<ACC> {
2      // Distance from car in front
3      @CVAR(min=0, max=100)
4      public double distance;
5
6      // Relative velocity
7      @CVAR(min=−14, max=14)
8      public double velocity;
9
10     // Distance we desire
11     public double desiredDistance = 50;
12
13     // Updates per second
14     public int ticks = 10;
15
16     @Override
17     public void next(double acceleration, ACC target) {
18         double random_acceleration = normal.sample(0, 4.0);
19         double nextVelocity =
20             velocity + (acceleration + random_acceleration) / ticks;
21         double nextDistance = distance −
22             (0.5 ∗ velocity + 0.5 ∗ nextVelocity) / ticks;
23         target.velocity = nextVelocity;
24         target.distance = nextDistance;
25     }
26
27     @Override
28         public double[] rewards(double acceleration) {
29         double[] rewards = new double[2];
30                 rewards[0] = −Math.abs(desiredDistance − distance);
31                 rewards[1] = −acceleration∗ acceleration;
32                 return rewards;
33         }
34 }
```

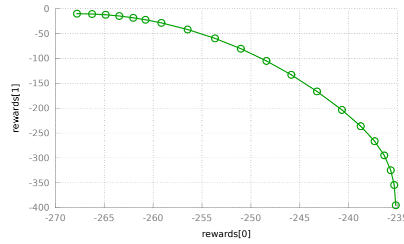


Figure 4.22: Pareto Curve of ACC

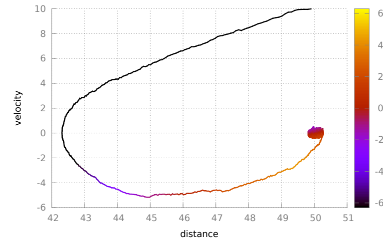


Figure 4.23: Trace of one run

A few features here are noteworthy.

- Variables `distance` and `velocity` are part of the model, while variables `desiredDistance` and `ticks` are not part of the model (because they are constant).
- We first sample a random acceleration for the other car, with mean zero and standard deviation 4. Next, we calculate the velocity of the next state, based on the current velocity, the random acceleration and the acceleration we got as input. Lastly, based on old velocity and distance and on the new velocity, we calculate the distance of the next state.
- Additionally, we define the rewards the controller gets for its decisions in method `double[] rewards(double acceleration)`. In the case of `ACC`, it receives a cost (negative reward) depending on how far the current distance is from the desired distance (`rewards[0]`), and a cost depending on how much it accelerates (`rewards[1]`).

Note that the rewards define exactly the trade-off mentioned before. On the one hand, we want to minimize both `rewards[0]` and `rewards[1]`, but applying less acceleration will lead to a greater deviation from our desired distance. On the other hand, being stricter about staying close to the desired distance requires more acceleration.

Controller generation. It was not clear to us what weights we should assign. We therefore generated a part of the Pareto curve based on minimal performance criteria. We display part of the curve in Figure 4.22. Based on this curve, we picked weight $(0.9788, 0.0211)$ to generate a controller.

We show the actions of a controller generated with weight $(0.9788, 0.0211)$ by our framework in Plot 4.24. On the x-axis we present the distance to

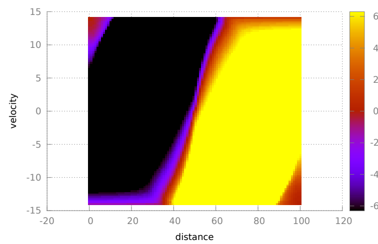


Figure 4.24: Controller Plot with weight $(0.9788, 0.0211)$

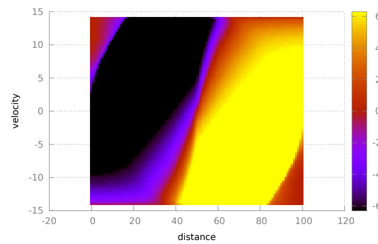


Figure 4.25: Plot of Controller weith weight $(0.9588, 0.0411)$

the car in front, while we present the relative velocity on the y-axis. The color indicates the applied acceleration. For example, where the distance is as desired (50 m), and the relative velocity is 0, no further acceleration is applied. Going through this point is a diagonal going from roughly (35, -15) to (65, 15) where applied acceleration equals zero. In this area, the controller judges the relative velocity just right to reach the desired distance quickly enough. As we move horizontally outwards from this narrow band, the acceleration the controller applies rises sharply. Especially, as either distance or relative velocity decreases, the controller increases the applied acceleration.

Plot 4.23 shows one trace of the interplay between controller and environment as it happens in the continuous environment (i.e., we run the program defined above as it is). It starts out in position (50,10), i.e., where the distance is as desired but we are closing in too fast. As we follow the trace, we see that the car equipped with an ACC gains on the car in front (because its velocity is greater than that of the other car). The color of the trace shows the applied braking force in each particular moment. As we can see, the controller breaks the car harshly until he reaches a relative velocity of -3 m/s. At this point it slowly decreases the de-acceleration until a relative velocity of about -5 m/s is reached. It now maintains speed until we reach a distance of about 47 m (i.e., the car is 3 meters too close). Would the controller maintain speed here, then it would overshoot the desired distance. Instead, it gently accelerates the car again until it reaches a relative velocity of 0 and is very close to the desired distance. The "ball" region around the desired distance and relative velocity 0 shows how the controller reacts to the random behaviour of the car in front.

In Plot 4.25, we present a controller generated with weight $(0.9588, 0.0411)$. In comparison to the weight above, we have decreased the importance of

`rewards[0]`, and increased the importance of `rewards[1]`. This decreases the importance of the distance to the other car and increases the importance of not applying too much acceleration. This has the effect of growing the band where relative velocity is judged adequate, and also moving the area of increased acceleration further out.

These two examples show that the weight chosen when optimizing a controller can have a strong influence on the one hand, and that choosing weights is not intuitive on the other hand, especially as the number of dimension increases. We therefore consider the easy availability of Pareto curves an asset of our framework.

Verification

Probabilistic model checking. We use probabilistic model checking to judge how the controller behaves if assumptions we made about the environment are not met and how the controller behaves with regard to properties that were not used for its construction. As an example of the latter, we can consider the stability of the system. In control theory, stability is the property of a system to reach a bounded set of states and never leave it. In our case, we define this set as a bound on the deviation of the distance of the two cars from the desired distance. We can easily state a desired bounded set of states via PCTL formula: $P_{=?}[\mathsf{G}(|d - 50| < c)]$, where d denotes the distance between the two cars and c is a constant. Our framework takes this formula as input and calculates the probability of being in a stable (i.e., in a state from which only other stable states can be reached) for each state. In Plot 4.26 we plot the probability of being in a stable state, where we arbitrarily judge a state stable if $c = 5$. Note, first, that any state with a distance not within 5 meters of the desired distance cannot be stable. Note second, that as the relative velocity becomes more extreme, the probability of a state being stable goes towards zero. At the very extreme ends, the controller is unable to maintain control over the relative velocity in a way that guarantees that the distance will stay within 5 meters of the desired distance. Closer to the area where relative velocity is 0, the probability lies between 0 and 1. The reason that there is no sharp threshold between probability 0 and 1 lies in the random acceleration of the car in front. With a certain probability, the car in front will contribute to moving the distance towards the desired distance (braking where the controller

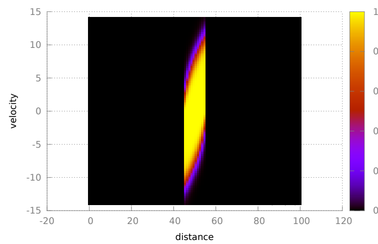


Figure 4.26: Probability of being in a stable state

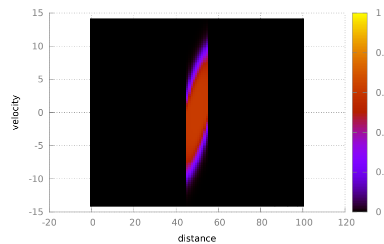


Figure 4.27: Probability of being in a stable state with 80% acceleration effectiveness.

needs to accelerate and vice versa). With a certain probability, the car will work against our controller (accelerating where we need to accelerate, braking where the controllers needs to break as well).

Judging the probability of reaching a stable state is an additional task. This can be easily done in our framework by checking the controller against formula $P = ?(\mathsf{F}\, P_{=1}[\mathsf{G}(|d - 50| < c)))$. As it turns out, the probability is 1 for all states of our model, i.e., under the given assumptions the controller is able to reach and maintain a low deviation from the desired distance to the other car almost surely.

We can now modify certain parameters of the system, and judge its behaviour under these modified assumptions. For example, consider very rainy weather, where we assume that acceleration only works at 80% efficiency of what the controller expects[2]. The probability of a state being stable is plotted in Figure 4.27. In this case, the probability is only about at most 40%[3].

Lastly, our framework also allows us to easily turn the tables around and choose actions for the car in front. In this new model, the braking force applied by the ACC is determined by a controller we previously generated, and we now synthesize worst-case accelerations for the car in front. This is easily achieved by replacing `next` above by the following.

```
1    public void next(double acceleration2, ACC target) {
2        double acceleration = controller.get(this);
3        double nextVelocity =
4            velocity + (accleration + acceleration2) / ticks;
```

[2]This assumes that we use the same controller in bad weather, and that we cannot compensate

[3]Note that there are techniques for dealing with uncertain parameters(e.g., robust Markov decision processes

```
5         double nextDistance = distance −
6             (0.5 ∗ velocity + 0.5 ∗ nextVelocity) / ticks;
7         target.velocity = nextVelocity;
8         target.distance = nextDistance;
9     }
```

Now we can apply the very same techniques we used above to compute the worst-case probability of a state being stable.

Bayesian Probabilistic Model Checking. As we have noted before, the models described in Java lend themselves directly to continuous state space simulation. We cannot check the PCTL formula above as it is, because it expresses properties over infinite runs. Instead we have to give time-bound formulas. As an example, we consider a formula expressing the property "What is the probability that we reach a state inside 5 meters around the desired distance in 1000 steps (where 1 step is 10 milliseconds long), and stay inside this area for the next 1000 steps." Bayesian probabilistic model checking allows us to make statements like "given the set of samples generated, the probability that this formula is true lies in the interval $[a, b]$ with probability c". In this framework, the width of the interval $b - a$ and confidence c are configurable. In our case it turns out, that with 95% confidence the formula holds with probability $[0.98, 1.00]$ from some randomly generated state. We assume that the remaining cases will require longer runs. For comparison, we decreased the efficiency of the applied acceleration to 80%. In this case, we get an interval [0.971297, 0.991297], which shows us that the controller performs well even under adverse conditions.

Related work.

Mountain car

The mountain car example [Sut95] is a famous example usually used for reinforcement learning. In this setting, a car is caught in a ditch, and its engine is too weak to just drive up at one of the sides. Figure 4.29 depicts the landscape. The car is assumed to start motionless at the bottom of the ditch (position -0.5).

The model. Figure 4.28 contains the code of this model. We have to state variables (`position` and `velocity`). It is our goal to get out of the ditch we

landed in, i.e., the car has to reach position 0.6. The controller can influence the car by accelerating forward or backward full-throttle, or by letting the motor idle. That is, the controller has three possible actions in each state. The velocity of the car is also influenced by gravity. The controller incurs a cost of 1 for each time step, and a different cost for each applied acceleration.

Synthesis. We present the Pareto curve in Figure 4.30. The almost linear trade-off between acceleration and time is uncommon for our case studies: we usually see curves that are entirely rounded. Only after an expected number of about 70 applied accelerations do we start to see a non-linear trade-off.

Analysis. We picked two controllers for analysis. One controller that was generated with weight $[0.9, 0.1]$, which we will call `slow` from now on, and one controller that was generated with weight $[0.1, 0.9]$, which we will call `fast` from now on. The expected reward of `slow` is $[-61.57, -176.73]$, and that of `fast` is $[-98.61; -106.24]$. Their outcomes are very different: the slow controller uses about 40% fewer accelerations, but takes about 50% more time than the fast controller.

How the different controllers achieve their different goals can be seen in the two trace plots in Figure 4.31, both of which were started at position -0.5 and velocity 0, i.e. motionless at the bottom of the ditch. In both plots we have the position of the car in the ditch on the x-axis, and the velocity on the y-axis. We additionally display the applied action in color: black means reverse, red means idle and yellow means forward.

We will first analyze the trace of the `fast` controller in Figure 4.31a. Right at the start the controller accelerates and manages to climb the hill up a little (yellow section going from position -0.5 to position -0.4). It then immediately reverses, rolls down the hill and back up the hill on the opposite site. It manages to climb the hill up almost all the way, where it reaches velocity 0 (black section going from -0.4 to -1). It then accelerates again, rolls down the one side of the ditch and up the other, and keeps accelerating until it reaches position 0.6. Note that velocity decreases after position -0.4, although the controller applies force: this is a result of the comparatively strong gravity.

Contrast this strategy with the strategy of the energy-conserving controller in Figure 4.31b. First, note the long stretches of red, which depict an idle motor. At the beginning, the car backs up the hill to the left a little (black section going from -0.5 to -0.7). It then rolls down the hill half-way with an

```
1  public class MountainCar extends HybridModel<MountainCar> {
2          @CVAR(min=−1.2, max=0.7)
3          public double position = −0.5;
4
5          @CVAR(min=−0.07, max=0.07)
6          public double velocity = 0.0;
7
8          @Override
9          public void next(int action, MountainCar target) {
10                 int accel = action − 1;
11                 target.velocity = velocity + 0.001 ∗ accel +
12                         Math.cos(3 ∗ position) ∗ (−0.0025);
13                 target.position = position + target.velocity;
14         }
15
16         @Override
17         public int nActions() {
18                 return 3;
19         }
20
21         private double[] rewards = new double[2];
22         @Override
23         public double[] initialRewards() {
24                 rewards[0] = rewards[1] = 0;
25                 return rewards;
26         }
27
28         @Override
29         public double[] rewards(int action) {
30                 int accel = action − 1;
31                 rewards[0] = −Math.abs(accel);
32                 rewards[1] = −1;
33                 return rewards;
34         }
35
36         @Override
37         public boolean isFinal() {
38                 return position >= 0.6;
39         }
40 }
```

Figure 4.28: Code of mountain car model

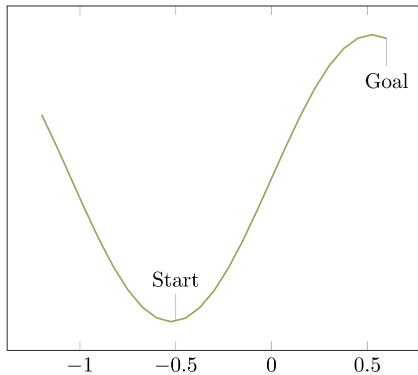


Figure 4.29: A mountain landscape

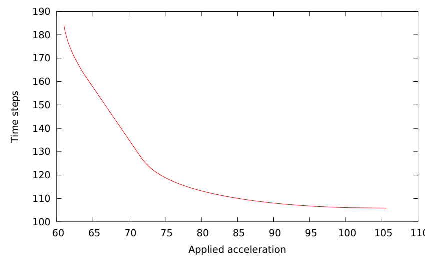
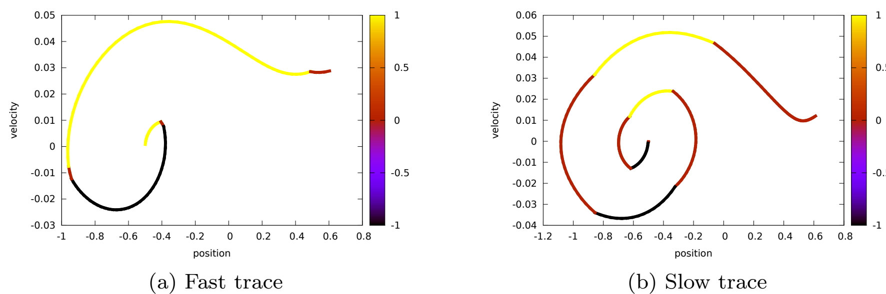


Figure 4.30: Pareto curve of mountain car controllers.



(a) Fast trace

(b) Slow trace

Figure 4.31: Traces of slow and fast controllers, started at the bottom of the ditch. X-axis depicts position, Y-axis depicts velocity, color depicts applied action.

idle motor until -0.6 and then accelerates until it reaches position -0.3 (yellow section), first passing the bottom of the ditch and then climbing up the right hill somewhat. It then starts idling (red section), continuing to roll the hill up a little until position -0.2, and then it continues to roll the hill down, first with an idle motor (until -0.3) and then with reversing the car (until -0.9), and then with an idle motor again. It passes the bottom of the ditch, rolls up the left hill until reaching position -1, and then starts rolling the hill down again, at first with an idling motor. While rolling the hill down, passing the ditch and rolling up the right hill, the controller gives a last push (last yellow section) from position -0.9 to position -0.1. The car then finally reaches the top of the hill (position 0.6). Note that the `slow` controller exits the ditch with a much lower velocity than the `fast` controller.

When comparing the two traces (which have been created using continuous dynamics, not discretized dynamics), we see that the `slow` controller needs 176

steps, while the `fast` controller needs only 106 steps. In contrast, the `slow` controller only applies 61 acceleration actions, while the `fast` controller applies 98 acceleration actions. Comparison of these (exact) numbers with the expected (discretized) numbers used for generation shows that the approximation used in our framework delivers good results in the case study.

Conclusion

We have presented a new framework and several case studies for probabilistic quantitative verification and synthesis. The intended contribution of this framework is to stress the need to do verification and synthesis in a loop in the same framework. In the course of this, we showed how we can use a Java EDSL as a modeling language. In addition, we have generalized and extended previous work on approximating Pareto curves of MDPs. We have used this EDSL on several case studies which have emphasized the need of Pareto curves and verification of synthesized controllers.

5 Analyzing the Next Generation Airborne Collision Avoidance System

In which we apply quantitative verification and synthesis to analyze and improve the next generation collision avoidance system for airplanes.

Résumé

La nouvelle génération de système anti-collision aérienne ACAS X se base sur le système détérministique traditionnel, comme le système actuel TCAS. Pour augmenter la puissance ACAS X dépend des modéles probabilistiques, ce que singifie la variation de l'incertitude. Le travail présenté dans cet article montre le défi de ACAS X et décrivent les études de l'application de la vérification probabilistique et méthodes de syntèse s'addressant ces défis. Comme ces méthodes probabilistique sont utilisées par défaut, nous avons développé une base pour gérer les systèmes avec les caractéristics similaires à celles d'ACAS X. Nous décrivont donc l'application de cette base pour ACAS X, les résultats et les recommendations suite à notre analyse.

5.1 Introduction

The current onboard collision avoidance standard, TCAS [KD07](Traffic Collision Avoidance System) has been successful in preventing mid-air collisions. However, its deterministic logic limits robustness in the presence of unanticipated pilot responses, as exposed by the collision of two aircraft in 2002 over Überlingen, Germany [Joh04]. To increase robustness, Lincoln Laboratory has

been developing a new system, provisionally known as ACAS X(Air Collision Avoidance System), which uses probabilistic models to represent uncertainty. Simulation studies with recorded radar data have confirmed that this novel approach leads to a significant improvement in safety and operational performance. The Federal Aviation Administration (FAA) has formed a team of organizations to mature the system, aiming to make ACAS X the next international standard for collision avoidance.

The adoption of a completely new algorithmic approach to a safety-critical system naturally poses a significant challenge for verification and certification. Our goal in this work is to study the applicability of formal probabilistic verification and synthesis techniques, which go beyond simulation studies [KNP11, KZH+11]. Our study was driven by tasks defined in collaboration with the ACAS X team to be complementary to their verification efforts. During the course of our work, we identified shortcomings of existing tools, which lead us to develop a framework customized for ACAS X (or similar systems). In our framework, models are expressed in a traditional programming language for increased expressiveness, and verification and synthesis algorithms are designed for scalability and efficiency.

The contributions of this work can be summarized as follows: 1) Development of a faithful model for synthesis of the ACAS X controller, based on the Lincoln Laboratory publications [KC11]; 2) Development of customized verification and synthesis algorithms for efficient handling of ACAS X (and like) systems; 3) Identification of design and verification challenges for ACAS X as related to probabilistic verification and synthesis; 4) Results obtained from the application of our framework to ACAS X and recommendations for the ACAS X effort.

The results of our work will serve as input for the certification of ACAS X. Due to access restrictions, we analyze a previous version of the system [KC11], but are currently working with the ACAS X team to extend our work to the current version. We believe that ACAS X presents researchers in probabilistic verification and synthesis with a unique opportunity to focus on a relevant, safety-critical case study. For this reason, we are preparing a public release of our models and framework, to encourage other members of the community to build on our work.

The remainder of this chapter is organized as follows. Section 5.2 describes

the ACAS X system as designed and deployed by the ACAS X team. In addition to these techniques, our work implements and applies formal verification and synthesis approaches, described in Sections 5.3 and 5.4. We discuss implementation details in Section 5.5, and Section 5.6 concludes the chapter.

5.2 The ACAS X system

Model Description. Similarly to the current standard TCAS, ACAS X [KC11] uses several sources to estimate the current state of the plane on which it is deployed, and the planes in its vicinity. If it detects the possibility of an imminent collision (less than 40 seconds away) and it produces vertical maneuver advisories (to climb or descend) in order to avoid the collision. Both TCAS and ACAS X operate at a frequency of one state update and advisory per second.

The ACAS X model consists of two airplanes on collision course. Loss of Horizontal Separation, from now on denoted as `LHS`, describes the situation where two airplanes are in the exact same location when their height difference is ignored. A Near Mid-Air Collision (NMAC) occurs when the two airplanes are within 100 ft of each other when `LHS` occurs. We refer to the plane equipped with ACAS X as *our* plane (often referred to as ownship in the literature), and the other plane as *intruder* (similarly to [KC11]).

The model has 5 parameters: (1) $h \in [-1000, 1000]$ ft, the height difference between the two planes, (2) $\delta h_0, \delta h_1 \in [-2500, 2500]$ ft / min, our and the intruder's climbing rates (3) adv the advisory produced by ACAS X one second ago (4) ps the pilot state. The state can be described in our framework (Chapter 4) as follows.

```
1  public class ACASModel extends HybridModel<ACASModel> {
2          // Height distance between airplanes
3          @CVAR(min=−1000, max=1000)
4          public double h;
5
6          // Our climbing rate
7          @CVAR(min=−2500, max=2500)
8          public double h0;
9
10         // Intruder's climbing rate
11         @CVAR(min=−2500, max=2500)
```

```
12      public double h1;
13
14      // State of pilot — encodes last advisory as well as pilot's reaction
15      @DVAR(min=0, max=12)
16      public int pilotState;
17
18      // ...
19  }
```

Listing 5.1: State space of `ACASModel`

Pilot state and advisories can take the following values.

- `COC` stands for "clear of conflict" — the pilot is free to choose how to control the plane.
- `CLI1500` / `DES1500` stand for "climb / descend with 1500 ft / min", respectively; they advise the pilot to change the climbing rate with $1/4g$ until reaching a climbing rate of 1500 ft / min / -1500 ft / min, respectively.
- Advisories `SCLI1500` / `SDES1500` and `SCLI2500` / `SDES2500` are similar but employ an acceleration of $1/3g$. Moreover, `SCLI2500` / `SDES2500` target a final climbing rate of 2500 ft / min / -2500 ft / min, respectively.

At each point, the state encodes the advisory given a second ago, as well as the pilot's reaction to it. Pilot state and advisories can take on any of these values, but not all combinations are possible: the pilot can either follow the advisory (i.e., `response == lastAdvisory`), or perform random maneuvers (i.e., `response == COC`), since studies have shown that pilots may not react immediately or at all to an advisory. This part can be modeled in our framework as follows.

```
1   public class ACASModel extends HybridModel<ACASModel> {
2       // ...
3       public enum Advisory {
4           COC,
5           CLI1500,
6           DES1500,
7           SCLI1500,
8           SDES1500,
9           SCL2500,
```

```
10              SDES2500;
11
12              public boolean isStrengthening(Advisory other) {
13                      // ...
14              }
15
16              public boolean isReversal(Advisory other) {
17                      // ...
18              }
19
20              public boolean isWeakening(Advisory other) {
21                      // ...
22              }
23
24              public boolean isAlert(Advisory other) {
25                      // ...
26              }
27      }
28
29      // Last advisory given − encoded in pilotState
30      public Advisory lastAdvisory;
31      // Pilot's response to last advisory − encoded in pilotState
32      public Advisory response;
33
34      //...
35 }
```

Listing 5.2: Advisory declaration of `ACASModel`

The dynamics of the system are governed by the physics of the two planes and by the behavior of the two pilots. We model the intruder as behaving randomly, with his acceleration drawn from a normal distribution. We model the pilot of our plane probabilistically. Whenever his reaction and the advisory we give do not agree, his behavior is governed by discrete probability distributions. The acceleration of our airplanes depends on the pilot's reaction. If the pilot is following the advisory, then the acceleration is determined by the advisory. Otherwise, it is random, with the same parameters the intruder uses. This can be modeled as follows in our framework.

```
1  public class ACASModel extends HybridModel<ACASModel> {
2          // ...
3          // Probability to react if pilot's reaction is slow
4          protected static final double responseSlow = 1/6.0;
5          // Probability to react if pilot's reaction is fast
6          protected static final double responseFast = 1/4.0;
7
8          // Probability distribution used if pilot is slow to react
9          public DiscreteDistribution<Boolean> slowReaction;
10         // Probability distribution used if pilot is fast to react
11         public DiscreteDistribution<Boolean> fastReaction;
12
13         // Earth's gravity in ft/s^2
14         public static final double G = 32.1740;
15
16         // Standard deviation of random acceleration
17         public static final double sigma = 3;
18
19         public ACASModel() {
20                 slowReaction = new DiscreteDistribution<Boolean>();
21                 fastReaction = new DiscreteDistribution<Boolean>();
22
23                 slowReaction.add(false, 1-responseSlow);
24                 slowReaction.add(true, responseSlow);
25                 fastReaction.add(false, 1-responseFast);
26                 fastReaction.add(true, responseFast);
27         }
28
29         @Override
30         public void next(int action, ACASModel next) {
31                 // Set lastAdvisory and response
32                 decodePilotState();
33                 // Turn action number into an advisory
34                 Advisory adv = decodeAdv(action);
35
36                 // If the advisory has not changed, or if the advisory is
37                 // COC, then the next pilot response is the advisory.
38                 // This models an immediate reaction to COC
```

```
39          if (adv == response || adv == Advisory.COC) {
40              next.response = adv;
41          }
42          // If the advisory changes and is not a COC, sample to see
43          // if the pilot reacts or is in COC mode
44          else if (adv == Advisory.CLI1500 || adv == Advisory.DES1500) {
45              next.response = slowReaction.sample() ? adv : Advisory.COC;
46          } else {
47              next.response = fastReaction.sample() ? adv : Advisory.COC;
48          }
49          // Remember the advisory we give
50          next.lastAdvisory = adv;
51          // Update intruder's climbing rate
52          next.h1 = this.h1 + normal.sample(0, sigma) * 60;
53
54          if (next.response == Advisory.COC) {
55              // Update our own climbing rate randomly if that is our reaction
56              next.h0 = this.h0 + normal.sample(0, sigma) * 60;
57          } else {
58              // Update our climbing rate by h0Diff, according to advisory and situation
59              double h0Diff = 0;
60              switch (next.response) {
61              case CLI1500:
62                  if (h0 < 1500)
63                      h0Diff = G / 4.0;
64                  break;
65              case SCLI1500:
66                  if (h0 < 1500)
67                      h0Diff = G / 3.0;
68                  break;
69              case SCL2500:
70                  if (h0 < 2500)
71                      h0Diff = G / 3.0;
72                  break;
73              case DES1500:
74                  if (h0 > -1500)
75                      h0Diff = -G / 4.0;
76                  break;
```

```
77                 case SDES1500:
78                     if (h0 > −1500)
79                         h0Diff = −G / 3.0;
80                     break;
81                 case SDES2500:
82                     if (h0 > −2500)
83                         h0Diff = −G / 3.0;
84                     break;
85                 }
86                 next.h0 = this.h0 + h0Diff ∗ 60;
87             }
88
89             // Update height difference based equally on the climbing rates
90             // of this round and the next round
91             next.h = h + ((h0 + next.h0)/2 − (h1 + next.h1)/2) / 60;
92
93             // Make sure values fall into their defined intervals
94             if (next.h0 >= 2500) next.h0 = 2500;
95             if (next.h0 <= −2500) next.h0 = −2500;
96             if (next.h1 >= 2500) next.h1 = 2500;
97             if (next.h1 <= −2500) next.h1 = −2500;
98             if (next.h >= 1000) next.h = 1000;
99             if (next.h <= −1000) next.h = −1000;
100
101            // encode next state's pilotState variable according to
102            // next.adv and next.response
103            next.encodePilotState();
104        }
105
106        // ...
107 }
```

Listing 5.3: Dynamics of `ACASModel`

In order to generate a controller, each ACAS X advisory receives a cost/reward, where costs are rewards with negative values. Reward `COC` is associated with switching from any alerting state to `COC`; `Alert` is a cost associated with switching from `COC` to either `CLI1500` or `DES1500`; `Reversal` is a cost associated with switching from any climbing to any descending advisory, or vice

versa; `Strengthening` is a cost associated with switching from any climb/descent advisory with goal 1500 ft / min to `SCLI2500`/`SDES2500`, respectively; `NMAC` is a cost associated with the occurrence of an NMAC.

```
1  public class ACASModel extends HybridModel<ACASModel> {
2      // ...
3      private static double[] nmacReward = new double[] {−1, 0, 0, 0, 0};
4      private static double[] alertReward = new double[] {0, −1, 0, 0, 0};
5      private static double[] strengtheningReward = new double[] {0, 0, −1, 0, 0};
6      private static double[] reversalReward = new double[] {0, 0, 0, −1, 0,};
7      private static double[] cocReward = new double[] {0, 0, 0, 0, 1};
8      private static double[] zeroReward = new double[] {0, 0, 0, 0, 0};
9
10     @Override
11     public double[] initialRewards() {
12         if (−100 <= h && h <= 100) {
13             return nmacReward;
14         } else {
15             return zeroReward;
16         }
17     }
18
19     @Override
20     public double[] rewards(int a) {
21         Advisory newAdv = decodeAdv(a);
22         decodePilotState();
23         if (newAdv.isAlert(lastAdvisory)) {
24             return alertReward;
25         } else if (newAdv.isStrengthening(lastAdvisory)) {
26             return strengtheningReward;
27         } else if (newAdv.isReversal(lastAdvisory)) {
28             return reversalReward;
29         } else if (newAdv == Advisory.COC && lastAdvisory != Advisory.COC) {
30             return cocReward;
31         }
32         return zeroReward;
33     }
34 }
```

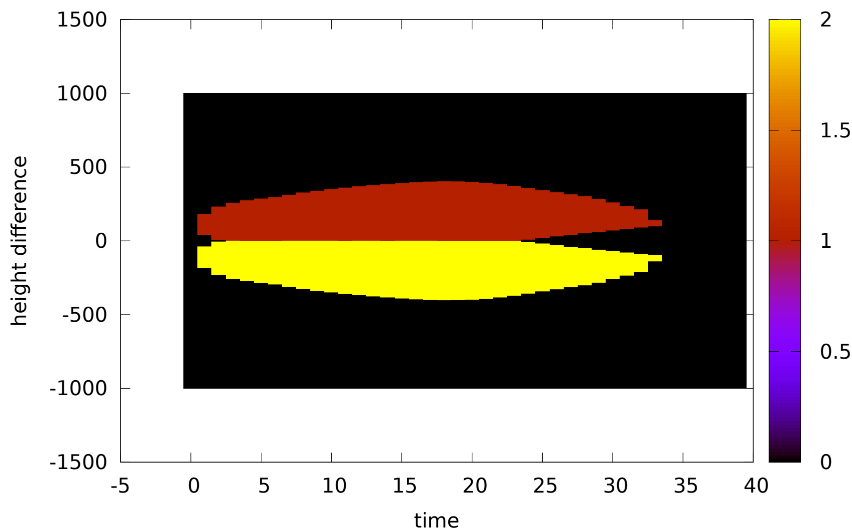


Figure 5.1: Controller generated in resolution $(10, 10, 10)$ with weights as in [KC11]. x-axis shows time until `LHS`, y-axis height difference. Parameters δh_0 and δh_1 are zero throughout, and adv = ps = `COC`. Color indicates selected advisory: black (0) for `COC`, red (1) for `CLI1500`, yellow (2) for `DES1500`.

Controller generation. For controller generation we use the framework described in Chapter 4. That is, we will define a discretized state space D_R, parameterized by a resolution vector $\mathbb{R} = (r_{\delta h_0}, r_{\delta h_1}, r_h) \in \mathbb{N}^3$, where $r_{\delta h_0}$ is the number of discretization points above and below 0 for parameter δh_0, and $r_{\delta h_1}$ and r_h describe the number of points for δh_1 and h analogously. That is, if $\delta h_0 = 10$, then we use 21 points to discretize δh_0. The ACAS X report [KC11] uses resolution $(10, 10, 10)$.

Controller deployment. For controller deployment, we select (based on [KC11]) the weighted voting scheme (see Section 4.4, Page 133).

In Figure 5.1 we illustrate a part of the interpolated strategy generated according to [KC11]. On the horizontal axis we denote the time until collision, running from zero (on the left hand side of the plot) to 40 seconds (on the right hand side of the plot). On the vertical axis we denote the height difference between the two planes. Both climbing rates are zero, and the last advisory and the pilot state are `COC`. Note that `LHS` occurs at time 0 where, if two airplanes are less that 100 feet apart, an `NMAC` occurs. As we move towards the right, potential collisions are therefore less imminent. Such plots are easy to generate with our framework.

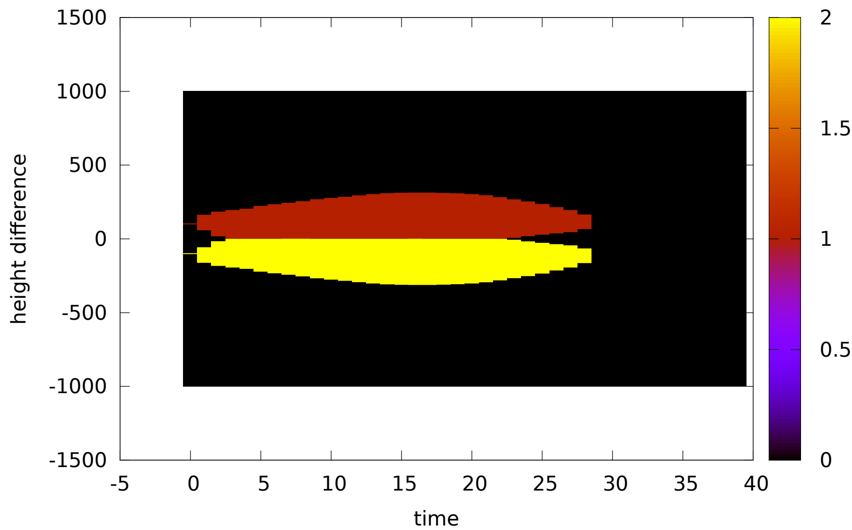


Figure 5.2: Controller generated resolution $(20, 20, 20)$ with weights as in [KC11]. x-axis shows time until `LHS`, y-axis height difference. Parameters δh_0 and δh_1 are zero throughout, and adv = ps = `COC`. Color indicates selected advisory: black (0) for `COC`, red (1) for `CLI1500`, yellow (2) for `DES1500`.

One way to intuitively understand these plots is to imagine the intruder on the left hand side of the plot at time and height zero, and our plane somewhere on the plot. The advisory is then determined by the color at the position of our plane. We want to emphasize that plots like these can only ever display a small slice of the whole state space. We have to fix both climbing rates and the pilot state in order to be able to generate a two dimensional plot.

The black area marks the part of the state space in which the controller advises `COC`. The red part marks the part of the state space in which the controller advises climbing, while the yellow part of the state space marks a descend advisory. Note the red zone, above the middle line, in which the controller advises to climb. It is the red and yellow zones in which the cost generated by the probability of an NMAC outweighs the cost of giving an advisory. In the black area above the red zone, the probability of an NMAC is not sufficiently high enough to outweigh the cost of issuing an alert.

We would like to point out two features of the generated controller. Firstly, if the airplanes start out on the same height, then the controller waits for a long time until giving an advisory, as witnessed by the black space between the two "tails" on the right. This is because it is very unlikely that the two planes will

remain on the same height for a long time (due to their random movement), and it is therefore better to wait until the intruder either starts climbing or descending and go in the opposite direction. Secondly notice the "mouth" shape close to time 0 and around height difference 0. In this collision situation, ACAS X is not giving any advisory, although one would intuitively expect that *some* advisory would be more informative to the pilot than a `COC`, which may be misleading. This is an artifact of the costs and rewards used for synthesis described above. Consider the state in which ps = adv = `COC`, $\delta h_0 = \delta h_1 = h = 0$ and time to `LHS` = 2 as an example. The framework evaluates all possible advisories it can give, which are `COC`, `DES1500` and `CLI1500` here. Intuitively, ACAS X should signal either `DES1500` or `CLI1500` to avoid a collision. But even if an advisory was given, a collision would still be very likely because of the lack of time for an effect to happen. Therefore, ACAS X expects to receive the costs of an NMAC with high probability even if an advisory is given. Let us call the expected cost incurred due to an NMAC if no advisory is given c_0, and the expected cost incurred due to an NMAC if an advisory is given c_1 (it does not matter which advisory since the situations are symmetric). Note that $c_1 < c_0$, since the probability of an NMAC is lower if an advisory is given. ACAS X will definitely receive the cost of an alert if it chooses to give one (i.e., `Alert`). If ACAS X gives an advisory it will incur costs $c_1 + \texttt{Alert}$, and if no advisory is given it will incur costs c_0. Due to the way costs are chosen, $c_1 + \texttt{Alert} > c_0$. Equivalently, $c_0 - c_1 < \texttt{Alert}$, i.e., the difference in the probability of an NMAC is not high enough to merit an advisory. We describe a technique that identifies situations like these in Section 5.3.

5.3 Verification

To complement the ACAS X work that primarily uses simulation, we apply formal analysis techniques to evaluate the ACAS X controller. Simulation-based techniques are studied and discussed in Section 5.4, where we explore the design-space of controllers and compare different generated controllers among themselves. In this section, we evaluate the ACAS X controller 1) in terms of the quality criteria used for its generation, and 2) through model checking of PCTL [HJ94] properties, which are ideal for probabilistic models such as ACAS X's. we discretize the continuous model with several different resolu-

tions for evaluation. We could even use different model characteristics and parameters (although we do not do the latter in the experiments presented here).

The type of analysis that we perform provides a value $v(s)$ for each state of the discretized model. To easily compare results of analyses with each other and with simulations, we define a probability distribution $I(s)$ over the states of the model. It is derived from the following continuous probability distribution defined in [KC11]. The only states considered are those at 40 seconds from LHS, and in which ps = adv = COC. Over those states, we define a continuous distribution over $(\delta h_0, \delta h_1, h) \in \mathbb{R}^3$ by sampling δh_0 and δh_1 uniformly from $[-1000, 1000]$ ft / min, denoted as $\delta h_0 \sim U(-1000, 1000)$ and $\delta h_1 \sim U(-1000, 1000)$. To make a collision likely, and therefore to provoke the controller into action, h is sampled from $N(40((\delta h_1 - \delta h_0)/60), 25)$, i.e., from a normal distribution centered at $40((\delta h_1 - \delta h_0)/60)$ with a standard deviation of 25.

To define an analogous distribution on the discretized state space D_R, we assign probability masses to all three parameters so as to soak up the probability of the space around them. For example, if the discretization uses $\{-2500, -2250, \ldots, 2250, 2500\}$ for δh_0 and δh_1 and $\{-1000, -900, \ldots, 900, 1000\}$ for h, then we assign to the points with $\delta h_0 = \delta h_1 = h = 0$ the probability mass of all states in which $h \in [-50, 50]$ and $\delta h_0, \delta h_1 \in [-125, 125]$.

That is, the probability of picking sample point δh_0 is defined as: $\mathrm{P}(\delta h_0 - \Delta_{\delta h_0}/2 \leq H_0 \leq \delta h_0 + \Delta_{\delta h_0}/2)$, where $H_0 \sim U(-1000, 1000)$ and $\Delta_{\delta h_0}$ is the distance between two discretization points of δh_0. We define the discretized probability of δh_1 analogously. The discretized probability of h is defined as: $\mathrm{P}(h - \Delta_h/2 \leq H \leq h + \Delta_h/2)$, where $H \sim 40((\delta h_1 - \delta h_0)/60) + N(0, 25)$, i.e., the probability distribution of h depends on δh_0 and δh_1. Here, Δ_h stands for the distance between two discretization points of h. We then use I to calculate the expected value $\mathbb{E}_{I(s)}[v(s)]$.

Influence of resolution on controller evaluation

Our first step in evaluating the ACAS X controller involves analyzing its performance in different resolutions. We picked $(10, 10, 20), \ldots, (10, 10, 50)$, $(20, 20, 10) \ldots (50, 50, 10)$ and $(20, 20, 20) \ldots (50, 50, 50)$ as values. For each of these resolutions, Figure 5.3 presents the evolution of the probability of seeing

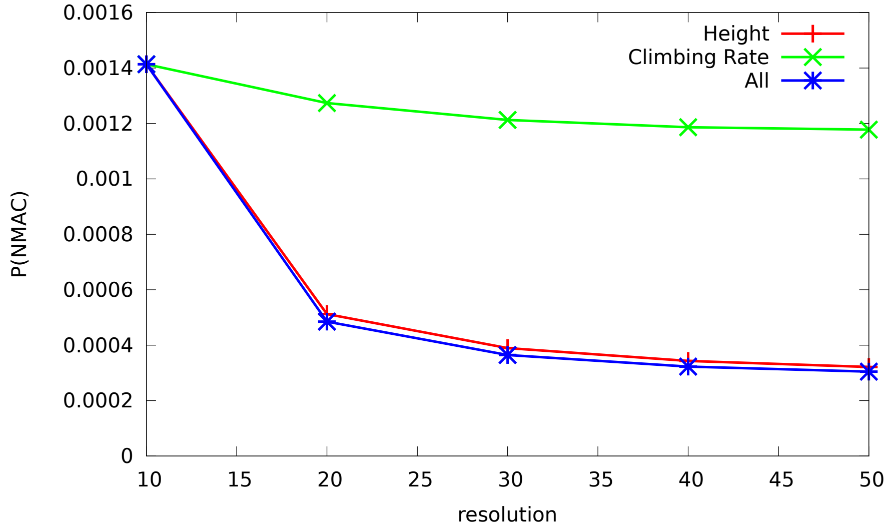


Figure 5.3: P(NMAC) of baseline controller in various resolutions

an NMAC versus the resolution. The three lines represent the three groups of increasing resolutions. Line "Height" represents resolutions $(10, 10, n)$, while line "Climbing Rate" represents the resolutions $(n, n, 10)$ and line "All" represents the resolutions (n, n, n), for $n \in \{10, 20, 30, 40, 50\}$.

These plots indicate that the probability of NMAC drops as we increase resolution. This in turn indicates (though does not guarantee) that a coarse resolution provides a conservative estimate for the quality criteria of the controller. Lines "Height" and "Climbing Rate" indicate that increasing the resolution of the height difference has a stronger influence on the quality of the analysis than the resolution of the climbing rate. This observation is reinforced by comparing lines "Height" and "All". The difference between these two lines is small, despite the fact that an n-fold increase in resolution of the climbing rate leads to an n^2-fold increase in state space.

PCTL model checking

The PCTL model checking engine that we have developed enables users to: (1) vary the resolution of the model to get more precise results, and (2) analyse non-trivial properties expressed in the PCTL formal property language. In contrast to simulation, PCTL model checking allows an exhaustive search of the state space and can thus uncover scenarios that simulations might easily

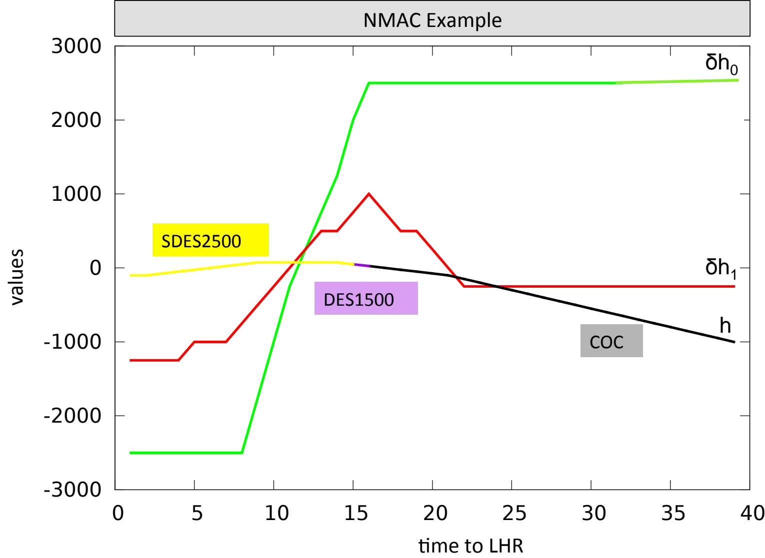


Figure 5.4: Trace plots for property 1. x-axis displays time to `LHS`, y-axis displays values of (δh_0, δh_1, h). The color of line h depicts the advisory, tagged above the line.

miss. This is important given the low probability of some of the properties we want to check.

Property 1: Near Mid-Air Collision.

Studies the probability of a near mid-air collision, formally $\mathrm{P}_{=?}[\mathsf{F}\,\mathtt{NMAC}]$. During analysis, we observed that the most likely cases of this undesirable scenario stem from late reactions from the pilot. We therefore decided to instead concentrate on NMACs that occur despite immediate reactions to advisories by the pilot. We formulate this as $\mathrm{P}_{=?}(\mathsf{F}\,\mathtt{NMAC} \mid \mathsf{G}\,\mathrm{adv} = \mathrm{ps})$, i.e., what is the probability of reaching an NMAC state although the pilot always reacts immediately.

The highest probability over all initial states that we encounter with the conditional probability formula is $2.30 \cdot 10^{-8}$, as opposed to $6.92 \cdot 10^{-4}$ with the original formula. This confirms that the vast majority of NMACs happen because the pilot does not react fast enough or at all. To understand the NMACs that occur despite the fact that the pilot reacts to advisories, we analyzed some traces that are most likely to fulfill $\mathrm{P}_{=?}(\mathsf{F}\,\mathtt{NMAC} \mid \mathsf{G}\,\mathrm{adv} = \mathrm{ps})$.

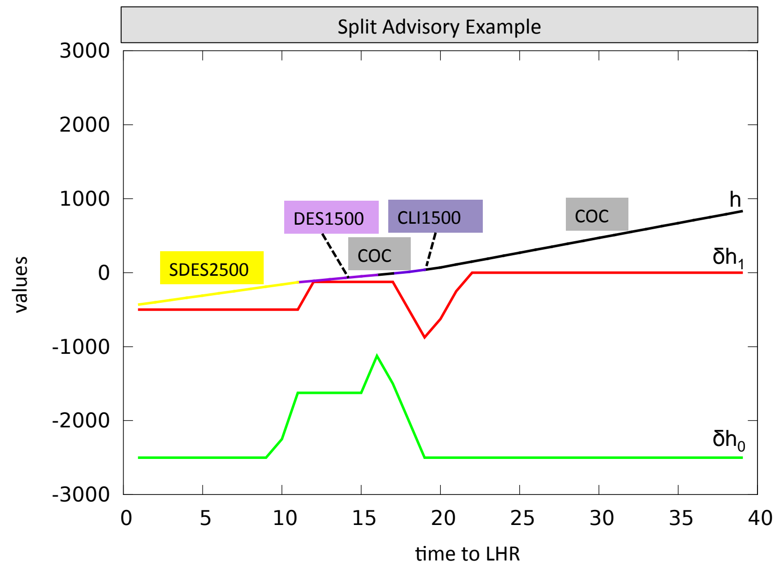


Figure 5.5: Trace plots for property 3. x-axis displays time to `LHS`, y-axis displays values of (δh_0, δh_1, h). The color of line h depicts the advisory, tagged above the line.

Figure 5.4 depicts such a scenario: initially, our airplane is 1000ft below the intruder and we are climbing with 2500 ft / min. The intruder, on the other hand, starts out with a climbing rate of -250 ft / min. Until 22 seconds to `LHS`, the two airplanes maintain their course, and therefore the height difference shrinks. If both planes were to continue to maintain their course, then our plane would be well above the intruder at time 0 to `LHS`, so ACAS X does not alert.

At this point, climbing rate of the intruder starts increasing, and the vertical distance becomes -150 ft. The height difference levels off as a result of the intruder's increase in climbing rate from now on. ACAS X signals the `DES1500` advisory seven seconds later, and `SDES2500` one second after that. As a result, our airplane starts descending steeply until it reaches -2500 ft / min. At the point of the first alarm, the vertical distance is 50 ft, i.e., our plane is slightly above the intruder. Unfortunately, the climbing rate of the intruder starts decreasing at exactly the same point and from that point on, the two climbing rates are not different enough to carry our plane outside of the danger zone

and we end up with a vertical distance of 100 ft, and hence an NMAC.

Traces like these capture exactly the type of unforeseen behaviour that led to the Überlingen accident [Joh04], and probabilistic model checking can detect cases like these easily. We consider it encouraging that the most likely case of collision requires relatively complex behaviour of the intruder (first increasing the climbing rate, then decreasing it, at exactly the right point in time).

Property 2: No advisory despite collision. Studies the probability of issuing no advisory although a future NMAC is likely, formally $\mathrm{P}_{=?}[\mathsf{F}(\mathrm{P}_{=1}[\mathsf{X}\,\mathtt{COC}] \wedge \mathrm{P}_{>0.1}[\mathsf{F}\,\mathtt{NMAC}])]$. This formula was motivated by our previous observation of Figure 5.1 in Section 5.2, according to which there is an area where ACAS X issues no advisory although an NMAC is imminent. Figure 5.6 shows the probability of the formula for all states in which $\delta h_0 = \delta h_1 = 0\,\mathsf{ft}\,/\,\mathsf{min}$ and $\mathrm{adv} = \mathrm{ps} = \mathtt{COC}$. This probability is 1 until about 12 seconds away if the height difference between the planes is less than a 100 ft. Model checking the formula, however, reveals that among all initial states, the highest probability is 0.3%, so getting into such a situation is improbable.

Property 3: Split Advisory. Studies the probability of issuing an alert, switching it off, and then switching an alert on again (a *split advisory*), formally $\mathrm{P}_{=?}[\mathsf{F}(\neg\,\mathtt{COC} \wedge P_{=1}[\mathsf{X}\,\mathtt{COC}] \wedge P_{>0}[\mathsf{F}\,\neg\,\mathtt{COC}])]$. Even though during controller generation ACAS X penalizes reversals, these costs only reflect immediate changes in controller advisories. Split advisories are also undesirable, but are harder to capture during controller generation. The PCTL property described above can however be used to study how likely such situations are. Analysis of the model checking results revealed that a main cause for such situations is the pilot not following the advisory. We therefore refined the property similarly to Property 1, by checking cases where split advisories occur under the condition that the pilot always reacts immediately to advisories.

Figure 5.5 depicts a split advisory scenario under the refined property. Initially (at 40 seconds to `LHS`), our plane is 830 ft above the intruder and descending with 2500 ft / min, while the intruder is in level flight. The vertical distance is therefore decreasing. Around 19 seconds into the scenario, the intruder starts descending, and soon after, ACAS X advises `CLI1500` and maintains this advisory for 2 seconds, before switching it off again. Accordingly, the rate of descent of our plane gradually reduces to 1500 ft / min. The advisory is then switched off, as the intruder stops descending, effectively moving out of the way of our

plane. ACAS X switches to COC but, a second later, gives advisories DES1500, followed by SDES2500, as the intruder's rate of descent increases again.

Let us further analyze this generated scenario. The first climb advisory aimed at avoiding a collision that would be likely if our plane continued to descend at the same rate. It could not force the pilot to increase the rate of descend further, since 2500 ft / min already is the maximum. Therefore, climbing was the only possibility. Then the intruder stopped descending, which reduced the probability of colliding with our current climbing rate. This may have caused ACAS X to shut the advisory off. Shortly before ACAS X switched the advisory back on, the difference in climbing rates was 1000 ft / min, and the height difference was -30 ft. Since we were about 15 seconds away from LHS, this amounted to a decreased vertical distance of about 260 ft. ACAS X decided to increase the vertical distance by increasing the rate of descent.

It would be interesting to study whether the cost function of ACAS X may encourage such cases of split advisories. Given that ($\texttt{Alert} + \texttt{COC} < \texttt{Reversal}$), it is possible that ACAS X decided to gain a small reward for selecting COC after the first advisory, and additionally avoid the cost of a reversal that would be incurred if the advisory was switched directly from a climb to a descend.

5.4 ACAS X design challenges

The generation of the ACAS X controller depends on two major design issues that have so far been unexplored: the selection of weights, and the discretization resolution. As reported in [KC11], the weights were selected based on an intuition of the relative importance of the different quality criteria. In this section, we study more systematic techniques for selecting controller weights, and investigate how discretization resolution influences the generated controller.

Generating controller weights

Our goal is to systematically explore deterministic controllers whose performances exceed requirements on NMAC, Alert, etc., provided by domain or certification experts. We approximate the Pareto curve of the model towards this goal.

Figure 5.7 presents a subset of the points generated by this approach on Alert and NMAC exclusively. The target point and the box it defines are plotted

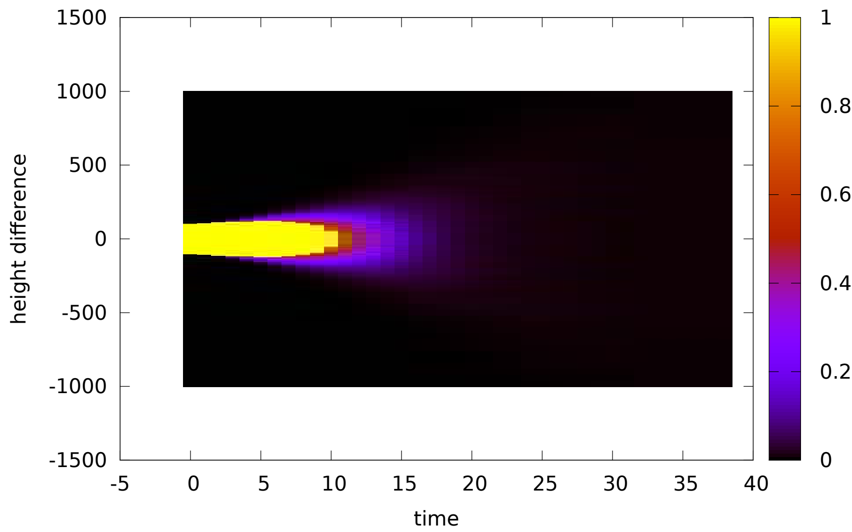


Figure 5.6: Probability of fulfilling property 2. Plot parameters as in Figure 5.1; color depicts probability

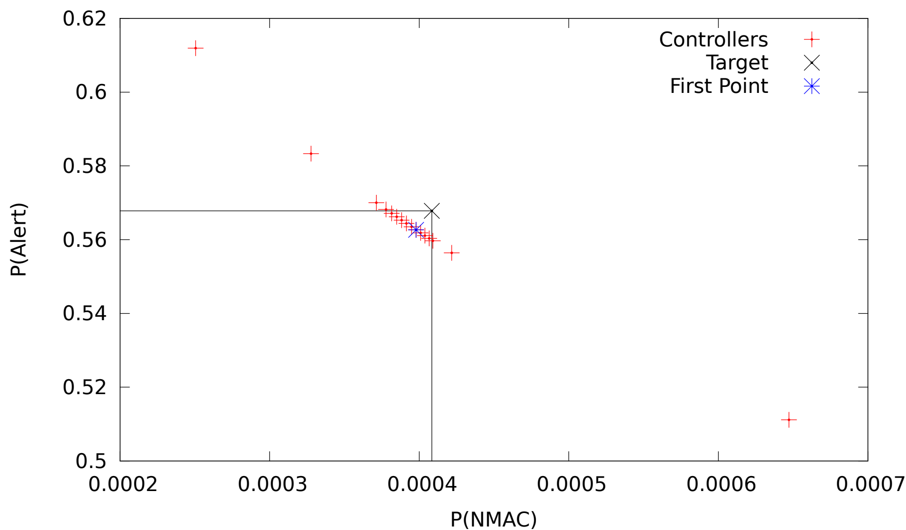


Figure 5.7: Points generated for two objectives.

in black, and the points generated are plotted in red. The algorithm first generated 8 points outside the box. The first point generated within the target box (the 9th overall) is depicted in blue. We generated 10 more points after we found it. We note that all subsequent 10 points that are generated also lie within the box. The same effect has been observed for three dimensions. We

conclude that this algorithm is good at approximating the interesting part of the Pareto front (that inside the box) once it finds the first point that meets the target specifications.

We have checked this algorithm against various targets, and it always either finds a controller meeting the requirement, or proves that no such controller exists. Note that finding a controller in the box is an NP-complete problem (easy adaptation of proof from [Cha07]). In the worst case, the algorithm has to generate all points of the Pareto front of the model, of which there are exponentially many. However, as the next section shows, little more than 100 points suffice to find a controller meeting the requirement for various resolutions.

We believe that this technique can be very helpful as the controller model ACAS X evolves. Each evolution (be it a change in discretization or a change in parameters), necessitates tuning weights anew (as witnessed by the first experiment in the next section). Our approach allows us to semi-automatically select these weights by presenting domain experts with the trade-offs. They can then select a controller they deem sufficient, or select an area for further refinement.

Discretization resolution

To study the effects of discretization resolution on the quality of the obtained controller, we designed a number of experiments described in this section. We will from now on refer to the controller presented in [KC11] as the "baseline" controller.

Experiment 1. This experiment aims to analyze the performance of controllers generated at resolutions $(20, 20, 20)$, $(30, 30, 30)$, $(40, 40, 40)$ and $(50, 50, 50)$, using the weights of the baseline controller. Our expectation was that a higher resolution would lead to a better performance, at least in terms of P(`NMAC`). However, the experiments showed that the controllers we generate by this method do not necessarily perform better in all the quality attributes. Instead, higher resolution controllers have a significantly higher P(`NMAC`) and significantly fewer alerts than the baseline controller in the same resolutions.

The reason becomes clear when we consider the controller plots in Figure 5.1 and Figure 5.2. The area in which an alert is signalled by the controller is significantly smaller in Figure 5.2 when compared to Figure 5.1. To understand the reason for this effect, we analyzed the controllers using the techniques from

Section 5.3. It turns out that controllers in higher resolutions indeed perform better in the sense of having a higher expected reward than the baseline controller. Intuitively, the controllers use the additional information they receive from a higher resolution to improve the score they receive. To this end, the controllers improve their score by reducing the expected number of alerts, at the cost of a higher P(NMAC).

This experiment made it clear to us that weights may balance out the quality attributes of a controller differently, when different resolutions are considered. As a consequence, we believe that it is more meaningful to systematically explore the design space of controllers based on specific target quality attributes, as presented in Section 5.4. One could then compute weights based on these target values, and within the resolution where the generation will occur.

Experiment 2. Given the first experiment, we decided to study whether it is possible to generate controllers that are better than the baseline controller in all quality attributes, in higher resolutions. To generate a controller that performs better than the baseline controller in a given resolution $R = (r_h, r_{\delta h_0}, r_{\delta h_1})$, we first evaluate the performance of the baseline controller in resolution R. The result is a vector $v = (\texttt{NMAC}, \texttt{Alert}, \texttt{Strengthening}, \texttt{Reversal}, \texttt{NMAC})$, which summarizes the performance of the baseline controller when model checked in resolution R (see Section 5.3 for more details). We then use the technique described above to approximate the Pareto front above v. From the generated controllers that meet the specification, we then pick the one with the lowest P(NMAC).

Figure 5.8 illustrates the obtained results. The bars show, for resolution factor n the performance of the baseline controller when checked against resolutions $(n, n, 10)$ (Climbing Rate), $(10, 10, n)$ (Height) and (n, n, n) (All) respectively. It can be seen that we were almost unable to decrease P(NMAC) using the climbing rate alone. The relative performance of these controllers is consistently around 99.5%. When we increase the resolution of the height, then we get a relative performance of about 85%. Finally, when increasing the resolution of both we see a relative performance of about 83%. As witnessed in Section 5.3, the discretization of height seems to have the biggest influence on controller quality. Interestingly, the relative performance does not improve as we increase the resolution.

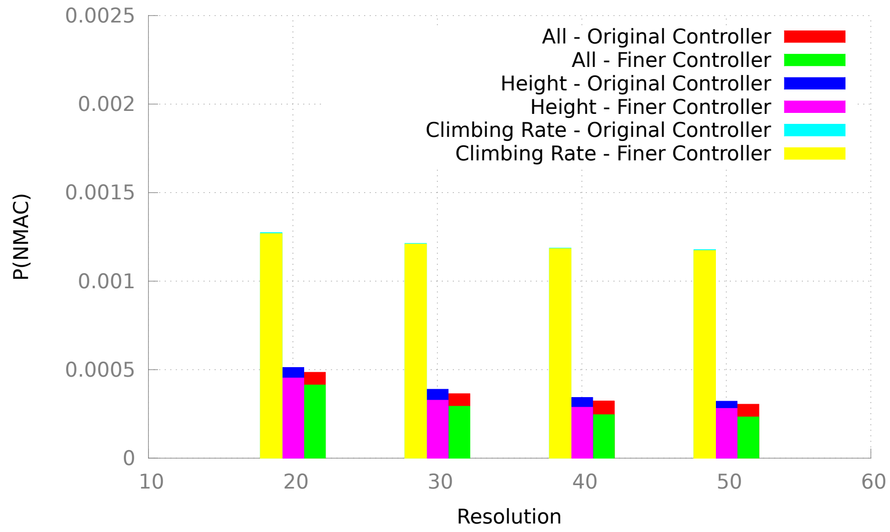


Figure 5.8: Controller quality vs resolution.

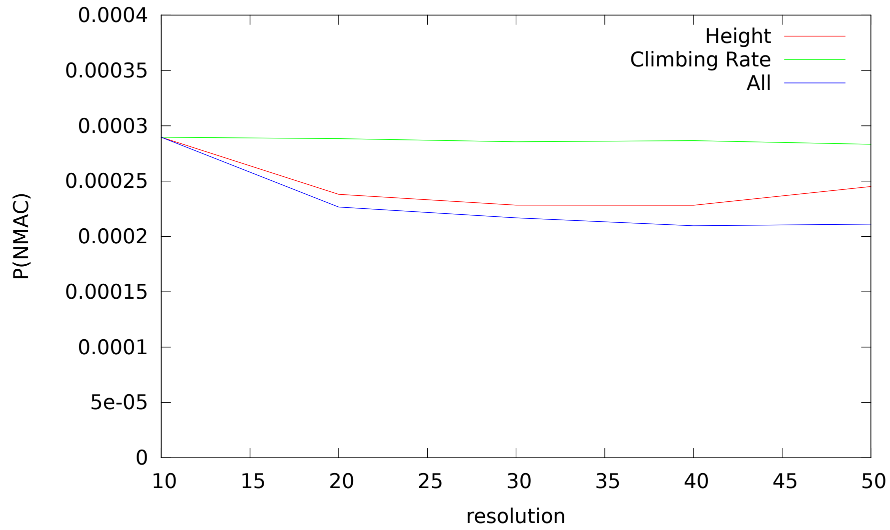


Figure 5.9: Controller quality checked in (50,50,100).

To further judge the quality of the generated controllers, we checked them against resolution $(50, 50, 100)$ and present the results in Figure 5.9. On the x-axis, we have the controller resolution, while on the y-axis we have the probability of a Near Mid-Air Collision. As before, “Height” stands for the controllers of resolution $(10, 10, n)$, “Climbing Rate” for the controllers of resolution $(n, n, 10)$ and “All” for the controllers of resolution (n, n, n). This experiment confirms that increasing the resolution of the height difference between the two planes

has the most impact up to and including $(10, 10, 30)$, after which we notice no further improvement. In contrast to this, we notice further improvements in category "All". Our experiments indicate that the best ratio of resolution for the three parameters is $(n, n, 3 \cdot n)$.

Experiment 3. Let $v_R(c)$ denote the quality vector of a controller c in resolution R (i.e., the vector of $\mathrm{P}(\mathtt{NMAC}), \mathrm{P}(\mathtt{Alert}), etc$). We organized this experiment to study if $\forall c_1, c_2, R_1, R_2 : v_{R_1}(c_1) \geq v_{R_1}(c_2) \wedge R_2 > R_1 \implies v_{R_2}(c_1) \geq v_{R_2}(c_2)$ holds. To this end, we compared the performance of the controller we generated in resolution $(20, 20, 20)$ to the baseline controller in resolutions $(20, 20, 20)$ and $(50, 50, 100)$, and present the results in the following table. Note that the higher resolution controller performs better than the baseline in all dimensions in resolution $(20, 20, 20)$; specifically, it is very close to the target performance in everything but NMAC, where it is notably better.

	NMAC	Alert	Strength.	Reversal	COC
$(10, 10, 10)$ in $(20, 20, 20)$	$-4.850 \cdot 10^{-4}$	-0.6310	-0.083	-0.019	0.629
$(20, 20, 20)$ in $(20, 20, 20)$	$-4.186 \cdot 10^{-4}$	-0.6306	-0.081	-0.019	0.631
$(10, 10, 10)$ in $(50, 50, 100)$	$-2.897 \cdot 10^{-4}$	-0.6245	-0.078	-0.020	0.622
$(20, 20, 20)$ in $(50, 50, 100)$	$-2.313 \cdot 10^{-4}$	-0.6308	-0.078	-0.019	0.630

Table 5.1: Performances of the baseline controller (resolution $(10, 10, 10)$ and controller generated in resolution $(20, 20, 20)$ with the performance of the baseline controller as target (see Section 4.5). Analysis resolution is $(50, 50, 100)$.

This attests to the efficacy of our Pareto front algorithm. When comparing this to the analysis results in resolution $(50, 50, 100)$, we observe that while the higher resolution controller and the baseline controller are still very close in all characteristics except NMAC, the higher resolution controller is no longer strictly better in all dimensions. For example, it uses slightly more alerts and slightly more reversals. This is offset by the fact that the $\mathrm{P}(\mathtt{NMAC})$ of the higher resolution controller is still significantly better than that of the baseline controller. To summarize, the general tendencies of the relation of controllers when checked in higher resolutions are the same, but the exact relations are not preserved.

Bayesian model checking

In this section, we evaluate the generated controllers using simulation (where discretization is not required), and compare the results with model checking. Our analysis reports that the probability of NMAC lies in range $[2.48 \cdot$

$10^{-4}, 2.58 \cdot 10^{-4}]$ with probability 95%. We generated 38'796'000 samples to reach this level of confidence for the given interval size [ZPC13].

We additionally applied this simulation technique to controllers of resolution $(10, 10, 10), \ldots, (10, 10, 50)$ generated previously. The following table presents the probability of seeing an NMAC for each of them.

Resolution	10	20	30	40	50
$\mathrm{P}(\mathtt{NMAC}) \cdot 10^4$	[2.51, 2.61]	[2.17, 2.27]	[2.08, 2.18]	[2.12, 2.22]	[2.27, 2.37]

Table 5.2: Probability of seeing an NMAC in controllers of resolution $(10, 10, 10)$ to $(10, 10, 50)$.

We conclude that the trend follows that depicted in Figure 5.8: improvements in performance are significant until we reach resolution $(10, 10, 30)$, at which point they taper off. We were unable to perform this analysis on controllers with resolution larger than $(20, 20, 20)$ because we could not fit the whole table into memory at once. For $(20, 20, 20)$, though, we receive $\mathrm{P}(\mathtt{NMAC}) \in [2.06 \cdot 10^{-4}, 2.16 \cdot 10^{-4}]$, i.e., a number very close to that of the controller generated for $(10, 10, 30)$.

Bayesian model checking and probabilistic model checking. Bayesian and probabilistic model checking have different strengths and weaknesses. On the one hand, Bayesian model checking allows us to use the continuous state space model. This avoids the discretization error probabilistic model checking introduces. It further allows us to employ a much more detailed model, although we have not done so in this study. For example, we can include more complicated pilot models or three dimensions.

On the other hand, probabilistic model checking calculates an exact (for its model) value for each state of the model. This can be more informative than the summarized information we receive from Bayesian model checking. It also avoids the uncertainty of confidence intervals. We can further employ some optimization techniques which allow us to avoid keeping the whole controller in memory. This becomes useful for high-detail controllers, where the table requires more memory than our test machine provides. This is the reason why we can check the controller generated, for example, for resolution $(30, 30, 30)$. This was not possible in our implementation of Bayesian model checking.

5.5 Implementation

We initially used existing probabilistic model checking tools for ACAS X but encountered several limitations. First, we could not express the linear interpolation needed in the controller evaluation. Second, we not only require capabilities for the specification of a model, but also for loading generated controllers for subsequent verification. Last but not least, for our mupliple experiments involving increasing resolution, the state spaces we generate grow prohibitively large, and there is a considerable slow-down that could benefit from parallelization, which is unavailable in current releases of existing tools.

More specifically, the size of the controller has $40 \cdot ((2r_{\delta h_0} + 1) \cdot (2r_{\delta h_1} + 1) \cdot (2r_h + 1) \cdot 13)$ states in resolution $(r_{\delta h_0}, r_{\delta h_1}, r_h)$. So, for example, the model from [KC11] has $4'815'720$ states overall. A controller with resolution $(50, 50, 50)$ has $535'756'520$ states. We wrote a simplified version of the model in [KC11] for PRISM [KNP11] (without linear interpolation, but with sigma point sampling). While PRISM succeeded in loading the model as a BDD model, analyzing it was not possible (we aborted conversion to the hybrid representation after 10 minutes).

These problems motivated us to create our own framework in Chapter 4 that uses traditional programming languages to describe models, and takes advantage of two key insights into the ACAS X model. Firstly, if we want to calculate the values of any property in this model at time t, then we only need to keep the value of time $t - 1$ in memory. This alone leads to a reduction of memory consumption to 2.5%. Secondly, since we need to calculate value iteration steps only a relatively small number of times for each state, it is possible to avoid storing the transition matrix in memory and generate the values on-demand.

In addition, we parallelized value iteration, and the speed-up obtained in experiments using up to 12 cores was almost linear (1.94 for 2 cores, 3.37 for 4 cores, 4.67 for 6 cores, 6.47 for 8 cores, 7.54 for 10 cores, 8.93 for 12 cores). Parallelization proved essential for our experiments involving increasing discretization resolution; generating the Pareto fronts for all cases took about 2 days, as opposed to more than a month.

5.6 Conclusions and Future Work

ACAS X is a safety-critical system that the FAA plans on introducing as the new standard for collision avoidance. The system that will be deployed is the look-up table that is generated by the techniques described in [KC11]. It is therefore reasonable that a large number of the verification efforts would focus on the verification of the generated controller in operation. However, we believe that it is meaningful to take advantage of the existence of models for additional formal analysis both of the controller itself, and of the design choices.

Our experiments related to the effects of resolution on controller generation were particularly interesting. For example, we observed that height discretization is more effective that climbing rate alone, when exploring the space of controllers better than a particular target. We therefore recommend increasing height resolution first, when there is an upper bound in controller size that does not allow for uniform discretization of all variables. In the future, we intend to carry out more experiments in this domain in order to give more precise recommendations.

Some of the results that we obtained were also unexpected: the fact that a higher resolution may balance the weights of quality attributes differently and therefore result in a drop in performance of NMAC; or the fact that the relative performance of two controllers may change when moving to higher resolutions. This cautions us, in exploring the space of controllers, to ultimately evaluate their relative performance in simulation. However, the Pareto-front-based techniques for controller generation provide a systematic way of generating and comparing controllers that can complement designer intuition.

PCTL model checking also proves valuable in studying properties of generated controllers. However, more useful than the model checking itself, is the capability to visualize its results and generate traces that help with understanding of the model checking results. We therefore found that latter aspect of our tools most helpful, together with a simulator that we built, which allows us to interactively explore generated controllers. In the future, we plan to connect the simulator to the model checker, to allow replay of the generated traces.

6 Conclusion

This thesis has contributed to the theoretical and practical aspects of quantitative verification and synthesis. We have added theoretical results and algorithms to rewards for single objective Markov decision processes. This field seems well studied now, with single rewards and combinations of two rewards well covered and practical algorithms available. We have also provided an algorithm for approximating the Pareto curve of many objectives.

We have shown that the idea of adding quantitative information to a qualitative does not necessarily mean using rewards or probabilities, but can also mean other information like closeness between original and repaired program. Here we have not only contributed to a reformulation of the program repair problem, but we have also analysed the limits of this reformation and showed where this reformulation makes program repair impossible. We have studied the reason for the impossibilities and presented two alternative formulations that might lead to success where the other approaches might fail. The usefulness of this approach was shown with several examples taken from classical model checking scenarios. Our prototype implementation showed that a reformulation was necessary, that it was useful and that it might contribute to faster repair by the way of partial specifications. More has to be done to make this reformulation practical. On the one hand, it seems promising to use techniques based on sequential equivalence checking to speed up the repair search process. On the other hand, it seems promising to marry our approach to partial program synthesis frameworks to extend it to real programs. In this area, our approach is the first to show the following difference between program repair from partial program synthesis, which hitherto were the same. Partial program synthesis is the term given to the process of filling in missing details in a partial program. Program repair as defined by us means modifying an existing program while leaving as many traces unchanged as possible.

This thesis took a more practical turn then. It took the known results and algorithms of verification and synthesis and forged them into a single framework. This was inspired by the need of verification of synthesized controllers. We showed that it is useful to check synthesized controllers, even though they are correct by construction. On the one hand, abstractions might have been employed during controller generation. We want to check the correctness of the abstraction or check whether the error introduced by abstraction is within bounds. On the other hand, we might want to check the controller against modeling errors, i.e,. how robust is it when the assumptions made during modeling do not hold? The novelty of this framework consists in the idea of targeting synthesis and verification at the same time such that the two can be used in a loop. This allows us to synthesize a controller under one condition and then verify it in different conditions. We showed by the way of several case studies that quantitative verification and synthesis techniques developed in our field can be useful for tasks formal methods do not commonly address. We showed that the feed-back loop approach of our framework is indeed useful for controller validation. On the performance side, we showed that the algorithms are easily parallelizable, and that performance is crucial, especially when controllers need to be generated many times. The case studies taught us that Pareto curves are important in the settings we looked at, because reward-based synthesis often involves trade-offs. Systematic exploration of the space of trade-offs and visual inspection proved very useful in our case studies.

Using an embedded domain specific language based on Java showed that we can use Java to write probabilistic programs, and that these can be used as a model for embedded control systems. The case studies showed that the EDSL provides a natural way of writing models that is accessible to everybody who has written a Java program.

For now, our framework only provides algorithms for purely quantitative objectives, and it would be useful to add, for example, synthesis for mean-payoff parity conditions, or for mean-payoff LTL conditions.

Application of our new framework to a real-world case study was a very exciting part of this thesis. It showed us that the techniques we study and develop are interesting not only from a research perspective, but also address pressing safety concerns. We showed how our framework can contribute to the exploration of the design space of an airborne collision avoidance system, and

how PCTL, a formalism classically used in our field, can be used outside of our field. We hope that this case study leads to a more practical orientation of quantitative verification and synthesis.

This thesis has contributed to theoretical and practical aspects of quantitative verification and synthesis. Besides using it as a basis to reformulating program repair, we have shown that quantitative synthesis can be practical in more than synthesizing automata. We have shown that pure quantitative synthesis is useful already, and that it provides very sensible controllers. We have shown that the same framework can and should be used for both synthesis and verification. Especially the successful application of this framework to a real world case study lends this idea weight.

6.1 Future work

While this thesis provides a number of advances to its field of study and even reformulation of a known problem, a large number of open and interesting questions remain. We already discussed some at the end of the respective chapters, but some general directions of future work remain.

Theoretical aspects. We have reformulated the program repair problem. It remains open how to find an appropriate lower bound of acceptable repairs. A related interesting problem is to find specifications that are good for repair, i.e., for which we can always find a lower bound. While we provided two formulations based on rewards, case studies of these remain open. Another interesting question is how to extend our idea of repairs without regrets to infinite state programs. An open and possibly very hard problem is the search for a polynomial time algorithm for games with parity conditions.

Practical aspects. To make the field of quantitative verification and synthesis interesting from a practical perspective, it seems necessary to make its core algorithms scale to large models. While BDDs were a means of choice for a long time, it seems that their limits have been reached. So we have to look for alternatives to beat the state space explosion program. On the one hand, alternative symbolic methods like antichains [WDHR06] have seen success in qualitative synthesis and verification, and so it can be hoped that they can be adapted to the quantitative setting. On the other hand, abstraction methods seem promising. Works like [DKP13] make a start for MDPs. In addition,

scalable distributed versions of all core algorithms are desirable.

Orientation. It seems desirable that quantitative verification and synthesis turn to practical aspects. A lot of theory has been produced, but is seems that practical example of synthesis are few and far in between. We have provided a start with Chapter 4 and more so in Chapter 5, and so hope that more attention will be given to these methods. Beside safety critical systems like ACAS X, an interesting field is the application of formal methods procedures to robotics scenarios. In turn to make these methods practical, it seems necessary to combine quantitative and qualitative aspects. So focused research into fast algorithms that produce synthesizable strategies that behave correct intuitively is necessary.

In addition, we have found that quantitative verification and synthesis can benefit from techniques from the field of operations research and artificial intelligence, and vice versa. More collaboration is desirable.

Bibliography

[BBC+11] Tomás Brázdil, Václav Brozek, Krishnendu Chatterjee, Vojtech Forejt, and Antonín Kucera. Two views on multiple mean-payoff objectives in markov decision processes. *CoRR*, abs/1104.3489, 2011.

[BBFR13] Aaron Bohy, Véronique Bruyère, Emmanuel Filiot, and Jean-François Raskin. Synthesis from ltl specifications with mean-payoff objectives. In Nir Piterman and Scott A. Smolka, editors, *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 169–184. Springer, 2013.

[BCD+11] Lubos Brim, Jakub Chaloupka, Laurent Doyen, Raffaella Gentilini, and Jean-François Raskin. Faster algorithms for mean-payoff games. *Formal Methods in System Design*, 38(2):97–118, 2011.

[BCHJ09] Rodric. Bloem, Krishnendu. Chatterjee, Thomas. A. Henzinger, and Barbara Jobstmann. Better quality in synthesis through quantitative objectives. In *CAV*, pages 140–156, 2009.

[BCJ14] Roderick Bloem, Krishnendu Chatterjee, and Barbara Jobstmann. *Handbook of Model Checking*, chapter Graph games and reactive synthesis. Springer, 2014.

[BCM+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^20 states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[BCW+10] Dietmar Berwanger, Krishnendu Chatterjee, Martin De Wulf, Laurent Doyen, and Thomas A. Henzinger. Strategy construction for parity games with imperfect information. *Inf. Comput.*, 208(10):1206–1220, 2010.

[BDL+06] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Håkansson, Paul Pettersson, Wang Yi, and Martijn Hendriks. Uppaal 4.0. In *QEST*, pages 125–126. IEEE Computer Society, 2006.

[BEGL99] Francesco Buccafurri, Thomas Eiter, Georg Gottlob, and Nicola Leone. Enhancing model checking in verification by ai techniques. *Artif. Intell.*, 112(1-2):57–104, 1999.

[BFL+08] Patricia Bouyer, Ulrich Fahrenberg, Kim Guldstrand Larsen, Nicolas Markey, and Jirí Srba. Infinite runs in weighted timed automata with energy constraints. In Franck Cassez and Claude Jard, editors, *FORMATS*, volume 5215 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2008.

[BFRR13] Véronique Bruyère, Emmanuel Filiot, Mickael Randour, and Jean-François Raskin. Meet your expectations with guarantees: Beyond worst-case synthesis in quantitative games. *CoRR*, abs/1309.5439, 2013.

[BGHJ09] Roderick Bloem, Karin Greimel, Thomas A. Henzinger, and Barbara Jobstmann. Synthesizing robust systems. In *FMCAD*, pages 85–92. IEEE, 2009.

[BH08] L. F. Bertuccelli and J. P. How. Robust Markov decision processes using sigma point sampling. In *American Control Conference (ACC)*, pages 5003–5008, 11-13 June 2008.

[BHHK03] Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, and Joost-Pieter Katoen. Model-checking algorithms for continuous-time markov chains. *IEEE Trans. Software Eng.*, 29(6):524–541, 2003.

[BHP97] Bruce Bukiet, Elliotte Rusty Harold, and José Luis Palacios. A markov chain approach to baseball. *Operations Research*, 45(1):14–23, 1997.

[BJP+12] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. *J. Comput. Syst. Sci.*, 78(3):911–938, 2012.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[BL69] J. R. Büchi and L. H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138:295–311, 1969.

[BN11] Nels E. Beckman and Aditya V. Nori. Probabilistic, modular and scalable inference of typestate specifications. In Mary W. Hall and David A. Padua, editors, *PLDI*, pages 211–221. ACM, 2011.

[BNR03] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In Alex Aiken and Greg Morrisett, editors, *POPL*, pages 97–105. ACM, 2003.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[CBGK08] Frank Ciesinski, Christel Baier, Marcus Größer, and Joachim Klein. Reduction techniques for model checking markov decision processes. In *QEST* [DBL08], pages 45–54.

[CCG+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *CAV*, 2002.

[CD11] Krishnendu Chatterjee and Laurent Doyen. Energy and mean-payoff parity markov decision processes. In Filip Murlak and Piotr Sankowski, editors, *MFCS*, volume 6907 of *Lecture Notes in Computer Science*, pages 206–218. Springer, 2011.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs*, pages 52–71, 1981.

[CGMZ95] Edmund M. Clarke, Orna Grumberg, Kenneth L. McMillan, and Xudong Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *DAC*, pages 427–432, 1995.

[Cha07] Krishnendu Chatterjee. Markov decision processes with multiple long-run average objectives. In Vikraman Arvind and Sanjiva Prasad, editors, *FSTTCS*, volume 4855 of *Lecture Notes in Computer Science*, pages 473–484. Springer, 2007.

[Cha11] Krishnendu Chatterjee. Graph games with reachability objectives - (invited talk). In Giorgio Delzanno and Igor Potapov, editors, *RP*, volume 6945 of *Lecture Notes in Computer Science*, page 1. Springer, 2011.

[CHJ05] Krishnendu Chatterjee, Thomas A. Henzinger, and Marcin Jurdzinski. Mean-payoff parity games. In *LICS*, pages 178–187. IEEE Computer Society, 2005.

[CHJS10] Krishnendu Chatterjee, Thomas A. Henzinger, Barbara Jobstmann, and Rohit Singh. Measuring and synthesizing systems in probabilistic environments. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 380–395. Springer, 2010.

[CMB08] Kai-Hui Chang, Igor L. Markov, and Valeria Bertacco. Fixing design errors with counterexamples and resynthesis. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(1):184–188, 2008.

[CMH06] Krishnendu Chatterjee, Rupak Majumdar, and Thomas A. Henzinger. Markov decision processes with multiple objectives. In Bruno Durand and Wolfgang Thomas, editors, *STACS*, volume 3884 of *Lecture Notes in Computer Science*, pages 325–336. Springer, 2006.

[CN06] Wai Ki Ching and Michael K. Ng. *Markov chains : models, algorithms and applications*. International series in operations research & management science. Springer, New York, 2006.

[CRR12] Krishnendu Chatterjee, Mickael Randour, and Jean-François Raskin. Strategy synthesis for multi-dimensional quantitative objectives. *CoRR*, abs/1201.5073, 2012.

[CTBB11] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. In *ICSE 2011*, pages 121–130, New York, NY, USA, 2011. ACM.

[dA97] Luca de Alfaro. *Formal Verification of Probabilistic Systems.* PhD thesis, Stanford University, 1997.

[DAT10] Cherki Daoui, Mohammed Abbad, and Mohamed Tkiouat. Exact decomposition approaches for markov decision processes: A survey. *Adv. Operations Research*, 2010, 2010.

[DBL08] *Fifth International Conference on the Quantitative Evaluaiton of Systems (QEST 2008), 14-17 September 2008, Saint-Malo, France*. IEEE Computer Society, 2008.

[Der62] Cyrus Derman. On sequential decisions and markov chains. *Management Science*, 9(1):16–24, 1962.

[DH94] Doron Drusinsky and David Harel. On the power of bounded concurrency i: Finite automata. *J. ACM*, 41(3):517–539, 1994.

[DKP13] Christian Dehnert, Joost-Pieter Katoen, and David Parker. Smt-based bisimulation minimisation of markov models. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *VMCAI*, volume 7737 of *Lecture Notes in Computer Science*, pages 28–47. Springer, 2013.

[EJ88] E. Allen Emerson and Charanjit S. Jutla. The complexity of tree automata and logics of programs (extended abstract). In *FOCS*, pages 328–337. IEEE Computer Society, 1988.

[EKB05] Ali Ebnenasir, Sandeep S. Kulkarni, and Borzoo Bonakdarpour. Revising unity programs: Possibilities and limitations. In *OPODIS*, pages 275–290, 2005.

[ELLL01] Stefan Edelkamp, Alberto Lluch-Lafuente, and Stefan Leue. Trail-directed model checking. *Electr. Notes Theor. Comput. Sci.*, 55(3):343–356, 2001.

[EM79] A. Ehrenfeucht and J. Mycielski. Positional strategies for mean payoff games. *International Journal of Game Theory*, 8(2):109–113, 1979.

[FGL12] Paolo Felli, Giuseppe De Giacomo, and Alessio Lomuscio. Synthesizing agent protocols from ltl specifications against multiple partially-observable environments. In Gerhard Brewka, Thomas Eiter, and Sheila A. McIlraith, editors, *KR*. AAAI Press, 2012.

[FKN+11] Vojtech Forejt, Marta Z. Kwiatkowska, Gethin Norman, David Parker, and Hongyang Qu. Quantitative multi-objective verification for probabilistic systems. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 112–127. Springer, 2011.

[FKP12] V. Forejt, M. Kwiatkowska, and D. Parker. Pareto curves for probabilistic model checking. In S. Chakraborty and M. Mukund, editors, *Proc. 10th International Symposium on Automated Technology for Verification and Analysis (ATVA'12)*, volume 7561 of *LNCS*, pages 317–332. Springer, 2012.

[FMY97] Masahiro Fujita, Patrick C. McGeer, and Jerry Chih-Yuan Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10(2/3):149–169, 1997.

[GBC06] Andreas Griesmayer, Roderick Bloem, and Byron Cook. Repair of boolean programs with an application to c. In Thomas Ball and Robert Jones, editors, *CAV*, volume 4144 of *LNCS*, pages 358–371. Springer, 2006.

[GBJV08] Karin Greimel, Roderick Bloem, Barbara Jobstmann, and Moshe Vardi. Open implication. In *ICALP*, pages 361–372, 2008. LNCS 5126.

[Gim07] Hugo Gimbert. Pure stationary optimal strategies in markov decision processes. In *STACS'07*, pages 200–211, Berlin, Heidelberg, 2007. Springer-Verlag.

[GKO+08] C. Grover, I. Knight, F. Okoro, I. Simmons, G. Couper, P. Massie, and B. Smith. Automated emergency brake systems: Technical requirements, costs and benefits, 2008.

[GV03] Alex Groce and Willem Visser. What went wrong: Explaining counterexamples. In Thomas Ball and Sriram K. Rajamani, editors, *SPIN*, volume 2648 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2003.

[Hah12] Ernst Moritz Hahn. *Model Checking Stochastic Hybrid Systems.* Doctoral dissertation, Universität des Saarlandes, Saarbrücken, 12/2012 2012.

[Hav98] Boudewijn R. Haverkort. *Performance of computer communication systems - a model-based approach.* Wiley, 1998.

[HJ94] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6:102–111, 1994.

[HMPS96] Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Markovian analysis of large finite state machines. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 15(12):1479–1493, 1996.

[HvdHvR09] Koen V. Hindriks, Wiebe van der Hoek, and M. Birna van Riemsdijk. Agent programming with temporally extended goals. In Carles Sierra, Cristiano Castelfranchi, Keith S. Decker, and Jaime Simão Sichman, editors, *AAMAS (1)*, pages 137–144. IFAAMAS, 2009.

[JGB05] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 226–238. Springer, 2005.

[JM06] M. U. Janjua and A. Mycroft. Automatic correction to safety violations in programs. Thread Verification (TV'06), 2006. Unpublished.

[Joh04] CW Johnson. Final report: review of the BFU Überlingen accident report. *Contract C/1.369/HQ/SS/04 to Eurocontrol, "http://www. dcs. gla. ac. uk/˜ johnson/Eurocontrol/Ueberlingen/Ueberlingen_Final_Report. PDF"*, 2004.

[JRS04] HoonSang Jin, Kavita Ravi, and Fabio Somenzi. Fate and free will in error traces. *STTT*, 6(2):102–116, 2004.

[JSGB12] Barbara Jobstmann, Stefan Staber, Andreas Griesmayer, and Roderick Bloem. Finding and fixing faults. *J. Comput. Syst. Sci.*, 78(2):441–460, 2012.

[JU04] S. J. Julier and J. K. Uhlmann. Unscented filtering and nonlinear estimation. *Proceedings of the IEEE*, 92(3):401–422, March 2004.

[KC11] Mykel J. Kochenderfer and James P. Chryssanthacopoulos. Robust airborne collision avoidance through dynamic programming. Project Report ATC-371, Massachusetts Institute of Technology, Lincoln Laboratory, 2011.

[KD07] JE Kuchar and Ann C Drumm. The traffic alert and collision avoidance system. *Lincoln Laboratory Journal*, 16(2):277, 2007.

[KGFP09] Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. Temporal-logic-based reactive mission and motion planning. *IEEE Transactions on Robotics*, 25(6):1370–1381, 2009.

[KKNP10] Mark Kattenbelt, Marta Z. Kwiatkowska, Gethin Norman, and David Parker. A game-based abstraction-refinement framework for markov decision processes. *Formal Methods in System Design*, 36(3):246–280, 2010.

[KLD+02] Seungchan Kim, Huai Li, Edward R. Dougherty, Nanwei Cao, Yidong Chen, Michael Bittner, and Edward B. Suh. Can markov chain models mimic biological regulation?, 2002.

[KMKH01] Zurab Khasidashvili, John Moondanos, Daher Kaiss, and Ziyad Hanna. An enhanced cut-points algorithm in formal equivalence verification. In *HLDVT*, pages 171–176, 2001.

[KNP02] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Probabilistic symbolic model checking with prism: A hybrid approach. In Joost-Pieter Katoen and Perdita Stevens, editors, *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 2002.

[KNP07] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Stochastic model checking. In Marco Bernardo and Jane Hillston, editors, *SFM*, volume 4486 of *Lecture Notes in Computer Science*, pages 220–270. Springer, 2007.

[KNP09] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Prism: probabilistic model checking for performance and reliability analysis. *SIGMETRICS Performance Evaluation Review*, 36(4):40–45, 2009.

[KNP11] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[KNPS06] Marta Z. Kwiatkowska, Gethin Norman, David Parker, and Jeremy Sproston. Performance analysis of probabilistic timed automata using digital clocks. *Formal Methods in System Design*, 29(1):33–78, 2006.

[KP13] Marta Z. Kwiatkowska and David Parker. Automated verification and strategy synthesis for probabilistic systems. In Dang Van Hung and Mizuhito Ogawa, editors, *ATVA*, volume 8172 of *Lecture Notes in Computer Science*, pages 5–22. Springer, 2013.

[KSHK07] Daher Kaiss, Marcelo Skaba, Ziyad Hanna, and Zurab Khasidashvili. Industrial strength sat-based alignability algorithm for hardware equivalence verification. In *FMCAD*, pages 20–26, 2007.

[KZH+11] Joost-Pieter Katoen, Ivan S. Zapreev, Ernst Moritz Hahn, Holger Hermanns, and David N. Jansen. The ins and outs of the probabilistic model checker MRMC. *Perform. Eval.*, 68(2):90–104, 2011.

[LAB11] M. Lahijanian, S.B. Andersson, and C. Belta. Control of markov decision processes from pctl specifications. In *American Control Conference (ACC), 2011*, pages 311–316, 2011.

[LDCd06] Julien Laumônier, Charles Desjardins, and Brahim Chaib-draa. Cooperative adaptive cruise control: a reinforcement learning ap-

proach. In *The Fourth Workshop on Agents in Traffic and Transportation, Hakodate, Hokkaido, Japan*. Citeseer, 2006.

[LP85] Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *POPL*, pages 97–107, 1985.

[LR81] Daniel J. Lehmann and Michael O. Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In John White, Richard J. Lipton, and Patricia C. Goldberg, editors, *POPL*, pages 133–138. ACM Press, 1981.

[McN93] Robert McNaughton. Infinite games played on finite graphs. *Ann. Pure Appl. Logic*, 65(2):149–184, 1993.

[MLOP07] Marta Cialdea Mayer, Carla Limongelli, Andrea Orlandini, and Valentina Poggioni. Linear temporal logic as an executable semantics for planning languages. *Journal of Logic, Language and Information*, 16(1):63–89, 2007.

[Nor03] J.R. Norris. *Markov Chains*. Cambridge University Press, 2003.

[NPK+05] Gethin Norman, David Parker, Marta Z. Kwiatkowska, Sandeep K. Shukla, and Rajesh Gupta. Using probabilistic model checking for dynamic power management. *Formal Asp. Comput.*, 17(2):160–176, 2005.

[NR11] Aditya V. Nori and Sriram K. Rajamani. Program analysis and machine learning: A win-win deal. In Hongseok Yang, editor, *APLAS*, volume 7078 of *Lecture Notes in Computer Science*, pages 1–2. Springer, 2011.

[Pit07] Nir Piterman. From nondeterministic büchi and streett automata to deterministic parity automata. *Logical Methods in Computer Science*, 3(3), 2007.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.

[PR89] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *POPL*, pages 179–190, 1989.

[PR97] Ronald Parr and Stuart J. Russell. Reinforcement learning with hierarchies of machines. In Michael I. Jordan, Michael J. Kearns, and Sara A. Solla, editors, *NIPS*. The MIT Press, 1997.

[Put94] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley-Interscience, April 1994.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, pages 337–351, 1982.

[Rab69] M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969.

[Rab82] Michael O. Rabin. N-process mutual exclusion with bounded waiting by 4 log_2 n-valued shared variable. *J. Comput. Syst. Sci.*, 25(1):66–75, 1982.

[RN10] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education, 2010.

[Ros97] Roni Rosner. *Modular Synthesis of Reactive Systems*. PhD thesis, Stanford University, 1997.

[RR03] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, pages 30–39. IEEE Computer Society, 2003.

[RS04] Kavita Ravi and Fabio Somenzi. Minimal assignments for bounded model checking. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2004.

[RvDdH11] Gijs Rennen, Edwin R. van Dam, and Dick den Hertog. Enhancement of sandwich algorithms for approximating higher-dimensional convex pareto sets. *INFORMS Journal on Computing*, 23(4):493–517, 2011.

[Sch09] Sven Schewe. Tighter bounds for the determinisation of büchi automata. In *FOSSACS*, pages 167–181, 2009.

[SDE08] Roopsha Samanta, Jyotirmoy V. Deshmukh, and E. Allen Emerson. Automatic generation of local repairs for boolean programs. In Alessandro Cimatti and Robert B. Jones, editors, *FMCAD*, pages 1–10, 2008.

[Sha53] L. S. Shapley. Stochastic Games. *Proceedings of the National Academy of Sciences of the United States of America*, 39(10):1095–1100, 1953.

[SLRBE05] Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In Vivek Sarkar and Mary W. Hall, editors, *PLDI*, pages 281–294. ACM, 2005.

[SNA12] Rahul Sharma, Aditya V. Nori, and Alex Aiken. Interpolants as classifiers. In P. Madhusudan and Sanjit A. Seshia, editors, *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 71–87. Springer, 2012.

[Sut95] Richard S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In David S. Touretzky, Michael Mozer, and Michael E. Hasselmo, editors, *NIPS*, pages 1038–1044. MIT Press, 1995.

[Tij03] H. C. Tijms. *A First Course in Stochastic Models*. Chichester: Wiley, 2003.

[Var85] Moshe Y. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *FOCS*, pages 327–338. IEEE Computer Society, 1985.

[VCD+12] Yaron Velner, Krishnendu Chatterjee, Laurent Doyen, Thomas A. Henzinger, Alexander Rabinovich, and Jean-François Raskin. The complexity of multi-mean-payoff and multi-energy games. *CoRR*, abs/1209.3234, 2012.

[VE03] Ardalan Vahidi and Azim Eskandarian. Research advances in intelligent collision avoidance and adaptive cruise control. *Intelligent Transportation Systems, IEEE Transactions on*, 4(3):143–153, 2003.

[vEG14] Christian von Essen and Dimitra Giannakopoulou. Analyzing the next generation airborne collision avoidance system. In Erika Ábrahám and Klaus Havelund, editors, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014), Held as Part of ETAPS 2014*, Lecture Notes in Computer Science, Grenoble, France, April 2014. Springer.

[vEJ11] Christian von Essen and Barbara Jobstmann. Synthesizing systems with optimal average-case behavior for ratio objectives. In Johannes Reich and Bernd Finkbeiner, editors, *iWIGP*, volume 50 of *EPTCS*, pages 17–32, 2011.

[vEJ12] Christian von Essen and Barbara Jobstmann. Synthesizing efficient controllers. In Viktor Kuncak and Andrey Rybalchenko, editors, *VMCAI*, volume 7148 of *Lecture Notes in Computer Science*, pages 428–444. Springer, 2012.

[vEJ13] Christian von Essen and Barbara Jobstmann. Program repair without regret. In Natasha Sharygina and Helmut Veith, editors, *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 896–911. Springer, 2013.

[VYY09] Martin Vechev, Eran Yahav, and Greta Yorsh. Inferring synchronization under limited observability. In *TACAS'09*, volume 5505 of *LNCS*, pages 139–154. Springer, 2009.

[VYY10] Martin T. Vechev, Eran Yahav, and Greta Yorsh. Abstraction-guided synthesis of synchronization. In Manuel V. Hermenegildo and Jens Palsberg, editors, *POPL*, pages 327–338. ACM, 2010.

[WBB+10] Ralf Wimmer, Bettina Braitling, Bernd Becker, Ernst Moritz Hahn, Pepijn Crouzen, Holger Hermanns, Abhishek Dhama, and Oliver E. Theel. Symblicit calculation of long-run averages for

concurrent probabilistic systems. In *QEST*, pages 27–36. IEEE Computer Society, 2010.

[WDH08] Ralf Wimmer, Salem Derisavi, and Holger Hermanns. Symbolic partition refinement with dynamic balancing of time and space. In *QEST* [DBL08], pages 65–74.

[WDHR06] Martin De Wulf, Laurent Doyen, Thomas A. Henzinger, and Jean-François Raskin. Antichains: A new algorithm for checking universality of finite automata. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 17–30. Springer, 2006.

[Whi93] D. J. White. A survey of applications of markov decision processes. *The Journal of the Operational Research Society*, 44(11):pp. 1073–1096, 1993.

[WVS83] Pierre Wolper, Moshe Y. Vardi, and A. Prasad Sistla. Reasoning about infinite computation paths (extended abstract). In *FOCS*, pages 185–194. IEEE, 1983.

[YBK10] Haidi Yue, Henrik C. Bohnenkamp, and Joost-Pieter Katoen. Analyzing energy consumption in a gossiping mac protocol. In Bruno Müller-Clostermann, Klaus Echtle, and Erwin P. Rathgeb, editors, *MMB/DFT*, volume 5987 of *Lecture Notes in Computer Science*, pages 107–119. Springer, 2010.

[ZH02] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.

[ZP96] Uri Zwick and Mike Paterson. The complexity of mean payoff games on graphs. *Theor. Comput. Sci.*, 158(1&2):343–359, 1996.

[ZPC13] Paolo Zuliani, André Platzer, and Edmund M. Clarke. Bayesian statistical model checking with application to stateflow/simulink verification. *Formal Methods in System Design*, 43(2):338–367, 2013.