Algorithmic Improvisation

Daniel J. Fremont

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2019-133
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-133.html

August 27, 2019

Acknowledgement

This is the Ph.D. dissertation of Daniel J. Fremont in the Group in Logic and the Methodology of Science at the University of California, Berkeley, written under the supervision of Professor Sanjit A. Seshia. The abstract was slightly shortened to fit on this web page.

Algorithmic Improvisation

by

Daniel Juon Fremont

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Logic and the Methodology of Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Sanjit A. Seshia, Chair
Professor Antonio Montalbán
Professor Stuart J. Russell
Professor Alistair Sinclair
Professor David Wagner

Summer 2019

Algorithmic Improvisation

Abstract

Algorithmic Improvisation

by

Daniel Juon Fremont

Doctor of Philosophy in Logic and the Methodology of Science

University of California, Berkeley

Professor Sanjit A. Seshia, Chair

The increasing use of autonomy for safety-critical tasks, from operating power grids to driving cars, has led to an acute need for reliable and secure systems. The ideal approach to obtaining rigorous reliability guarantees is to automatically construct systems from formal specifications using *correct-by-construction synthesis*. A new dimension in this area is the synthesis of *randomized* systems, which, as we show in this thesis, enables a broad range of new applications in safe autonomy and other fields. This is because randomness can provide several crucial benefits to a system, including *robustness*, *variety*, and *unpredictability*. For example, a robot following a random route can be harder for an adversary to intercept, making the system more secure; a synthetic data generator for a machine learning algorithm can use randomness to produce diverse training data, making the ML model more robust. When building these types of systems, we cannot simply add randomness in an *ad hoc* way: we need provable guarantees that the system is safe and satisfies our desired specifications. The key question, then, is *how can we automatically synthesize a system with random behavior but formal guarantees?* Our answer to this question is *algorithmic improvisation*. This thesis proposes a theory of algorithmic improvisation enabling the correct-by-construction synthesis of randomized systems, and explores its applications to safe autonomy.

The first part of the thesis studies the theory of algorithmic improvisation in depth. We begin by introducing *control improvisation (CI)*, the core computational problem of algorithmic improvisation, which requires constructing an *improviser*, a randomized algorithm generating finite sequences of symbols subject to hard, soft, and randomness constraints. We develop a general approach to building improvisers, instantiate it to obtain efficient synthesis algorithms for several practical classes of CI problems, and prove hardness results for more difficult classes. Next, we generalize CI to the *reactive control improvisation (RCI)* problem, which allows us to synthesize *open* systems that interact with an uncontrolled and potentially adversarial environment: our goal is an *improvising strategy* that ensures the hard, soft, and randomness constraints hold no matter what actions are taken by the environment. We

again give efficient algorithms for constructing improvising strategies in some useful cases, and hardness results in others. Finally, we investigate *language-based improvisation*, a variant of algorithmic improvisation which uses a probabilistic programming language to provide greater control over the distribution of the improviser. We design a *domain-specific probabilistic programming language*, Scenic, for defining distributions over *scenes*, configurations of physical objects and agents. Scenic significantly decreases the effort required to specify the highly complex environments of systems like self-driving cars.

In the second part of the thesis, we demonstrate how algorithmic improvisation can help with the design, analysis, and testing of autonomous systems. First, we show how to synthesize *randomized planners for mobile robots*, for example a patrolling security robot which uses randomness to make its route less predictable while still guaranteeing safety and efficiency requirements. Next, we study using algorithmic improvisation to create *human models* with realistic stochasticity and tunable behavior, a vital prerequisite for the design of a system which interacts with people. Finally, we propose a methodology for using language-based improvisation to train, test, and debug cyber-physical systems like autonomous cars by *generating synthetic data* from customizable distributions. We apply our methodology to an industrial convolutional neural network for object detection, finding bugs in the system, eliminating them through retraining, and boosting the performance of the network beyond what could be achieved with prior techniques by using Scenic to design training sets in a more intelligent way.

In summary, algorithmic improvisation is a mathematical framework for synthesizing randomized systems satisfying formal specifications. It has already proved useful in a wide range of fields, including robotics, cyber-physical systems, computer music, and machine learning, and shows promise in a variety of further applications to the design of secure and dependable systems.

To my family

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgments

This thesis would not exist without the efforts of many, many, people, for whose help I am extremely grateful.

First and foremost I thank my family for their unwavering love and support, which made everything else possible. What I owe you is truly beyond evaluation.

I also thank my many wonderful teachers over the years, in particular Art Squillante and Mike Thibodeaux, who nurtured my enthusiasm for math from a casual interest to an enduring love. Significant credit for this also goes to Douglas Hofstadter, whose *Gödel, Escher, Bach: an Eternal Golden Braid* introduced me to mathematical logic, setting me unknowingly on a path that determined my career — *GEB* remains my favorite book.

I am also very grateful to Abhinav Kumar and Shankar Raman, who mentored me in research projects at MIT, as did Henry Cohn. I am particularly grateful to Henry, since his class on logic and set theory was the greatest class I have ever attended, and my research project with him bounced from computability of Newton fractals to reachability problems, ultimately sending me to my first academic conference, RP'12 in Bordeaux, where I discovered the field of formal methods. Thus, Douglas Hofstadter and Henry Cohn are to blame for my chosen career.

Of course, not guiltless is my advisor Sanjit Seshia, who took me into his group knowing not much beyond that I had a vague interest in formal methods, despite the fact that I was not even a member of his department (by some oversight, Sanjit was not in the Logic Group at the time). I am very grateful to Sanjit for allowing me free choice of what to work on, suggesting multiple possible topics which always turned out to be fascinating later on. Sanjit also provided excellent advice on all aspects of graduate school and research in general, and made a point to help connect me to other people in formal methods, which was not always an easy task, given my introversion. For pointing me in the right direction, and making it possible for me to become a professor, Sanjit has my eternal gratitude.

Last but not least, I thank my many collaborators, without whom I would know and have done much less — Ilge Akkaya, Robert Bocchino, Supratik Chakraborty, Johnathan Chiu, Ankush Desai, Alexandre Donzé, Tommaso Dreossi, Adrian Freed, Shromona Ghosh, Rajeev Joshi, Edward Kim, Orna Kupferman, Edward Lee, Kuldeep Meel, Nathan Mull, Markus Rabe, Hadi Ravanbakhsh, Rafael Valle, Moshe Vardi, Marcell Vazquez-Chanlatte, Xiangyu Yue, and David Wessel — as well as many colleagues who I did not have the pleasure of working closely with but who helped me with their advice and their friendship: Daniel Bundala, Ben Caulfield, Kevin Cheang, Rafael Dutra, Mark Ho, Susmit Jha, Garvit Juniwal, Eric Kim, Jonathan Kotker, Forrest Laine, Caroline Lemieux, Wenchao Li, Rohan Padhye, Cameron Rasmussen, Dorsa Sadigh, Indranil Saha, Yasser Shoukry, Rohit Sinha, Pramod Subramanyan, Wei Yang Tan, Nishant Totla, Vasu Raman, Zach Wasson, and Yi-Chin Wu.

To many other people who have helped me in one way or another but were omitted here: my thanks.

Chapter 1

Introduction

As computers are integrated into every aspect of our lives, and are given control of ever more complex tasks, the need for dependable, secure systems has become acute. With little or no human supervision, computers routinely operate *safety-critical* systems such as power grids, airplanes, and cars, whose failures can have catastrophic consequences, Ensuring the safety and reliability of such systems is increasingly important, yet providing such guarantees is increasingly difficult due to massive system complexity[1], massive diversity of environments (compare the environment of an autonomous car to that of a traditional computer program), and the growing use of black-box machine learning (ML) algorithms to handle such environments. As a result, we have seen software bugs with increasingly dire consequences, ranging from the destruction of expensive systems in the failure of the Ariane 5 rocket [72, 110] and the shutdown of critical utilities in the 2015 hacking of the Ukrainian power grid [194] to loss of life in accidents involving semi- or fully-autonomous vehicles [181, 183].

One promising approach to avoid such disasters is to use *formal methods*, which provides rigorous techniques to design, model, and analyze systems based on formal, mathematical reasoning [185, 36]. The starting point of formal methods is *specification*, the process of encoding the properties a system is intended to satisfy into a precise, machine-readable formalism: a *formal specification*. Given a system and a specification for it, *verification* algorithms generate either a *proof* that the system satisfies the specification or a *counterexample* showing how it can violate the specification. More powerfully, one can start with only a specification and use an algorithm to automatically generate an implementation for the system which is guaranteed to satisfy the specification: this is *correct-by-construction synthesis*. Synthesis has been widely studied and successfully used in practice to design both hardware and software [83, 56]. However, most synthesis techniques target the design of *deterministic* systems. In this thesis, we argue that there are many situations where we want a system to exhibit behavior which is to a certain extent *random*.

[1] For example, as of 2016 some cars ran software containing 150 million lines of code — this is up from only 10 million lines in 2010, and with the imminent deployment of self-driving cars, we can expect further explosive growth [22].

1.1 Synthesis of Randomized Systems

There are a number of different reasons why randomness can be desirable in a system, corresponding to several crucial benefits that randomness can provide:

Variety. A black-box fuzz tester for a network service should generate packet sequences which conform to the protocol (perhaps only most of the time), but the space of such sequences may be too large to search exhaustively — randomness lets us cover the space evenly. Likewise, in computer music improvisation we can use randomness to generate diverse variations on a given melody, subject to the conventions of the genre [44].

Robustness. A controller for a robot exploring an unknown environment can use randomness to reduce systematic bias and increase coverage without needing to construct a detailed map or assume the environment is static [182]. Similarly, a scheduler for a power microgrid can use randomness to gain robustness to correlated failures [2].

Unpredictability. A controller for a surveillance robot can use randomness to make its route less predictable and thereby more difficult to plan against in advance [81]. Similarly, a compiler can introduce randomness into a program to make it more difficult for an attacker to develop exploits based on low-level details of the implementation [105].

Realism. A lighting controller for a house that mimics typical human behavior when its occupant is on vacation can use randomness to realistically model the human's stochastic behavior [2]. Similarly, a synthetic data generator for an ML algorithm needs randomness to model the distribution of the real-world data the algorithm will be used on [64].

Note how the fuzz testing and synthetic data generation applications show that building randomized systems can be helpful even when designing deterministic systems: we can build randomized generators producing specialized tests, environment behaviors, or training data to help us design the actual system of interest. Furthermore, since testing only requires that the system can be simulated, it applies to systems which are currently well out of reach of formal verification, including cyber-physical systems like autonomous cars. Such systems, particularly those using ML, pose serious challenges for formal methods [155, 149], with most techniques not scaling to realistic systems and specifications [111, 191], but test generation is eminently scalable and widely-used in practice [114]. So a general way to construct randomized systems with desired properties would allow us to do intelligent testing and training of a wide variety of safety-critical systems.

In fact, in all of these applications — software fuzz testing, music improvisation, robotic planning, microgrid scheduling, human behavior modeling, and synthetic data generation — there are constraints on the behavior of the system that we construct. For example, the patrolling robot should not run into obstacles. To automatically build systems which are guaranteed to satisfy such constraints, we can use correct-by-construction synthesis. However, as we mentioned above, no existing synthesis algorithms allow a requirement for

randomness as part of the system's specification, and we cannot just add randomness to the system in an *ad hoc* way, since this could easily break correctness. Furthermore, even if adding randomness does not break functional correctness, there are likely *trade-offs* between the amount of randomness added and efficiency or some other quantitative objective. To explore such trade-offs, we need to integrate the requirement for randomness into the synthesis process.

The question, then, is *how can we automatically synthesize a system with random but controlled behavior?* Our answer to this question is *algorithmic improvisation.*

1.2 Algorithmic Improvisation

Algorithmic improvisation describes a class of problems requiring the synthesis of a randomized algorithm, called an *improviser*, which must satisfy three different types of constraints:

Hard Constraint. *The behavior of the improviser must always satisfy some desired property, given by a formal specification.* This hard constraint is just like the specification in traditional synthesis, which defines functional correctness properties the synthesized implementation must respect.

Soft Constraint. *The behavior of the improviser must satisfy some other given property to a certain extent, which can be tuned according to a parameter.* In this thesis, we will consider soft constraints which require a property to hold *with at least a given probability*, although other types of quantitative requirements are useful as well.

Randomness Constraint. *The behavior of the improviser must exhibit a specified type of randomness.* There are again several ways this can be formalized; here, we will mainly study constraints which require the distribution of the improviser's behavior to be sufficiently close to uniform.

The intuition for why algorithmic improvisation combines hard, soft, and randomness constraints is straightforward: the hard constraint is inherited directly from traditional synthesis problems, allowing us to give formal guarantees about the behavior of the system we construct. The randomness constraint ensures we obtain a diversity of behaviors, rather than simply finding one way to satisfy the hard constraint. Finally, the soft constraint allows us to control more precisely how randomness is added to the system, and to trade off randomness for correctness.

We can see how algorithmic improvisation captures the applications of randomized synthesis mentioned above through a couple of examples. In the case of robotic planning, consider generating routes for a surveillance robot. The hard constraint allows us to enforce safety properties like avoiding collisions, as well as mission goals like visiting each of a designated set of locations. Through the randomness constraint, we can achieve a randomized policy for the robot that accomplishes the mission while making it difficult for a third party to predict the robot's trajectory in advance. Finally, we can use the soft constraint to

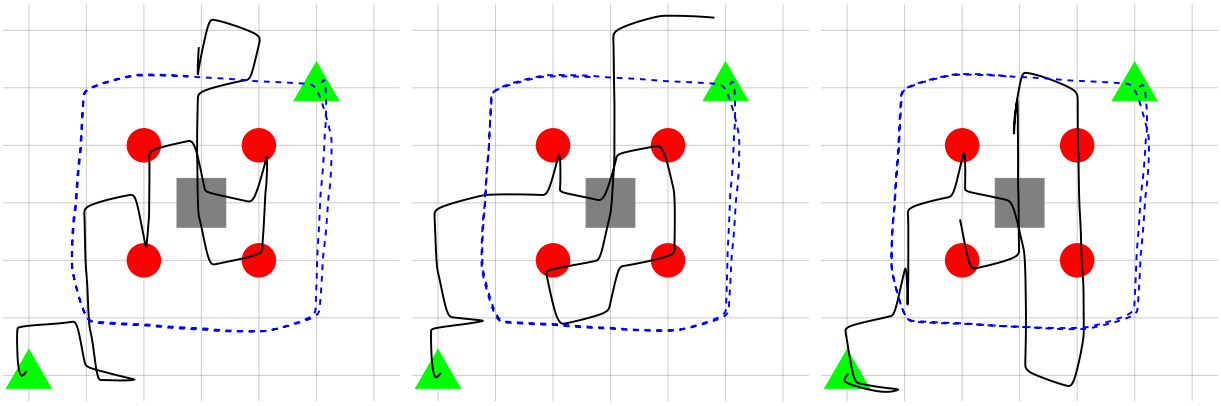


Figure 1.1: Three improvised trajectories for a patrolling drone (solid) which must visit the circled locations while avoiding collisions with another drone (dashed).

ensure that the generated routes are usually close in length to the shortest possible route, rather than achieving randomness by being arbitrarily circuitous. Figure 1.1 shows several trajectories generated using algorithmic improvisation in this way (see Chapter 6 for details).

These types of constraints again prove useful in a completely different domain, namely software fuzz testing. Suppose we want to generate audio files to test a media player application. Two popular heuristics in fuzz testing are the *generative* approach, using a grammar to generate random files in a given file format, and the *mutational* approach, generating random variations on real-world seed files [165]. Using algorithmic improvisation, we can synthesize a test generator which combines both approaches: we enforce the file format using the hard constraint, ensure similarity (most of the time) to a seed input using the soft constraint, and ensure a diversity of files using the randomness constraint.

A New Class of Computational Problems

In both examples above, all three fundamental aspects of algorithmic improvisation — hard, soft, and randomness constraints — are essential, and this is equally true for the other applications we consider in this thesis. However, existing techniques in the literature only capture some of these aspects., as summarized in Table 1.1.

In one direction, simple probabilistic models like the factor oracles used in music improvisation [9] and the heuristics in mutational fuzz testing [165] allow creating random variations with tunable similarity to a seed input, a kind of soft constraint. However, enforcing probability bounds like our randomness constraint is not directly supported by such techniques, and at any rate they do not allow imposing hard constraints. Conversely, algorithms for uniform random sampling from the languages of automata and grammars [86, 93] and from the satisfying assignments of a Boolean formula [89, 15, 29] allow hard but not soft constraints. As we will see in Chapter 3, however, our work is closely connected to such algorithms and builds on top of them.

Table 1.1: Related work allowing synthesis under hard, soft, or randomness constraints. The symbol $\sim$ indicates that the method allows defining randomized algorithms but not *ensuring* that a randomness constraint is satisfied. The **Adversarial** column indicates whether the synthesized algorithm must be correct in the presence of a potentially-adversarial environment (providing input to the algorithm). Citations are to representative or survey papers.

	Hard	**Soft**	**Random**	**Adversarial**
Factor Oracles [9]	⊘	✓	$\sim$	⊘
Sampling from Automata/Grammars [86, 93]	✓	⊘	✓	⊘
Sampling from Boolean Formulas [89, 15, 29]	✓	⊘	✓	⊘
Probabilistic Programming [79]	✓	✓	$\sim$	⊘
Reactive Synthesis [56]	✓	⊘	⊘	✓
Program Synthesis [83]	✓	⊘	⊘	✓
Control Improvisation [63, 62]	✓	✓	✓	⊘
Reactive Control Improvisation [67]	✓	✓	✓	✓

Possibly the closest prior work to ours is the field of *probabilistic programming languages (PPLs)* [79]. Such languages allow the construction of generative processes subject to hard and soft constraints, although they do not provide a way to check that the resulting distribution will satisfy our randomness constraint. However, we will see in Chapter 5 that PPLs *are* ideally suited to a variant of algorithmic improvisation where we need more complex and interdependent hard, soft, and randomness constraints, and detailed control over the generative process underlying the improviser.

Finally, in an orthogonal direction, there has also been much work on the synthesis of *reactive* systems [56] which interact with a potentially-adversarial environment. As we mentioned above, reactive synthesis algorithms have focused on deterministic systems[2], and do not provide a way to *ensure* randomness. The same is true of work on program synthesis [83], which generates a program which is correct even when its input is chosen adversarially, but does not support randomness constraints. Although there have been proposals to synthesize multiple variants of programs to make the task of developing exploits harder [113, 30], these approaches are heuristic and provide no diversity guarantees.

As a result, algorithmic improvisation is *a fundamentally new type of problem in computer science*, and requires the development of its own theory. A primary goal of this thesis is to develop such a theory, as we will detail in the next section. First, however, we close our general discussion of algorithmic improvisation with a history of the concept.

[2]Although potentially with stochastic *environments*; see Chapter 4 for a discussion.

The History of Algorithmic Improvisation

Algorithmic improvisation was originally conceived by Sanjit A. Seshia and David Wessel in 2013. The core computational problem of algorithmic improvisation, *control improvisation (CI)*, was introduced by Donzé et al. [43, 44]. Their main motivating application was computer music improvisation (a topic with a long history; see e.g. Rowe [147]), specifically generating random variations of a given melody subject to the conventions of the music genre. Thus, although their problem definition was somewhat different than the one we use here, it contained the three essential elements of algorithmic improvisation: a hard constraint, a tunable soft constraint, and a randomness constraint. The name "control improvisation" (alternatively, "controlled improvisation"), coined by Seshia, reflects the problem's original application to music improvisation, as well as its applicability to various control tasks, such as robotic planning; in fact, the initial formulation of CI was a randomized variant of supervisory control [25].

Our work on algorithmic improvisation began in Fremont et al. [63], which redefined control improvisation, giving a definition suited to a broader range of applications. We also studied the theory of the problem in depth for the first time, which allowed us to give the first algorithms for CI with formal guarantees: by contrast, the improvisation algorithm proposed by Donzé et al. [43, 44] was heuristic, not providing a guarantee of satisfying the soft constraint (or a way of checking whether the soft and randomness constraints were in fact compatible). We also further studied the application of CI to music improvisation [178, 177], as well as a new application to human behavior modeling [2]. Next, we refined the definition of CI in Fremont et al. [62], obtaining the formulation used in this thesis and further extending its theory in several directions. In Fremont and Seshia [67] we proposed a more far-reaching generalization, *reactive control improvisation*, which allows the synthesis of randomized reactive systems. Finally, inspired by control improvisation, in Fremont et al. [64] we explored a different variant of algorithmic improvisation based on probabilistic programming languages. All of this work (excluding that on music improvisation, for which see Valle [176]) comprises the content of this thesis, which we now describe in detail.

1.3 Thesis Contributions

This thesis proposes a theory of algorithmic improvisation enabling the correct-by-construction synthesis of randomized systems, and explores its applications to safe autonomy.

The thesis is divided into two parts: in Part I, we lay the theoretical foundations of algorithmic improvisation, defining and studying the problems of *control improvisation* (Chapter 3), *reactive control improvisation* (Chapter 4), and *language-based improvisation* (Chapter 5). As we have suggested above, these problems have many applications, including software and protocol fuzz testing, microgrid scheduling, and computer music improvisation. Part II develops in detail three of these applications that are particularly relevant to the challenge of building safe autonomous systems: robotic planning (Chapter 6), human

behavior modeling (Chapter 7), and synthetic data generation for cyber-physical systems (Chapter 8). We now give an overview of each of these chapters.

1.3.1 Theory

In Part I, we study the theory of algorithmic improvisation in three different forms:

Control Improvisation

We begin in Chapter 3 by introducing *control improvisation (CI)*, the computational problem at the core of algorithmic improvisation. Control improvisation is the problem of constructing an *improviser*, a randomized algorithm generating finite sequences of symbols satisfying three requirements: a *hard constraint* which must always be satisfied (like the specification in traditional synthesis), a *soft constraint* which need only be satisfied with some probability, and a *randomness constraint* requiring that the improviser's output distribution be sufficiently uniform.

We prove a precise characterization of when CI problems are solvable, and give a general procedure for constructing improvisers by reducing the problem to model counting and uniform sampling. We use this procedure to develop polynomial-time synthesis algorithms for several practical classes of CI problems where the hard and soft constraints are defined by deterministic finite automata or unambiguous context-free grammars. We also analyze various more general classes of CI problems, showing that their complexity is equivalent to the counting class #P (our complexity results are summarized on the left of Table 1.2). This is true in particular for CI problems with specifications given by Boolean formulas, but we show how to *approximately* solve such CI problems using only an NP oracle, or in practice, a SAT solver. Finally, we discuss a generalization of CI allowing multiple hard and soft constraints, giving an EXP synthesis algorithm as well as evidence that multiple constraints do in fact increase the difficulty of the problem.

Chapter 3 is based on Fremont et al. [63] and its extended version [62], joint work with Alexandre Donzé, Sanjit A. Seshia, and David Wessel. These papers generalize the initial CI proposal by Donzé et al. [43, 44].

Reactive Control Improvisation

Next, in Chapter 4 we generalize control improvisation by adding *reactivity*, allowing the synthesis of *open* systems that interact over time with an uncontrolled and potentially adversarial environment. Like CI, *reactive control improvisation (RCI)* has hard, soft, and randomness constraints, but instead of an improviser we now seek an *improvising strategy* which ensures these constraints regardless of the actions taken by the environment.

We again begin our study of RCI by analyzing when the problem is solvable. To do this, we introduce the notion of the *width* of a 2-player game, which counts the number of ways a player can win from a given position (generalizing the usual concept of a "winning" position).

Table 1.2: Complexity of the control improvisation (CI) problem (left) and *reactive* CI problem (right) for various types of hard and soft specifications $\mathcal{H}$, $\mathcal{S}$. **DFA**/**NFA**: deterministic/nondeterministic finite automaton. (**U**)**CFG**: (unambiguous) context-free grammar. **RSG**: reachability or safety game. **LTL**: formula of linear temporal logic.

$\mathcal{H}\backslash\mathcal{S}$	**DFA**	**UCFG**	**CFG**	**NFA**
DFA	poly-time			
UCFG				
CFG			#P-equivalent	
NFA				

$\mathcal{H}\backslash\mathcal{S}$	**RSG**	**DFA**	**NFA**	**LTL**
RSG	poly-time			
DFA				
NFA			poly-space	
CFG				
LTL				

Using width, we prove a precise characterization of when RCI problems can be solved; interestingly, we show that some simple RCI problems can only be solved by strategies with memory, even when the corresponding non-randomized 2-player games admit memoryless strategies. We also give a general construction of an improvising strategy based on computing widths, and instantiate it to provide a polynomial-time synthesis algorithm for RCI problems based on safety/reachability games and more generally, DFAs. We prove that for more general specifications including temporal logic formulas (popular in reactive synthesis), the complexity of RCI increases to PSPACE (our complexity results are summarized on the right of Table 1.2). However, in all cases we show that finding a randomized strategy using RCI is no harder, in a coarse-grained complexity sense, than finding a single winning strategy.

Chapter 4 is based on joint work with Sanjit A. Seshia [67].

Language-Based Improvisation

We conclude our theoretical study of algorithmic improvisation in Chapter 5 with an investigation of *language-based improvisation (LBI)*. In LBI, specifications are represented using a *probabilistic programming language*, which in addition to supporting declarative hard and soft constraints also allows detailed control of probability distributions. This enables applications of algorithmic improvisation where we want non-uniform distributions, as when generating synthetic training data for a machine learning algorithm which must match the distribution of real-world data.

We design a domain-specific probabilistic programming language, Scenic, for defining distributions over *scenes*, configurations of physical objects and agents. Scenes, which form the environments of cyber-physical systems like autonomous cars and robots, are a highly complex domain, making it possible for a DSL to significantly decrease the effort required to specify reasonable distributions for training and testing data. Scenic is designed with this in mind: we illustrate with examples many features of the language which simplify the task of environment modeling, as well as giving a detailed description of its syntax and a

```
from gta import Car, curb, roadDirection

ego = Car

spot = OrientedPoint on visible curb
badAngle = Uniform(1.0, -1.0) * (10, 20) deg
Car left of (spot offset by -0.5 @ 0),
    facing badAngle relative to roadDirection
```

Figure 1.2: A SCENIC program describing a badly-parked car, and two sampled scenes.

formal operational semantics. Finally, we show how the domain-specific design of SCENIC enables specialized algorithms for *scene improvisation*, the problem of sampling scenes from a SCENIC program, which would not be possible with general-purpose probabilistic programming languages. Figure 1.2 shows an example SCENIC program, as well as two scenes sampled from it.

Chapter 5 is based on joint work with Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia [64, 65].

1.3.2 Applications

In Part II, we explore several applications of algorithmic improvisation to the construction of safe autonomous systems:

Robotic Planning

Chapter 6 investigates the use of algorithmic improvisation to synthesize *randomized planners* for mobile robots. In particular, we study surveillance problems where a robot must visit a set of locations (while ensuring safety or other constraints), using randomness to make its route less predictable, as in Figure 1.1. We show how to formulate such problems as instances of control improvisation if the environment is known ahead of time, or reactive control improvisation if not. We present experiments demonstrating both formulations,

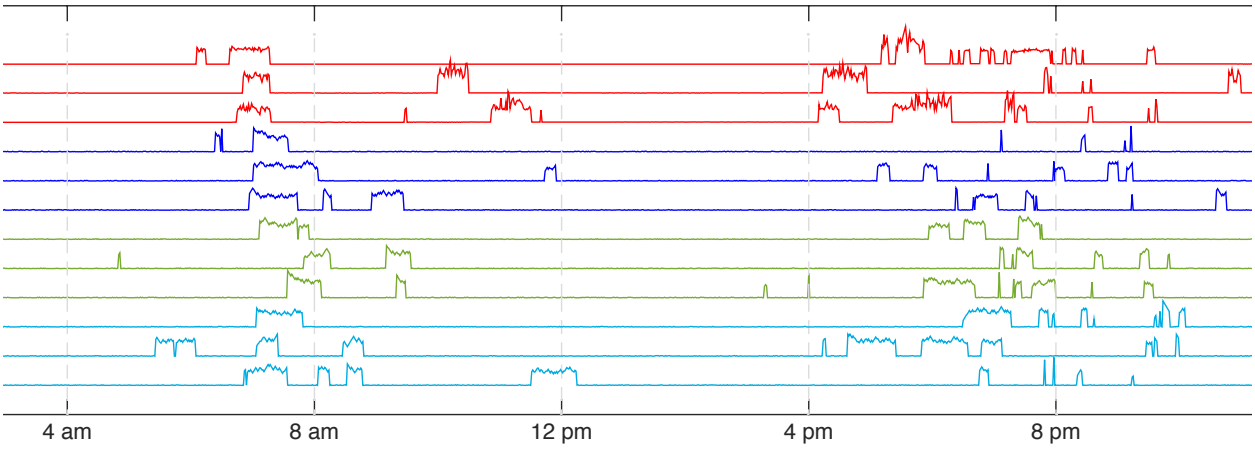


Figure 1.3: Human (red, top) and improvised power consumption traces.

constructing planners using our improvisation schemes for finite automata specifications developed in Chapters 3 and 4. We test these planners on an actual drone as well as in simulation.

Chapter 6 is based on Fremont et al. [63] and Fremont and Seshia [67], as cited above for Chapters 3 and 4. It also references experiments conducted jointly with Ankush Desai, Brent Schlotfeldt, Yasser Shoukry, and Dinesh Thakur.

Human Modeling

Next, Chapter 7 studies using algorithmic improvisation to synthesize *models of human behavior* subject to constraints. Such models are critical when designing systems like autonomous cars which must interact with humans: without a model of how humans behave, we cannot even do meaningful testing, let alone formal verification.

We focus on a case study in home automation, where the task is to design a lighting controller which mimics typical human behavior to obscure the fact that the occupant is away, while also respecting soft constraints on power consumption. We show how to partially formulate this task as a multi-constraint control improvisation problem as studied in Chapter 3. However, while this formulation allows us to impose constraints on the controller, it does not allow us to *learn* the concept of what behaviors are "human-like" directly from historical human data. We therefore propose a heuristic procedure which first learns a probabilistic model from data, then iteratively adjusts the model until the soft constraints are satisfied. Experiments demonstrate that this approach yields synthetic lighting behaviors that satisfy our desired constraints, while still being qualitatively and quantitatively similar to human behaviors (see Figure 1.3 for examples).

Chapter 7 is based on joint work with Ilge Akkaya, Rafael Valle, Alexandre Donzé, Edward A. Lee, and Sanjit A. Seshia [2].

Synthetic Data Generation

Finally, Chapter 8 proposes a methodology for using language-based improvisation, introduced in Chapter 5, to train, test, and debug cyber-physical systems by *generating synthetic data* from specialized distributions. In particular, such data can be used to design more effective training sets by emphasizing rare events, assess system performance under different conditions, find the root cause of bugs by constructing orthogonal variations on a single failure case, and eliminate bugs by generalizing failure cases into broader scenarios suitable for retraining.

We demonstrate all of these applications using Scenic, the language developed in Chapter 5, to specify environment distributions for a number of different systems. Our main case study examines a convolutional neural network used for object detection in autonomous cars, using Scenic to generate synthetic traffic images in a video game with high-fidelity rendering. Using our methodology, we are able to find bugs in the system and eliminate them through retraining. We also boost the performance of the network significantly beyond what could be achieved by prior synthetic data generation techniques, using Scenic to design training sets in a more intelligent way. In addition to this case study, we mention several other experiments where we have successfully applied our methodology to *controllers* as well as perception systems, using several other simulators.

Chapter 8 is primarily based on Fremont et al. [64], as cited above for Chapter 5. It also references some results from joint work with Tommaso Dreossi, Shromona Ghosh, Edward Kim, Hadi Ravanbakhsh, Marcell Vazquez-Chanlatte, and Sanjit A. Seshia [47], as well as unpublished work with Johnathan Chiu.

Chapter 2

Background

In this chapter we define various fundamental concepts which will be used throughout the thesis. We start with basic mathematical concepts in Section 2.1. The main notions we need from the theory of computation, in particular complexity theory, are described in Section 2.2. Next, Section 2.3 defines several formalisms we will use to represent specifications, namely finite automata, context-free grammars, and Boolean formulas. Finally, Section 2.4 outlines prior work on counting and uniform sampling, both core problems underlying algorithmic improvisation.

2.1 Basic Concepts and Notation

For basic mathematical concepts, we generally use standard notation; here we mention several symbols that can be ambiguous or are not fully standardized. We denote by $\mathbb{N}$ the *natural numbers*, i.e. the nonnegative integers (note we include zero). For any positive $k \in \mathbb{N}$, we use $[k]$ to denote the set $\{1, \ldots, k\}$. For any $x \in \mathbb{R}$ we write $\lfloor x \rfloor$ and $\lceil x \rceil$ for the floor and ceiling of x, respectively the greatest integer which is at most x and the least integer which is at least x.

Given a set Σ, the *Kleene star* Σ^* denotes the set of all finite sequences of elements of Σ. Following the convention in formal language theory, we sometimes call Σ an *alphabet*, in which case its elements are *symbols* and a sequence $w \in \Sigma^*$ is a *word* (or *string*) over Σ. Any set of words $S \subseteq \Sigma^*$ is called a *language*, and we write $\overline{S}$ for its *complement* $\Sigma^* \setminus S$. Given a word $w \in \Sigma^*$, we denote its length by $|w|$. We denote the *empty word* of length zero by λ. We write Σ^n for the set of words of length n, and $\Sigma^{\leq n}$ for the words of length at most n. Finally, we define $\Sigma^{m:n} = \{w \in \Sigma^* \mid m \leq |w| \leq n\}$, the words whose length is between m and n.

2.2 Computational Models and Complexity

Next we briefly describe the mathematical formalism from the theory of computation which underlies our work. For formal definitions of all of the concepts in this section, see Arora and Barak [8].

Algorithms and Computational Problems

When referring to computational procedures we will use the generic term *algorithm*. Since we will usually only be concerned with questions of coarse-grained complexity, e.g. can a problem be solved in polynomial time or not, the exact details of the model are not important. For concreteness, the reader can view "algorithm" as referring to a Turing or RAM machine (the latter being what is meant in the few cases where we discuss fine-grained complexity, e.g. linear vs. quadratic time). We will often discuss *randomized algorithms*, which can be formalized in the usual way by adding to the machine the ability to flip an unbiased coin (with the result being written to memory or determining which of two states the machine enters).

The input and output of an algorithm are binary strings, i.e. elements of $\{0,1\}^*$, which we can think of as representing numbers or other objects by an appropriate encoding. Thus we can think of a computational *problem* as a function $f : \{0,1\}^* \to \{0,1\}^*$ mapping problem *instances* to the corresponding *answers*. An algorithm *solves* the problem if given any instance $x \in \{0,1\}^*$ as input the algorithm eventually terminates and outputs $f(x)$. For *decision problems* whose answer is yes or no, we can alternatively represent the problem as a language $P \subseteq \{0,1\}^*$ whose elements are the problem instances for which the answer is yes.

It will sometimes be useful for us to discuss algorithms which use another algorithm as a subroutine. If A is an algorithm which internally uses a black-box algorithm for solving the problem P, we formalize A in the usual way as an *oracle machine* A^P, which has a special primitive operation for querying its *oracle* for P on an instance which has already been written to memory. This operation takes a single step, so that the runtime of A does *not* include any time needed to actually solve the queried instances of P (which would be nonzero in practice, with the oracle being implemented by another algorithm).

Computational Complexity

To measure the difficulty of solving various computational problems, we use the standard notions of *complexity classes* and *reductions*. We make use of the following standard complexity classes of decision problems:

P: Decision problems solvable in polynomial time.

NP: Problems solvable in polynomial time by a nondeterministic machine, i.e. a machine which may fork into multiple independent runs, the entire machine accepting if and

only if any of the individual runs do. Informally, this class consists of problems whose solutions can be *verified* in polynomial time.

PH: The *polynomial-time hierarchy*, whose first level is NP. Informally, thinking of the acceptance condition for an NP-machine as *existentially quantifying* over the nondeterminism of the machine, each successive level of the hierarchy adds one additional quantifier (e.g. at level 2 we have problems of the form $P(x) = \forall y.\exists z.M(x, y, z)$ with M a polynomial-time algorithm).

PP: Problems solvable in polynomial time by a nondeterministic machine whose acceptance condition is that a *majority* of its runs accept. Equivalently, problems solvable by a polynomial-time *randomized algorithm* which returns the correct answer strictly more than half the time.

PSPACE: Problems solvable using polynomial space (i.e. memory).

EXP: Problems solvable in exponential time.

We also use two standard classes of *function problems*, whose answer is a number (encoded in binary as above) rather than simply yes or no:

FP: Function problems solvable in polynomial time.

#P: *Counting problems*, i.e. function problems defined by a polynomial-time nondeterministic machine (as for NP), where the answer is *how many* runs of the machine are accepting.

We use these classes to measure the difficulty of problems in the ordinary way. A problem P is *hard* for a complexity class $\mathcal{C}$ if every problem in the class can be *reduced* to P: informally, an algorithm for P can be used to solve every problem in $\mathcal{C}$. The type of reduction used depends on the class: for almost all of the classes above, the usual choice is a polynomial-time many-one reduction, also called a *Karp reduction*. A Karp reduction from P to Q is a function $f : \{0,1\}^* \to \{0,1\}^*$ which is computable in polynomial time and has the property that for all instances $x \in \{0,1\}^*$, we have $x \in P$ if and only if $f(x) \in Q$. The one exceptional class is #P, for which completeness is defined (following Valiant [174]) using a polynomial-time Turing reduction, also called a *Cook reduction*. A Cook reduction from P to Q is a polynomial-time oracle Turing machine R^Q which computes P given an oracle for Q. This is a more powerful type of reduction, since the machine is allowed to query any number of instances of Q, whereas in a Karp reduction we can only apply the mapping f once and not do any additional computation.

A problem P is *complete* for a complexity class $\mathcal{C}$ if P is $\mathcal{C}$-hard and $P \in \mathcal{C}$: intuitively, P is one of the hardest problems in $\mathcal{C}$. Thus, when P is $\mathcal{C}$-complete we will say its complexity *is* $\mathcal{C}$. For example, the SAT problem (defined below) is NP-complete, and therefore not in P unless P = NP, so we say that SAT has complexity NP.

Sometimes it will be convenient to compare the complexity of a function problem P to a class of decision problems $\mathcal{C}$: we say a problem P is $\mathcal{C}$-*equivalent* if there are Cook reductions from P to some $\mathcal{C}$-complete problem and vice versa. So for example a $\#\mathsf{P}$-equivalent problem P is not necessarily itself a counting problem, but has the same difficulty: an algorithm for P can be used to solve all counting problems, and there is some counting problem that can be used to solve P.

2.3 Specification Formalisms

Informally, a specification for a system is a *property* which the system should have. The simplest and most common type of property are *trace properties*: these are properties of individual behaviors of the system, which the overall system satisfies if all of its possible behaviors do. For example, given a mobile robot, the property of always reaching a particular location is a trace property (for any given behavior, we either reach the location or we don't). Viewing the behavior of the system as a word over some alphabet (a *trace*), a trace property is simply a language, consisting of the set of traces which satisfy the property. In the robot example, traces might be sequences of visited locations, and we could specify that a particular location X is reached using the language consisting of all words which include the symbol X.

In order to develop algorithms operating on specifications, we need a finite, formal, and precisely-defined encoding of the desired property, or its equivalent language. Thus in this thesis the term *specification*, unless stated otherwise, means a finite representation of a language. Given a specification S, we write $L(S)$ for the language it defines. In practice we do not work with languages as explicit sets of words, since they could be very large or even infinite. Rather, we use more or less *implicit* representations which define a language through some other formalism like automata or logical formulas. Below, we describe the main types of specifications we use (see Hopcroft et al. [87] for more detailed discussions of automata and grammars).

Finite Automata

A *deterministic finite automaton (DFA)* consists of a finite alphabet Σ, a set of *states* Q, an initial state $q_0 \in Q$, a set of *accepting states* $F \subseteq Q$, and a *transition function* $\delta : Q \times \Sigma \to Q$. A DFA D *runs* on an input word $w \in \Sigma^*$ by starting in q_0 and reading each successive symbol of w, moving between states according to δ (i.e. at state s, if the next symbol is a, we move to the state $\delta(s, a)$). The sequence of states obtained in this way is the *path* corresponding to w. This terminology is suggested by the fact that we can view a DFA as a directed (multi-)graph whose vertices are states and whose edges represent transitions, labeled by the input symbol which triggers them: a path through the DFA for the word w is exactly a path through the graph (starting from q_0) whose edge labels spell out w. If the path ends at an

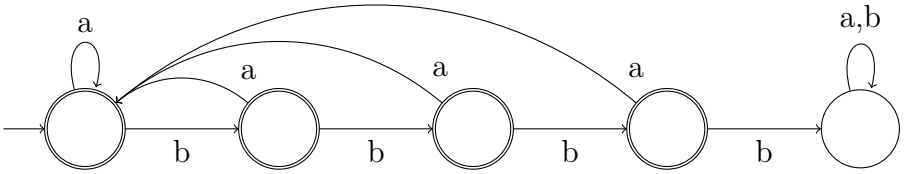


Figure 2.1: A DFA accepting all words in $\{a, b\}^*$ which do not have 4 bs in a row.

accepting state, we say it is an *accepting path*, and D *accepts* w; otherwise D *rejects* w. A DFA D defines a language $L(D)$ over Σ given by the set of all words accepted by D.

An example DFA over the alphabet $\Sigma = \{a, b\}$ is shown in Figure 2.1. We visualize the DFA as a graph in the standard way, with accepting states having a doubled border and the initial state indicated by an incoming arrow. For example, the DFA accepts the empty word λ, since the initial state is accepting, and also the word *bab*, since the second state from the left is accepting, but rejects the word *bbbba*, since the rightmost state is not accepting. It is easy to see that the language of this DFA consists exactly of those words which never have 4 or more consecutive bs.

A *nondeterministic finite automaton (NFA)* is a generalized DFA where we allow *nondeterministic transitions*: for a given state and input symbol, there can be multiple possible transitions (in the graph view, a vertex can have multiple outgoing edges with the same label). Thus the transition function δ becomes a *transition relation* $\delta \subseteq Q \times \Sigma \times Q$ specifying which states we can move to given the current state and input symbol. As a result, a single word $w \in \Sigma^*$ can now have multiple paths corresponding to it, and the acceptance condition for the NFA is that *at least one* of these paths is an accepting path.

There are well-known algorithms for performing various basic operations on finite automata: in particular, we can *complement* the language of a DFA by flipping the accepting/non-accepting states, and we can *intersect* the languages of two DFAs by constructing their *product DFA* [87].

Context-Free Grammars

A *context-free grammar (CFG)* consists of a finite alphabet Σ (of *terminal symbols*), a set of *variables* (or *nonterminal symbols*) V, a *start symbol* $S \in V$, and a set of *productions* $P \subseteq V \times (V \cup \Sigma)^*$. The productions are *rewrite rules* allowing a variable to be replaced by a sequence of terminal and nonterminal symbols. For example, the production $(X, aYbY)$, which we can write as the rule $X \to aYbY$, allows us to expand the word cXY into $caYbYY$. We say a word $w \in \Sigma^*$ can be *derived* from a CFG G if it is possible to reach w using the rewrite rules starting from the start symbol S. The language $L(G)$ of the grammar is the set of all derivable words in Σ^*.

A derivation of a word $w \in L(G)$ can be represented by a *derivation tree* with S as the root and the (ordered) children of a nonterminal node being the symbols it was expanded

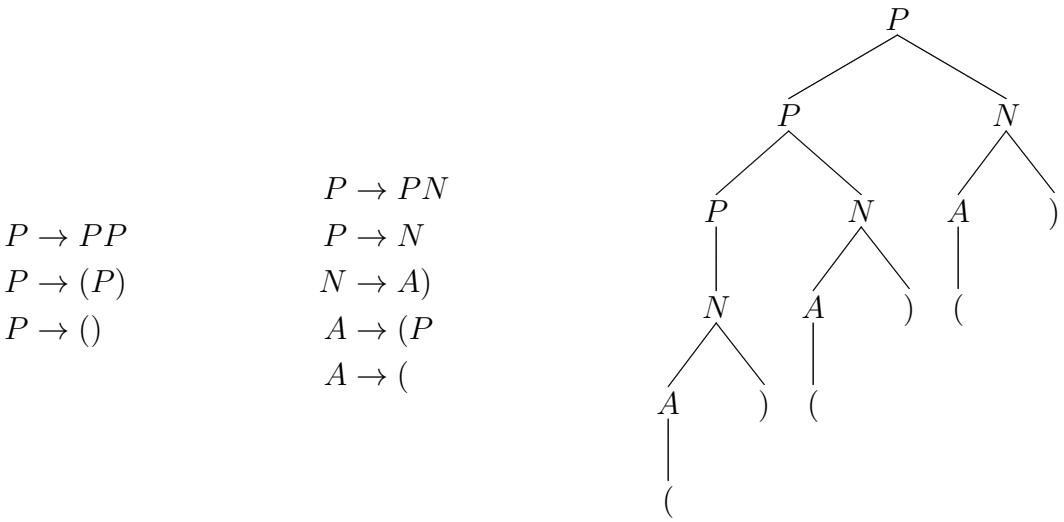


Figure 2.2: Ambiguous and unambiguous CFGs for nested parentheses, and an example derivation tree.

into by a production. It is possible for a single word to have multiple derivation trees, so that the grammar allows the word to be parsed in more than one way. When this happens we say that G is *ambiguous*. An example is shown in Figure 2.2: the grammar at left for correctly-nested parentheses is ambiguous, since the word ()()() can be parsed in two ways: either the first two pairs of parentheses share a parent in the tree, or the last two do. The grammar in the middle, on the other hand, has the same language but is unambiguous, with the unique derivation tree for ()()() shown on the right. This grammar also indicates how we can always write a CFG in a special form where the right-hand side of every production has at most two symbols (we will return to this as needed later) [87].

Boolean Formulas

Another important type of specification are *Boolean formulas* (equivalently, sentences of *propositional logic*) [17]. A Boolean formula is an expression built by applying the Boolean operations AND (*conjunction*, written $\wedge$), OR (*disjunction*, written $\vee$), and NOT (*negation*, written $\neg$) to Boolean *variables* which can take the values *true* ($\top$) and *false* ($\bot$). Given an *assignment* of truth values to the variables of a formula, the formula evaluates to true or false according to the usual semantics of the Boolean operations. For example, the formula $x \wedge \neg(y \wedge z)$ is true if x and y are true and z is false. Assignments which cause a formula to evaluate to true are called *satisfying assignments* or *models* of the formula. We can view a Boolean formula as defining a language of words of some fixed length n by mapping words of Σ^n to truth assignments of Boolean variables by a suitable encoding. The language of the formula is then simply all words whose encodings are satisfying assignments of the formula.

If a formula has a satisfying assignment it is said to be *satisfiable*, whereas if it has no satisfying assignments it is *unsatisfiable*. The problem SAT of determining whether

a Boolean formula is satisfiable is NP-complete (see Arora and Barak [8]). Despite this, *SAT solvers* using heuristic search techniques have been very successful in a wide variety of applications. For a comprehensive survey of the theory and practice of SAT and many of its generalizations, see Biere et al. [17].

We will occasionally make use of two normal forms of Boolean formulas. A formula in *conjunctive normal form (CNF)* is a conjunction of *clauses*, each of which is a disjunction of *literals*, which are variables and their negations. For example, $(x \vee \neg y) \wedge z$ is a CNF formula. *Disjunctive normal form (DNF)* is the dual form, being a disjunction of *cubes*, each of which is a conjunction of literals. For example, the negation of the formula above, $(\neg x \wedge y) \vee \neg z$, is a DNF formula. Any Boolean formula can be efficiently converted into CNF while preserving satisfiability [17].

Finally, we will also occasionally make use of *quantified* Boolean formulas (QBFs). The existential ($\exists$) and universal ($\forall$) quantifiers have their usual meanings: the formula $\exists x.\phi(x)$ is true when either $\phi(\bot)$ or $\phi(\top)$ are, and $\forall x.\phi(x)$ is true when both are. The problem QBF of determining whether a quantified Boolean formula is true (or satisfiable if it has free, i.e. unquantified, variables) is PSPACE-complete (again, see Arora and Barak [8]). A QBF is said to be in *prenex normal form* if it consists of a sequence of quantifiers followed by a quantifier-free formula; the latter is called the *matrix* of the formula. For example, $\exists x.\forall y.(x \vee y)$ is in prenex normal form, while the equivalent formula $\exists x.(x \vee \forall y.y)$ is not. Any QBF can be efficiently converted into prenex normal form with a CNF matrix while preserving truth/satisfiability [21].

2.4 Counting and Uniform Sampling

As we will see, algorithmic improvisation will require us to solve harder problems than simply finding a solution to a specification (i.e. a word in its language): we will need to *count* how many solutions there are, and *uniformly sample* from those solutions. There is a large body of work on these problems for different types of specifications, much of which we will build on below, so we summarize it here.

There is a close connection between counting and uniform sampling, going back to Wilf [184], who first showed in a general setting how algorithms for counting can be used to perform uniform sampling. The essential idea is to perform a *random walk on partial solutions*: if for example there are two ways A and B to extend our current partial solution, we use counting to determine how many solutions are consistent with A and how many are consistent with B, then pick A or B randomly with probability proportional to these counts. This type of construction has been used in many settings, for example by Hickey and Cohen [86] to do efficient uniform generation of strings from the languages of DFAs and unambiguous CFGs. The procedure was generalized in the context of SAT by Jerrum et al. [89], who showed that *approximate* counting suffices to do uniform sampling, and also that there is an efficient reduction in the opposite direction, from (almost-) uniform sampling to approximate counting.

For counting the languages of DFAs and unambiguous CFGs, there are classical polynomial-time algorithms by Hickey and Cohen [86] based on dynamic programming, who as mentioned above used them to do uniform sampling from such languages. In the case of grammars, faster algorithms were later developed, culminating in that of McKenzie [118]. For non-deterministic automata and ambiguous grammars, the problems are much more difficult, with counting being #P-complete and the best known sampling algorithms requiring quasi-polynomial ($n^{\Theta(\log n)}$) time [93, 80][1].

The problem of counting the number of solutions to a Boolean formula is #SAT, also called *model counting*. #SAT is a prototypical #P-complete problem [174], and there are many types of exact and approximate algorithms for it, most of which use SAT solvers internally (see Gomes et al. [74] for a survey). In fact, it had long been known theoretically that approximate model counting (with some probability of failure) and sampling could be done in polynomial time relative to an NP oracle (first observed explicitly by Jerrum et al. [89], drawing on Stockmeyer [164] and Valiant and Vazirani [175]), and more recently that this is true even for *exact* uniform sampling [15]. These algorithms were largely of theoretical interest since the NP queries they generated were too difficult for even modern SAT solvers. However, in the last several years significant progress has been made on practical approximate model counting algorithms using the same *universal hashing* paradigm as the theoretical algorithms, to the point where they can handle quite large problems. For a survey up to 2015, see Meel et al. [120]; however, this is an active research area, and there have already been significant further improvements, e.g. Chakraborty et al. [28] and Soos and Meel [161]. Details of the hashing-based approach, as well as a discussion of many of its applications, can be found in Meel [119].

We will also use an extension of model counting, *projected model counting*, which given a Boolean formula $\phi(\overline{x}, \overline{y})$ with two sets of variables asks how many assignments to $\overline{x}$ can be extended to a complete satisfying assignment of ϕ (the name arising since the problem counts models of ϕ *projected* onto a subset of its variables). The *projected uniform sampling* problem analogously asks us to uniformly sample from these assignments to $\overline{x}$. The projected model counting problem was originally introduced by Valiant [174] under the name #NSAT, for "nondeterministic SAT", although practical algorithms were developed only recently. The hashing-based model counting and sampling algorithms in particular can be straightforwardly extended to handle projection [29].

As a final note, we have worked on various further generalizations of model counting and sampling:

- *weighted* model counting and sampling, where different solutions contribute more or less to the total count, or are sampled with higher/lower probabilities, according to a specified weight function (useful for example in probabilistic inference) [26];

[1] In fact, it was very recently shown that *approximate* counting for NFAs can be done in polynomial time [6], which implies that sampling can also be done in polynomial time by the random-walk reduction.

- generating multiple samples at once when strict independence is not required, in particular when trying to cover the space of solutions evenly (e.g. in constrained-random verification or other test generation settings) [27];
- *maximum* model counting, which combines counting and optimization, asking for an assignment to one set of variables which maximizes the number of solutions in a second set (useful for quantitative information flow analysis of programs) [66].

Since we have not used these more general problems in our work on algorithmic improvisation so far, we do not discuss them in this thesis. However, they could well be useful for extended versions of control improvisation allowing greater control over the distribution of behaviors or with quantitative soft constraints, which we will discuss as future work in Chapter 3.

Part I

Theory

Chapter 3

Control Improvisation

In this chapter, we define and study *control improvisation* (CI), the core computational problem of algorithmic improvisation [63, 62]. CI provides the foundation for the more sophisticated versions of algorithmic improvisation we will consider in Chapters 4 and 5, but it is also practically useful in its own right. As we mentioned in the Introduction, its original application was computer music improvisation [44]; we will discuss two additional applications, robotic planning and human behavior modeling, in Chapters 6 and 7.

As we saw in Chapter 1, the fundamental goal in algorithmic improvisation is to construct an algorithm whose behavior is subject to hard, soft, and randomness constraints. In control improvisation, these behaviors are finite sequences of symbols from a finite alphabet, which can represent musical notes or actions for a robot, for example. The hard and soft constraints are defined using finite automata, Boolean formulas, or some other specification formalism, while the randomness constraint simply requires that no sequence be generated with too high or too low probability. A probabilistic algorithm which generates such sequences, while satisfying all three types of constraints, is called an *improviser*. The *control improvisation problem*, then, is to construct an improviser for a given set of constraints. In fact, we do not simply want improvisers for particular constraints, but rather a synthesis algorithm: a general procedure for constructing improvisers from specifications, which we call an *improvisation scheme*. We define all of these concepts formally in Section 3.1.

We then proceed to study the theory of CI, starting in Section 3.2 with the most fundamental question: when is a CI problem solvable? This is a more subtle question than in traditional synthesis problems, where solving the problem typically amounts to finding a single solution to a (possibly quantified) constraint satisfaction problem. For CI, it is not enough for the hard and soft constraints to have a solution, since the randomness constraint requires us to produce *many* different sequences, and therefore we must have many solutions available to choose from. We show exactly how many solutions are required, giving a precise characterization of when CI problems are feasible.

Our proof of the CI feasibility conditions is constructive, providing a recipe for how to build improvisers. In Section 3.3, we turn this construction into a generic improvisation scheme, applying to any kind of specification which supports several operations: intersection,

difference, counting, and uniform sampling. When these operations can be implemented efficiently, our result yields an efficient improvisation scheme. This result also allows us to show that under very general conditions (i.e., for any kind of specification which can be checked on a trace in polynomial time), the complexity of CI is at most #P.

Next, in Section 3.4 we investigate the complexity of CI in more detail, establishing it to be either P or #P for several practical types of specifications. Specifically, we consider finite automata, context-free grammars, and Boolean formulas. For deterministic finite automata and unambiguous context-free grammars, we construct polynomial-time improvisation schemes. For nondeterministic finite automata and general context-free grammars, we show that CI is #P-equivalent; interestingly, this is also true when *both* the hard and soft constraints are unambiguous grammars. For Boolean formulas, CI is trivially #P-equivalent, but we give an *approximate* improvisation scheme using only an NP oracle, which therefore can be implemented using a SAT solver.

Finally, in Section 3.5 we discuss a more general version of CI which allows multiple hard and soft constraints: *multi-constraint control improvisation (MCI)*. This generalization arises naturally in some applications, for example the human-like lighting control application we will discuss in Chapter 7. We perform much of the same analysis we did earlier for CI, investigating when MCI problems are feasible and the complexity of solving them. However, MCI turns out to be far more involved than CI, and we are not able to establish simple feasibility conditions or pin down its complexity exactly. We do show that for DFA specifications the problem is already #P-hard, suggesting that allowing multiple constraints makes the CI problem significantly more difficult.

We conclude the chapter in Section 3.6 with a summary and directions for future work.

3.1 Problem Definition

We start by formally defining the control improvisation problem and its associated concepts. As was discussed in Chapter 1, the definitions have changed in some (mostly minor) ways over time. We will use the definitions from Fremont et al. [62], noting from time to time how they differ from the earlier definitions in Donzé et al. [43, 44] and Fremont et al. [63] and the reasons for the change.

3.1.1 Elements of the Definition

The control improvisation problem is defined in terms of several elements. First, we have a finite alphabet Σ of symbols which the improviser may output. For example, for a robot moving in a two-dimensional (2D) gridworld, Σ could be the set of possible movements $\{North, South, East, West, Stop\}$, so that a word over Σ corresponds to a behavior of the robot.

Next, we need representations of the hard and soft constraints. The hard constraint is straightforward, since it simply restricts the output of the improviser, saying that all

generated sequences must have some desired property (in the robot example, we could require that the robot not move into a location where there is an obstacle, for instance). We can therefore represent it by a *hard specification* $\mathcal{H}$ over Σ, whose language $L(\mathcal{H}) \subseteq \Sigma^*$ consists of the words having the property. Together with bounds defining the desired lengths, this defines the *improvisations*, the words the improviser is allowed to generate:

Definition 3.1. Fix a finite alphabet Σ, a *hard specification* $\mathcal{H}$ over Σ, and length bounds $m, n \in \mathbb{N}$. An *improvisation* is any word $w \in L(\mathcal{H})$ such that $m \leq |w| \leq n$. We write I for the set of all improvisations.

The soft constraint is similar to the hard constraint in that it specifies a property which the output of the improviser must satisfy, except that the property need only hold with at least some probability $1 - \epsilon$. Therefore, we can represent the soft constraint by a *soft specification* $\mathcal{S}$ over Σ, together with the *error probability* ϵ. We call improvisations which satisfy the soft specification *admissible*:

Definition 3.2. Fix a *soft specification* $\mathcal{S}$ over Σ and an *error probability* $\epsilon \in [0, 1]$. An improvisation $w \in I$ is *admissible* if $w \in L(\mathcal{S})$, and we write A for the set of all admissible improvisations.

Finally, the randomness constraint requires that every improvisation be generated with a probability within a desired range $[\lambda, \rho]$ (with $0 \leq \lambda \leq \rho \leq 1$). This requirement is designed to accommodate two different needs for randomness. In fuzz testing, for example, in order to ensure coverage we might put $\lambda > 0$ to require that every test case can be generated. By contrast, in music improvisation or robotic planning it is not important that every possible melody or plan can be generated; rather, we simply want no single improvisation to arise too frequently. So in such applications we might put $\lambda = 0$ but $\rho < 1$. Our definition allows any combination of these two cases. Note that if there are N improvisations (i.e. $|I| = N$), then setting λ or ρ equal to $1/N$ forces the distribution to be uniform. So random sampling as used in fuzz testing or constrained-random verification is a special case of control improvisation. We also note that other randomness requirements are possible, for example ensuring variety by imposing some minimum distance between generated improvisations. This could be reasonable in a setting such as music or robotics where there is a natural metric on the space of improvisations, but we choose to keep our setting general and not assume such a metric.

Running Example. Throughout the chapter, we will illustrate our concepts and algorithms with a simple example. Suppose we want to generate variations of the binary string $s = 001$ which have length 3, subject to the constraint that there cannot be two consecutive 1s. We can formalize this as a CI problem as follows:

Alphabet. $\Sigma = \{0, 1\}$, since we want to generate binary strings.

Length Bounds. $m = n = 3$, so that the length must be exactly 3.

Hard Constraint. $\mathcal{H}$ is a DFA accepting all strings that do not have two 1s in a row.

Soft Constraint. $\mathcal{S}$ is a DFA accepting words with Hamming distance at most 1 from s, to ensure that our variations are usually similar to s. We can put the error probability $\epsilon = 1/4$ for the similarity property to hold at least 3/4 of the time.

Randomness Constraint. $\rho = 1/4$, which implies the improviser can generate at least 4 words. We leave $\lambda = 0$ since we only care that sufficiently many improvisations can be generated.

Then for example the word 000 is an admissible improvisation, satisfying both hard and soft constraints, and so is in A. The word 010 on the other hand satisfies $\mathcal{H}$ but not $\mathcal{S}$ (having Hamming distance 2 from s), so it is in I but not A. All together we have $I = \{000, 001, 010, 100, 101\}$ and $A = \{000, 001, 101\}$, while words like 011 and 00100 are not improvisations at all.

3.1.2 The Control Improvisation Problem

Combining all of the elements above, we obtain our definitions of an acceptable distribution over improvisations and thus of an improviser:

Definition 3.3. Given a *CI instance* $\mathcal{C} = (\mathcal{H}, \mathcal{S}, m, n, \epsilon, \lambda, \rho)$ with elements as described above, a distribution on words $D : \Sigma^* \to [0, 1]$ is an *improvising distribution* if it satisfies the following requirements:

Hard constraint: $\Pr[w \in I \mid w \leftarrow D] = 1$

Soft constraint: $\Pr[w \in A \mid w \leftarrow D] \geq 1 - \epsilon$

Randomness constraint: $\forall w \in I,\ \lambda \leq D(w) \leq \rho$

If an improvising distribution exists, we say that $\mathcal{C}$ is *feasible*. An *improviser* for a feasible $\mathcal{C}$ is a probabilistic algorithm, taking no input and with finite expected runtime, whose output distribution is an improvising distribution.

Running Example. The CI instance $\mathcal{C} = (\mathcal{H}, \mathcal{S}, 3, 3, 1/4, 0, 1/4)$ described in our running example above is feasible: for example, we could generate each of the improvisations 000, 001, 101, and 100 with probability 1/4. The first three are admissible, so we would generate an admissible improvisation with probability $3/4 = 1 - \epsilon$ and satisfy the soft constraint. On the other hand, if we tightened the soft constraint by setting $\epsilon = 0$, the problem would be infeasible: $\epsilon = 0$ means we can only generate admissible improvisations, and since there are only 3 of these ($|A| = 3$), we cannot possibly give them all probability at most $\rho = 1/4$. We could make the problem feasible again by relaxing the randomness constraint, reducing ρ to 1/3 so that we could simply sample from $\{000, 001, 101\}$ uniformly at random. Note

that this would not be an improvising distribution if $\lambda > 0$, since the improvisation 010 is generated with probability 0.

Definition 3.4. Given a CI instance $\mathcal{C}$, the *control improvisation (CI)* problem is to decide whether $\mathcal{C}$ is feasible, and if so to generate an improviser for $\mathcal{C}$. For this problem, the size of the instance $\mathcal{C}$ is measured with m and n being represented in unary, and ϵ, λ, and ρ represented in binary.

We measure the size of CI instances as being linear in the values of m and n (rather than their binary encodings) to simplify our discussion of the computational complexity of the CI problem. Since an improviser must generate words of length at least m, its runtime cannot be less than m. So if we encoded m and n in binary in a CI instance, all improvisers would have to run in at least exponential time in the size of the instance (whereas, of course, we want improvisers whose runtimes are small polynomials in $|\mathcal{H}|$, $|\mathcal{S}|$, etc.). By encoding m and n in unary, we allow a linear-time improviser, for example, to take time proportional to n while still being linear in the size of the hard and soft specifications.

As mentioned above, we are interested not only in solving particular CI problems, but general algorithms for entire *classes* of such problems. The most natural classes of CI problems are those whose hard and soft constraints are defined by specifications of a particular type. For example, we can consider CI problems defined by DFAs, or by CFGs. Such classes allow us to study the trade-off between expressivity and complexity: small finite automata can only express simple properties, but admit very efficient algorithms for reasoning about them. Conversely, using Boolean formulas we can encode very complex constraints in a compact form, but analyzing the constraints becomes much more difficult.

Definition 3.5. If $\mathcal{A}$ and $\mathcal{B}$ are classes of specifications, CI($\mathcal{A}$,$\mathcal{B}$) is the class of CI instances $\mathcal{C} = (\mathcal{H}, \mathcal{S}, m, n, \epsilon, \lambda, \rho)$ where $\mathcal{H} \in \mathcal{A}$ and $\mathcal{S} \in \mathcal{B}$. When discussing decision problems, we use the same notation for the feasibility problem associated with the class: given $\mathcal{C} \in$ CI($\mathcal{A}$,$\mathcal{B}$), decide whether it is feasible.

For example, CI(UCFG,DFA) is the class of instances where the hard specification is an unambiguous context-free grammar and the soft specification is a deterministic finite automaton.

Given a class of CI problems, we call an algorithm for solving them an *improvisation scheme*. This is exactly analogous to a classical synthesis algorithm: for example, an algorithm for reactive synthesis from LTL formulas takes a specification in the form of an LTL formula and generates an implementation satisfying it [136]. Likewise, an improvisation scheme takes a specification in the form of a CI instance and generates an improviser satisfying its requirements. For such a scheme to be efficient, the scheme itself as well as the improvisers it produces should have small runtimes with respect to the size of the input specification.

Definition 3.6. A *polynomial-time improvisation scheme* for a class $\mathcal{P}$ of CI instances is an algorithm S with the following properties:

Correctness: For every instance $\mathcal{C} \in \mathcal{P}$, if $\mathcal{C}$ is feasible then $S(\mathcal{C})$ is an improviser for $\mathcal{C}$, and otherwise $S(\mathcal{C}) = \perp$ (a symbol indicating no improviser exists).

Scheme efficiency: There is a polynomial $p : \mathbb{R} \to \mathbb{R}$ such that the runtime of S on every $\mathcal{C} \in \mathcal{P}$ is at most $p(|\mathcal{C}|)$.

Improviser efficiency: There is a polynomial $q : \mathbb{R} \to \mathbb{R}$ such that for every $\mathcal{C} \in \mathcal{P}$, if $G = S(\mathcal{C}) \neq \perp$ then G has expected runtime at most $q(|\mathcal{C}|)$.

Exponential-time and *polynomial-space improvisation schemes* are defined analogously.

Note that both efficiency requirements are necessary: otherwise, we could have a trivial scheme which did nothing itself but offloaded an exponential search into the improviser it generates.

As we have now seen, the control improvisation problem is a problem of randomized synthesis: taking hard, soft, and randomness specifications and generating an improviser satisfying them. An improvisation scheme is a uniform way to do this for an entire class of specifications. In the remainder of the chapter, we will study theoretically when CI problems can be solved and which classes of problems admit efficient improvisation schemes.

3.2 Existence of Improvisers

A control improvisation problem need not have a solution: for example, if there are no words satisfying the hard specification, obviously an improviser cannot exist. However, there are more subtle cases: as we saw in the running example above, even if the hard, soft, and randomness requirements are all satisfiable individually, they can conflict with each other. In fact, part of the utility of the formulation of the CI problem is that it allows studying such trade-offs: how much can we strengthen a soft requirement while maintaining a given amount of randomness, or how much randomness can we introduce into a system without compromising its correctness? Thus, our first goal studying the theory of CI is to determine precise conditions for when a CI problem can be solved.

As we saw above, a CI instance being feasible means that an improvising distribution exists. A first observation is that since the set of improvisations I is finite, we can parametrize all distributions over I with finitely many variables representing the probability of each word $w \in I$. Then the requirements in Definition 3.3 on an improvising distribution are just linear inequalities, and so the existence of such a distribution is equivalent to the feasibility of a linear program. We will return to this formulation in Section 3.5, where the linear program will be written out formally for the more general case of multi-constraint control improvisation. However, for ordinary CI problems it is more instructive to try to build an improvising distribution directly. We can do this in a constructive way, thereby yielding not just an improvising distribution but an improviser (an algorithm) as well.

The intuition behind our construction is as follows. To satisfy the hard constraint, we can generate only members of I; within I, we should prefer members of A, which satisfy the soft

specification, to members of $I \setminus A$, which do not. Therefore, to satisfy the soft specification with as high a probability as possible, we assign the minimum allowed probability λ to each element of $I \setminus A$, spreading the remainder of the probability uniformly over A. This will certainly satisfy the hard constraint and the lower bound of the randomness constraint, but may violate the upper bound, since each element of A could receive more than probability ρ. In that case, we instead clamp the probabilities of each element of A at ρ, and spread the remainder uniformly over $I \setminus A$. In order for this strategy to yield an improvising distribution, various conditions on the sizes of I and A must be satisfied. These conditions turn out to be necessary for an improvising distribution, so we get an exact characterization of feasibility:

Theorem 3.1. *For any $\mathcal{C} = (\mathcal{H}, \mathcal{S}, m, n, \epsilon, \lambda, \rho)$, the following are equivalent:*

1. *$\mathcal{C}$ is feasible.*
2. *The following inequalities hold:*
   a) $1/\rho \leq |I| \leq 1/\lambda$
   b) $(1-\epsilon)/\rho \leq |A|$
   c) $|I| - |A| \leq \epsilon/\lambda$
3. *There is an improviser for $\mathcal{C}$.*

Proof. **(1)⇒(2):** Suppose D is an improvising distribution. Then

$$\rho|I| = \sum_{w \in I} \rho \geq \sum_{w \in I} D(w) = \Pr[w \in I \mid w \leftarrow D] = 1,$$

so $|I| \geq 1/\rho$. Similarly,

$$\lambda|I| = \sum_{w \in I} \lambda \leq \sum_{w \in I} D(w) = \Pr[w \in I \mid w \leftarrow D] = 1,$$

so $|I| \leq 1/\lambda$. Since $A \subseteq I$, we also have

$$\rho|A| = \sum_{w \in A} \rho \geq \sum_{w \in A} D(w) = \Pr[w \in A \mid w \leftarrow D] \geq 1 - \epsilon,$$

and therefore $|A| \geq (1-\epsilon)/\rho$. Finally, we have

$$\begin{aligned}
\lambda|I \setminus A| &= \sum_{w \in I \setminus A} \lambda \\
&\leq \sum_{w \in I \setminus A} D(w) \\
&= \Pr[w \in I \setminus A \mid w \leftarrow D] \\
&= \Pr[w \in I \mid w \leftarrow D] - \Pr[w \in A \mid w \leftarrow D] \\
&\leq 1 - (1-\epsilon) = \epsilon,
\end{aligned}$$

so $|I| - |A| \leq \epsilon/\lambda$.

(2)⇒(3): Define $\epsilon_{\text{opt}} = \max(1 - \rho|A|, \lambda|I \setminus A|)$ (the notation for this quantity will be explained later). Since $|I \setminus A| \leq |I| \leq 1/\lambda$, we have $0 \leq \epsilon_{\text{opt}} \leq 1$. Let D be the distribution on I which picks uniformly from A with probability $1 - \epsilon_{\text{opt}}$ and otherwise picks uniformly from $I \setminus A$. Note that this distribution is well-defined, since if $A = \emptyset$ then $\epsilon_{\text{opt}} = 1$, and if $I \setminus A = \emptyset$ then $\rho|A| = \rho|I| \geq 1$ and so $\epsilon_{\text{opt}} = 0$. Clearly, D satisfies the hard constraint.

Now if $\epsilon_{\text{opt}} = 1 - \rho|A|$ we have $\epsilon_{\text{opt}} \leq 1 - \rho \cdot (1 - \epsilon)/\rho = \epsilon$. Otherwise, $\epsilon_{\text{opt}} = \lambda|I \setminus A| \leq \lambda(\epsilon/\lambda) = \epsilon$. So in either case we have $\Pr[w \in A \mid w \leftarrow D] = 1 - \epsilon_{\text{opt}} \geq 1 - \epsilon$, and D satisfies the soft constraint.

For any $w \in A$, we have $D(w) = (1 - \epsilon_{\text{opt}})/|A| \leq (1 - (1 - \rho|A|))/|A| = \rho$. If $\epsilon_{\text{opt}} = 1 - \rho|A|$, then $D(w) = \rho \geq \lambda$; otherwise $\epsilon_{\text{opt}} = \lambda|I \setminus A|$, so $D(w) = (1 - \lambda|I \setminus A|)/|A| = (1 - \lambda|I| + \lambda|A|)/|A| \geq (1 - 1 + \lambda|A|)/|A| = \lambda$. Thus $D(w) \geq \lambda$ in either case. Similarly, for any $w \in I \setminus A$ we have $D(w) = \epsilon_{\text{opt}}/|I \setminus A| \geq \lambda|I \setminus A|/|I \setminus A| = \lambda$. If $\epsilon_{\text{opt}} = 1 - \rho|A|$, then $D(w) = (1 - \rho|A|)/|I \setminus A| = (1 - \rho|A|)/(|I| - |A|) \leq (1 - \rho|A|)/((1/\rho) - |A|) = \rho$; otherwise $\epsilon_{\text{opt}} = \lambda|I \setminus A|$, so $D(w) = (\lambda|I \setminus A|)/|I \setminus A| = \lambda \leq \rho$. Therefore for any $w \in I$ we always have $\lambda \leq D(w) \leq \rho$, and thus D satisfies the randomness constraint.

This shows that D is an improvising distribution. Since it has finite support and rational probabilities, there is a probabilistic algorithm with finite expected runtime sampling from it, and this algorithm is an improviser for $\mathcal{C}$.

(3)⇒(1): The output distribution of an improviser is an improvising distribution, so if an improviser for $\mathcal{C}$ exists then $\mathcal{C}$ must be feasible. □

Remark. In the Theorem, if $\lambda = 0$, i.e. we impose no lower bound on the probabilities of individual improvisations, we treat division by zero as yielding ∞, so that both the inequalities involving λ are trivially satisfied. The remaining inequalities are precisely those given in the corresponding theorem in the first paper on the theory of CI [63, Theorem 3.1], which did not allow a lower bound in the randomness requirement.

Theorem 3.1 shows that whenever improvisers exist, there is one of a quite simple form: it flips a biased coin to pick one of two sets (A or $I \setminus A$), and then samples from that set uniformly at random. Of course there can be many other improvising distributions which assign a variety of probabilities between λ and ρ, but one of this form is guaranteed to exist. This suggests that algorithms for solving CI problems should be closely related to algorithms for uniform sampling from constraints, which in fact we will see later in the chapter.

Running Example. Recall that for our running example we have $I = \{000, 001, 010, 100, 101\}$ and $A = \{000, 001, 101\}$, so $|I| = 5$ and $|A| = 3$. We noted above that this CI problem is feasible with $\epsilon = \rho = 1/4$ and $\lambda = 0$. This is consistent with Theorem 3.1: the inequalities involving λ drop out since $\lambda = 0$, and we have $1/\rho = 4 \leq |I|$ and $(1 - \epsilon)/\rho = 3 \leq |A|$. On the other hand, this last inequality would not be satisfied if we decreased ϵ below $1/4$, and indeed as we saw above the problem does become infeasible in that case.

Since the intuition behind our construction was to satisfy the soft constraint with as high a probability as possible, we might expect it to be somehow optimal among all improvising distributions. In fact, the error probability ϵ_{opt} achieved by our construction is the smallest error probability consistent with the hard and randomness constraints, justifying the notation we use for it:

Corollary 3.1. *Let* $\epsilon_{\text{opt}} = \max(1 - \rho|A|, \lambda|I \setminus A|)$. *Then* $\mathcal{C}$ *is feasible if and only if* $1/\rho \leq |I| \leq 1/\lambda$ *and* $\epsilon \geq \epsilon_{\text{opt}}$.

Proof. Immediate from Theorem 3.1, noting that $\epsilon \geq \epsilon_{\text{opt}}$ if and only if inequalities (2b) and (2c) in the Theorem hold. □

Running Example. With $\rho = 1/4$ and $\lambda = 0$ as above, we have $\epsilon_{\text{opt}} = \max(1/4, 0) = 1/4$, mirroring our observation above that the problem becomes infeasible with $\epsilon < 1/4$. If we increased λ to $1/5$, since $|I \setminus A| = 2$ we would have $\epsilon_{\text{opt}} = \max(1/4, 2/5) = 2/5$ and so it would not be possible to generate an admissible improvisation more than $3/5$ of the time. This makes sense, since $|I| = 5$ and so to satisfy $\lambda = 1/5$ we will need to use the uniform distribution over I, obtaining an admissible improvisation with probability $|A|/|I| = 3/5$.

3.3 A Generic Improvisation Scheme

Now that we understand when CI problems are solvable, the next step is to develop algorithms for solving them. The proof of Theorem 3.1 suggests a generic procedure, requiring a few basic operations on specifications that were used in the construction of an improvising distribution:

Intersection: Given two specifications $\mathcal{X}$ and $\mathcal{Y}$, compute a specification $\mathcal{Z}$ such that $L(\mathcal{Z}) = L(\mathcal{X}) \cap L(\mathcal{Y})$.

Difference: Given two specifications $\mathcal{X}$ and $\mathcal{Y}$, compute a specification $\mathcal{Z}$ such that $L(\mathcal{Z}) = L(\mathcal{X}) \setminus L(\mathcal{Y})$.

Counting: Given a specification $\mathcal{X}$ and $n, m \in \mathbb{N}$ in unary, compute $|L(\mathcal{X}) \cap \Sigma^{m:n}|$.

Uniform Sampling: Given a specification $\mathcal{X}$ and $n, m \in \mathbb{N}$ in unary, sample uniformly at random from $L(\mathcal{X}) \cap \Sigma^{m:n}$.

If these operations can be implemented efficiently for a particular class of specifications, then we can efficiently solve the corresponding CI problems:

Theorem 3.2. *Suppose* Spec *is a class of specifications that supports the operations above. Suppose further that the operations can be done in polynomial time (expected time for uniform sampling). Then there is a polynomial-time improvisation scheme for* CI(Spec,Spec).

Proof. Given a problem $\mathcal{C} \in \text{CI}(\text{SPEC},\text{SPEC})$ with $\mathcal{C} = (\mathcal{H}, \mathcal{S}, m, n, \epsilon, \lambda, \rho)$, the scheme works as follows. First, we construct representations of all the sets we need. Note that $L(\mathcal{H}) \cap \Sigma^{m:n} = I$ by definition. Applying intersection to $\mathcal{H}$ and $\mathcal{S}$, we get a specification $\mathcal{A}$ such that $L(\mathcal{A}) \cap \Sigma^{m:n} = L(\mathcal{H}) \cap L(\mathcal{S}) \cap \Sigma^{m:n} = I \cap L(\mathcal{S}) = A$. Finally, applying difference to $\mathcal{H}$ and $\mathcal{S}$ gives a specification $\mathcal{B}$ such that $L(\mathcal{B}) \cap \Sigma^{m:n} = I \setminus A$.

Next, applying counting to $\mathcal{H}$ and $\mathcal{A}$ with bounds n and m, we compute $|I|$ and $|A|$. We can then check whether the inequalities in Theorem 3.1 are satisfied. If not, then $\mathcal{C}$ is not feasible and we return $\perp$. Otherwise, $\mathcal{C}$ is feasible and we will build an improviser sampling from the same distribution constructed in the proof of Theorem 3.1. As there, let $\epsilon_{\text{opt}} = \max(1 - \rho|A|, \lambda|I \setminus A|)$ (easily computed since we know $|I|$ and $|A|$, and $|I \setminus A| = |I| - |A|$), and let D be the distribution on I that with probability $1 - \epsilon_{\text{opt}}$ picks uniformly from A and otherwise picks uniformly from $I \setminus A$. Since the inequalities in Theorem 3.1 are true, its proof shows that D is an improvising distribution for $\mathcal{C}$.

We can easily build a probabilistic algorithm G sampling from D: it simply flips a coin, applying uniform sampling to $\mathcal{A}$ with probability $1 - \epsilon_{\text{opt}}$ and otherwise applying uniform sampling to $\mathcal{B}$ (with length bounds m and n). Since $L(\mathcal{A}) \cap \Sigma^{m:n} = A$ and $L(\mathcal{B}) \cap \Sigma^{m:n} = I \setminus A$, the output distribution of G is D and so G is an improviser for $\mathcal{C}$.

This procedure is clearly correct. Since the intersection and difference operations take polynomial time, the constructed specifications $\mathcal{A}$ and $\mathcal{B}$ have sizes polynomial in $|\mathcal{H}|$ and $|\mathcal{S}|$, and thus in $|\mathcal{C}|$. So the subsequent counting and sampling operations performed on these specifications will also be polynomial in $|\mathcal{C}|$ (recalling that n and m are encoded in unary). In particular, the computed values of $|I|$ and $|A|$ have polynomial bitwidth, so the same is true of ϵ_{opt}, and the arithmetic performed by the procedure also takes time polynomial in $|\mathcal{C}|$. Therefore in total the procedure runs in polynomial time. The improvisers generated by the procedure run in expected polynomial time, since the bitwidth of ϵ_{opt} is polynomial and uniform sampling takes expected polynomial time. So the procedure is a polynomial-time improvisation scheme. □

Running Example. Recall that for our running example $\mathcal{C} = (\mathcal{H}, \mathcal{S}, 3, 3, 1/4, 0, 1/4)$, we have $I = \{000, 001, 010, 100, 101\}$ and $A = \{000, 001, 101\}$. In Section 3.4.1 we will see that all the operations needed by Theorem 3.2 can be performed efficiently for DFAs like $\mathcal{H}$ and $\mathcal{S}$. Applying intersection and difference, we obtain DFAs $\mathcal{A}$ and $\mathcal{B}$ such that $L(\mathcal{A}) = A$ and $L(\mathcal{B}) = I \setminus A = \{010, 100\}$. Applying counting to $\mathcal{H}$ and $\mathcal{A}$ we find that $|I| = 5$ and $|A| = 3$, so the inequalities in Theorem 3.1 are satisfied. Next we compute $\epsilon_{\text{opt}} = \max(1 - \rho|A|, \lambda|I \setminus A|) = \max(1/4, 0) = 1/4$. Finally, we return an improviser G that samples uniformly from $L(\mathcal{A})$ with probability $1 - \epsilon_{\text{opt}} = 3/4$ and from $L(\mathcal{B})$ with probability $\epsilon_{\text{opt}} = 1/4$. So G returns 000, 001, and 101 with probability $3/4 \cdot 1/3 = 1/4$ each, and 010 and 100 with probability $1/4 \cdot 1/2 = 1/8$ each. This distribution satisfies the hard, soft, and randomness constraints, so it is indeed an improviser for $\mathcal{C}$.

Remark. Note that all 4 types of operations used in Theorem 3.2 are in some sense necessary. As we will see later, we can use the fact that the feasibility of a CI instance depends on the sizes of I and A to reduce counting to CI feasibility. Once the size of a set S is known,

uniform sampling becomes a special case of CI, simply letting the hard specification encode S and putting $\lambda = \rho = 1/|S|$. Finally, it is easy to see how CI feasibility can be used to solve decision versions of intersection and difference, building CI instances which are feasible if and only if two sets have nonempty intersection or difference.

Although Theorem 3.2 was stated in terms of polynomial time for simplicity, its construction only uses the operations on specifications as black boxes, and therefore relativizes to any amount of computational resources needed to perform the operations. More precisely, if the operations can be done in polynomial time relative to an oracle $\mathcal{O}$, then the Theorem yields an improvisation scheme that is polynomial-time relative to $\mathcal{O}$ (where both the scheme and the improvisers it generates may query $\mathcal{O}$). This closely relates the complexity of CI to the complexities of the 4 operations, and in particular to counting and sampling, since these are typically the most difficult (e.g. for Boolean formulas, intersection and difference are straightforward but counting and sampling are very hard). In the next section we will follow this approach to determine the complexity of CI for a variety of different types of specifications.

However, without any reasoning about specific types of specifications we can already use Theorem 3.2 to upper bound the complexity of CI for a very broad range of specifications: any property testable in polynomial time. For such specifications, counting and sampling can be done with a $\#\mathsf{P}$ oracle, which leads to a polynomial-time improvisation scheme relative to a $\#\mathsf{P}$ oracle. Applying Theorem 3.2 abstractly is somewhat tricky since we need to keep track of the runtimes of the polynomial-time algorithms testing the specifications, so we again implement the construction used in its proof directly. In fact, using Toda's theorem we can generalize the result so that it applies to any specification testable using a PH oracle:

Theorem 3.3. *Suppose* Spec *is a class of specifications such that membership in the language of any $\mathcal{X} \in$ Spec can be decided in polynomial time relative to a PH oracle. Then* CI(Spec,Spec) *has an improvisation scheme that is polynomial-time relative to a $\#\mathsf{P}$ oracle.*

Proof. By assumption there is a polynomial-time algorithm $M^{\mathsf{PH}}(w, \mathcal{X})$ that decides whether $w \in L(\mathcal{X})$ for any specification $\mathcal{X} \in$ Spec. Then $I_{\mathcal{C}} = \{w \in L(\mathcal{H}_{\mathcal{C}}) \,|\, m_{\mathcal{C}} \leq |w| \leq n_{\mathcal{C}}\} = \{w \in \Sigma^* \,|\, m_{\mathcal{C}} \leq |w| \leq n_{\mathcal{C}} \wedge M^{\mathsf{PH}}(w, \mathcal{H}_{\mathcal{C}}) = 1\}$, so whether $w \in I_{\mathcal{C}}$ is decidable in time polynomial in $|w|$ and $|\mathcal{C}|$ relative to a PH oracle. Similarly, $A_{\mathcal{C}} = I_{\mathcal{C}} \cap L(\mathcal{S}_{\mathcal{C}}) = I_{\mathcal{C}} \cap \{w \in \Sigma^* \,|\, M^{\mathsf{PH}}(w, \mathcal{S}_{\mathcal{C}}) = 1\}$, so whether $w \in A_{\mathcal{C}}$ is decidable in time polynomial in $|w|$ and $|\mathcal{C}|$ relative to a PH oracle. Since any $w \in I_{\mathcal{C}}$ has length at most $n_{\mathcal{C}}$, and in particular polynomial in $|\mathcal{C}|$, this shows that the relations $R_I = \{(\mathcal{C}, w) | w \in I_{\mathcal{C}}\}$, $R_A = \{(\mathcal{C}, w) | w \in A_{\mathcal{C}}\}$, and $R_{I \setminus A} = \{(\mathcal{C}, w) | w \in I_{\mathcal{C}} \setminus A_{\mathcal{C}}\}$ are $\mathsf{NP}^{\mathsf{PH}}$-relations[1]. Therefore computing $|I_{\mathcal{C}}| = |\{w \in \Sigma^* \,|\, (\mathcal{C}, w) \in R_I\}|$ is a $\#\mathsf{P}^{\mathsf{PH}}$ problem, and likewise for $|A_{\mathcal{C}}|$. Furthermore, sampling uniformly from $A_{\mathcal{C}}$ and $I_{\mathcal{C}} \setminus A_{\mathcal{C}}$ is equivalent to sampling uniformly from the witnesses of $\mathcal{C}$ under the relations R_A and $R_{I \setminus A}$ respectively. This can be done in polynomial expected time relative to a $\#\mathsf{P}^{\mathsf{PH}}$ oracle (relativizing the algorithms of Jerrum et al. [89] or Bellare et al. [15]). So the construction in Theorem 3.2

[1] In the terminology of Bellare et al. [15], relativized to the PH oracle; these relations (unrelativized) are called p-relations by Jerrum et al. [89].

yields an improvisation scheme for CI(Spec,Spec) that is polynomial-time relative to a $\#\mathsf{P}^{\mathsf{PH}}$ oracle.

To remove the PH oracle, note that $\mathsf{PP}^{\mathsf{PH}} \subseteq \mathsf{P}^{\mathsf{PP}}$ by the stronger form of Toda's theorem [171]. Since $\#\mathsf{P}^{\mathsf{PH}} \subseteq \mathsf{FP}^{\mathsf{PP}^{\mathsf{PH}}}$ by a standard binary search argument, we have $\#\mathsf{P}^{\mathsf{PH}} \subseteq \mathsf{FP}^{\mathsf{PP}^{\mathsf{PH}}} \subseteq \mathsf{FP}^{\mathsf{P}^{\mathsf{PP}}} = \mathsf{FP}^{\mathsf{PP}} \subseteq \mathsf{FP}^{\#\mathsf{P}}$. So the polynomial-time improvisation scheme (and the improvisers it generates) can simulate the $\#\mathsf{P}^{\mathsf{PH}}$ oracle using only a $\#\mathsf{P}$ oracle. □

3.4 Complexity of Control Improvisation

By Theorem 3.3, for a very wide class of specifications the complexity of CI is between P and #P. In this section we examine several natural classes of CI problems and pin down their complexity as being at one end or the other of this range.

3.4.1 Finite Automata

Finite automata are a simple and tractable form of specification capturing the notion of *finite-memory* properties. These are properties which can be tested by examining a word left-to-right, only being able to remember a finite number of already-seen symbols. Examples of such specifications include properties of a word w which hold if and only if each subword of w of a fixed constant length satisfied some condition. Such properties are useful, for example, in music improvisation:

Example (Factor Oracles). A practical approach to computer music improvisation, used for example in the well-known OMax system [10], uses a data structure called the *factor oracle*, originally introduced for string matching [3, 37]. Given a word w_{ref} of length n representing a reference melody as a sequence of notes, the corresponding factor oracle F is an automaton with the following key properties: F has $n+1$ states, all accepting and arranged in a linear chain. There are *direct* transitions moving along the chain, labeled with the corresponding symbol of w_{ref}, and potentially additional forward and backward transitions. Figure 3.1 shows the factor oracle for the word $w_{\text{ref}} = bbac$.

Note that when we follow the direct transitions, we exactly reproduce the corresponding part of w_{ref}. The extra forward and backward transitions allow us to jump around inside w_{ref}, producing a word which is the concatenation of different subwords (*factors*) of w_{ref}. The more non-direct transitions we use, the more dissimilar the resulting word will be to w_{ref}. Therefore, we can use the factor oracle for improvisation by assigning probabilities to the transitions, giving the bulk of the probability to direct transitions. This turns the automaton into a generative Markov model generating variations on w_{ref} [9]. The first paper on control improvisation by Donzé et al. [44] used this model to satisfy the equivalent of our soft and randomness constraints, heuristically enforcing a hard constraint on top of it (but without providing guarantees of being able to do so).

Here, we can still make use of the factor oracle F by encoding the notion of divergence from w_{ref} that F captures as a DFA. Specifically, we can require that the number of non-

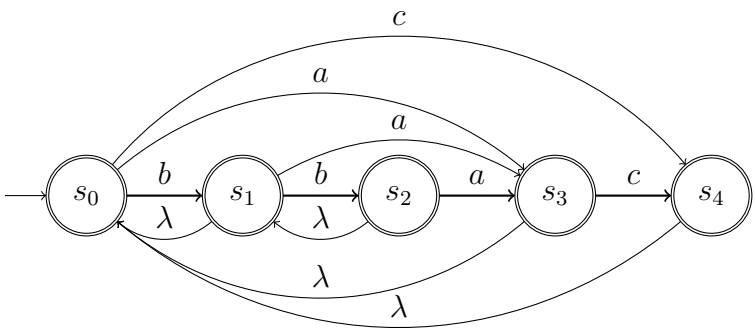


Figure 3.1: Factor oracle corresponding to the word $w_{\text{ref}} = bbac$. The direct transitions are shown in bold.

direct transitions taken by F on any generated word w be large enough to not reproduce w_{ref} too strictly, but small enough so as to get a recognizable variation on it. Since DFAs cannot do unbounded counting, we use a sliding window of some finite size k. Then our soft specification $\mathcal{S}$ can be that at any point as F processes w, the number of the previous k transitions which were non-direct lies in some interval $[\ell, h]$. This predicate can be encoded as a DFA of size $O(|F| \cdot 2^k)$ as follows: we have a copy of F, denoted F_s, for every string $s \in \{0,1\}^k$, each bit of s indicating whether the corresponding previous transition (out of the last k) was non-direct. As each new symbol is processed, we execute the current copy of F as usual, but move to the appropriate state of the copy of F corresponding to the new k-transition history, i.e., if we were in F_s, we move to F_t where t consists of the last $k-1$ bits of s followed by a 0 if the transition we took was direct and a 1 otherwise. Making the states of F_s accepting if and only if the number of 1s in s is in $[\ell, h]$, this automaton represents α as desired. The size of the automaton grows exponentially in the size of the window, but for small windows it can be reasonable.

Finite automata can also capture safety and reachability properties in robotic planning when the state space of the robot and its environment is finite, as on a gridworld:

Example (Patrolling in a Grid World). Consider synthesizing a route for a security robot which patrols a museum. Discretizing the map into a grid, a route can be viewed as a word over the four movement directions the robot can take at each step. Specifications like avoiding collisions with static obstacles and visiting a set of important locations can then easily be encoded as DFAs and enforced by the hard constraint. The randomness constraint then allows us to generate many different routes satisfying these specifications, so that the robot's position is harder to predict ahead of time. To prevent such random routes from being too inefficient, we can use the soft constraint to, for example, require that the robot usually complete its tour within a certain amount of time. We will discuss this example in more detail in Chapter 6.

For deterministic finite automata, there are efficient classical algorithms for counting and

uniform sampling [86], which allow us to use Theorem 3.2 and obtain a polynomial-time improvisation scheme:

Theorem 3.4. *There is a polynomial-time improvisation scheme for* CI(DFA,DFA).

Proof. We instantiate the operations needed by Theorem 3.2.

Intersection: Given two DFAs $\mathcal{X}$ and $\mathcal{Y}$, we can construct a DFA $\mathcal{Z}$ such that $L(\mathcal{Z}) = L(\mathcal{X}) \cap L(\mathcal{Y})$ with the standard product construction. The time needed to do this and the size of $\mathcal{Z}$ are both $O(|\mathcal{X}||\mathcal{Y}|)$.

Difference: Given two DFAs $\mathcal{X}$ and $\mathcal{Y}$, we can construct a DFA $\mathcal{Z}$ such that $L(\mathcal{Z}) = L(\mathcal{X}) \setminus L(\mathcal{Y})$ by complementing $\mathcal{Y}$ with the standard construction, and then taking the product with $\mathcal{X}$. The time required and resulting automaton size are polynomial, as for intersection.

Counting: Using the dynamic programming algorithm of Hickey and Cohen [86] (which does counting as a preliminary step to uniform sampling, along the lines of Wilf [184]), for any $\ell \in \mathbb{N}$ we can compute $|L(\mathcal{X}) \cap \Sigma^\ell|$ in time polynomial in $|\mathcal{X}|$ and ℓ. Summing these for all ℓ from m to n, we obtain $|L(\mathcal{X}) \cap \Sigma^{m:n}|$ as desired[2].

Uniform Sampling: As mentioned above, the algorithm of Hickey and Cohen [86] also allows us to sample uniformly from $|L(\mathcal{X}) \cap \Sigma^\ell|$ for any length $\ell \in \mathbb{N}$ in polynomial time. Since we want to sample uniformly from all words of lengths from m to n, we pick a random length ℓ in this range with probability proportional to the number of words of that length. Specifically, for each ℓ from m to n we compute the count $c_\ell = |L(\mathcal{X}) \cap \Sigma^\ell|$ as above. We then pick a random ℓ so that the probability of obtaining the value j is $c_j / \sum_k c_k$. Sampling from $|L(\mathcal{X}) \cap \Sigma^\ell|$ uniformly at random, the probability of obtaining any $w \in L(\mathcal{X}) \cap \Sigma^{m:n}$ is $(c_{|w|} / \sum_k c_k) \cdot (1/|L(\mathcal{X}) \cap \Sigma^{|w|}|) = 1/\sum_k c_k = 1/|L(\mathcal{X}) \cap \Sigma^{m:n}|$, as desired. Computing the counts and performing the sampling take polynomial time in total.

Since we can perform all of these operations in polynomial time, Theorem 3.2 yields a polynomial-time improvisation scheme. □

This construction depends critically on the fact that the automata involved are *deterministic*, because the algorithm of Hickey and Cohen [86] actually counts and samples accepting *paths* through the automaton rather than accepting *words*. For deterministic automata, every word in the language has exactly one corresponding accepting path, so that the algorithm effectively counts and samples words. However, for *nondeterministic* automata, there

[2] In practice, invocations of the algorithm of Hickey and Cohen [86] to compute the sizes of slices of a language of different lengths should not be done independently. The algorithm builds a memoization table storing the sizes of the ℓ-slice of the language for all ℓ up to the desired length, so we can simply run the algorithm once for length n and query the table for all smaller lengths.

can be multiple accepting paths for a single word in the language, which will cause it to be over-counted. It is easy to construct an NFA with language consisting of two words $\{A, B\}$, with A having a single accepting path and B having exponentially-many accepting paths. On such an automaton, the algorithm of Hickey and Cohen [86] would yield a count with exponentially-large error, and would sample A with only exponentially-small probability. This phenomenon is not simply a flaw with this particular algorithm: counting the language of an NFA is known to be hard, yielding a lower bound on the complexity of CI with such specifications:

Theorem 3.5. CI(NFA,DFA) *and* CI(DFA,NFA) *are #P-hard.*

Proof. We prove this for CI(NFA,DFA) — the other case is similar. As shown by Kannan et al. [93], the problem of determining $|L(\mathcal{M}) \cap \Sigma^\ell|$ given an NFA $\mathcal{M}$ over an alphabet Σ and $\ell \in \mathbb{N}$ in unary is #P-complete. We give a polynomial-time reduction from this problem to checking feasibility of a CI instance in CI(NFA,DFA) (note that as per the definition of #P-completeness, we may use a Cook reduction, i.e., we may call the CI(NFA,DFA) oracle multiple times).

For any NFA $\mathcal{M}$, length $\ell \in \mathbb{N}$, and positive $N \in \mathbb{N}$, consider the CI(NFA,DFA) instance $\mathcal{C}_N = (\mathcal{M}, \mathcal{T}, \ell, \ell, 0, 0, 1/N)$ where $\mathcal{T}$ is the trivial DFA accepting all of Σ^*. Clearly for this instance we have $I = A = L(\mathcal{M}) \cap \Sigma^\ell$. Since $\epsilon = \lambda = 0$ and $\rho = 1/N$, by Theorem 3.1 we have that $\mathcal{C}_N$ is feasible if and only if $|L(\mathcal{M}) \cap \Sigma^\ell| \geq 1/(1/N) = N$. So we can determine whether the latter is the case using a feasibility query for a CI(NFA,DFA) instance. Since $|L(\mathcal{M}) \cap \Sigma^\ell| \leq |\Sigma|^\ell$, using binary search we can find the exact value of $|L(\mathcal{M}) \cap \Sigma^\ell|$ with polynomially-many such queries (recalling that ℓ is given in unary). □

This result indicates that control improvisation problems with NFA specifications probably do not admit a polynomial-time improvisation scheme. However, this does not necessarily mean that such problems are unsolvable in practice: in Section 3.4.3 we will see how to solve them approximately using SAT solvers.

Finally, we note that the original report on control improvisation, Donzé et al. [43], considered using an even more powerful class of automata as specifications, namely *probabilistic automata* (PFAs) [139]. Under the original definition, CI with PFA specifications was actually undecidable [63], due to the undecidability of the problem of checking whether the language of a PFA is empty [126, 38]. However, the latter result depends on the language of the automaton being able to contain arbitrarily-long words, and therefore does not apply to our definition of CI, which includes a length bound. Indeed, since testing whether a word is in the language of a PFA can be done in polynomial time [139], CI problems with PFA specifications can be solved with a #P oracle by Theorem 3.3.

3.4.2 Context-Free Grammars

Another useful class of specifications, more general than finite automata, are context-free grammars. These are widely used to specify file formats, making CI with such grammars

valuable for testing programs that read files:

Example (Software Fuzz Testing). Consider testing an audio player using the *generative* and *mutational* approaches to fuzz testing [165]. A generative fuzz tester might produce random files from the WAVE grammar (shown in part here):

$$S \to \text{`RIFF'}\ \textit{dword}\ \text{`WAVE'}\ \langle\text{format}\rangle\ \left([\langle\text{cue}\rangle]\ [\langle\text{sample}\rangle]\ [\langle\text{playlist}\rangle]\right)^*\ \langle\text{wave-data}\rangle$$
$$\langle\text{wave-data}\rangle \to \langle\text{data}\rangle \mid \langle\text{wave-list}\rangle$$
$$\langle\text{data}\rangle \to \text{`data'}\ \textit{cksize}\ \textit{byte}^*\ \textit{ckend}$$

A mutational fuzz tester would instead produce random variations on a given seed input:

$$(\text{`RIFF'}, 32, \text{`WAVE'}, \text{`fmt '}, 16, 12, 2, 4096, 1024, 7, 3, \text{`data'}, 3178, \dots)$$

Control improvisation enables a *hybrid* approach: generating random variations on a given file that *also* satisfy the grammar. We simply use the grammar as the hard specification, and use a soft specification encoding some appropriate notion of similarity to the seed input.

In general, CFGs are more concise than NFAs, which means that our hardness result for the latter immediately rules out a polynomial-time improvisation scheme:

Theorem 3.6. CI(CFG,DFA) *and* CI(DFA,CFG) *are #P-hard.*

Proof. Follows from Theorem 3.5, since an NFA can be converted to an equivalent CFG in polynomial time [87]. □

Recall that the reason CI with NFAs is hard is that a word in the language of an NFA can have multiple corresponding accepting paths. The same difficulty occurs for CFGs: a word may have multiple derivation trees, making the grammar *ambiguous*. However, since ambiguity slows down parsing (and can lead to confusion about the correct semantics in programming languages, for example), grammars used in practice are often designed to be *unambiguous*. Like deterministic finite automata, such grammars (UCFGs) do admit polynomial-time counting and uniform sampling algorithms [86, 118], which gives hope for a polynomial-time improvisation scheme.

One complication is that UCFGs are not closed under intersection [87], and our generic scheme requires intersecting the hard and soft specifications to construct the set A. In fact, we will see below that when both hard and soft specifications are UCFGs, the CI problem is still #P-hard, as for general grammars. But when only one specification is a UCFG and the other is a DFA, we will obtain a polynomial-time improvisation scheme.

In order to use the construction of Theorem 3.2 in this case, we need to be able to intersect a UCFG G and a DFA D. By a classical result of Bar-Hillel et al. [13], their intersection is a context-free language, and we can compute a CFG H such that $L(H) = L(G) \cap L(D)$ in polynomial time. However, in order to then sample from this intersection, we need H to be unambiguous. Fortunately, as noted by Ginsburg and Ullian [71], the construction of Bar-Hillel et al. [13] actually ensures this when G is itself unambiguous. However, their

presentation does not explicitly demonstrate this fact, so for completeness we present a proof. We also modify the algorithm in several ways to improve its complexity from $\Theta(|G|^2|D|^4)$ to $\Theta(|G||D|^3)$, which makes the algorithm actually feasible for relatively small DFAs.

Lemma 3.1. *Given a UCFG G and a DFA $D = (Q, \Sigma, \delta, q_0, F)$ over a common alphabet Σ, there is an algorithm which computes a UCFG H such that $L(H) = L(G) \cap L(D)$ in $O(|G||D|^3)$ time.*

Proof. First convert G to a CFG $G' = (V, \Sigma, P, S)$ such that

1. $L(G') = L(G) \setminus \{\lambda\}$ (where λ is the empty word), and
2. the RHS of every production in P has length at most 2 and does not contain λ.

This transformation can be done in a way that the time required and the size of G' are both $O(|G|)$, and in the process we determine whether $\lambda \in L(G)$ [87, Section 7.4.2]. Furthermore, it is simple to check that since G is unambiguous, this procedure ensures that G' is also. The rest of our algorithm will build an unambiguous CFG H such that $L(H) = L(G') \cap L(D)$, and thus $\lambda \notin L(H)$. If $\lambda \in L(G) \cap L(D)$ (which we can easily check, since we know whether $\lambda \in L(G)$ from above, and checking if $\lambda \in L(D)$ is trivial), we can simply add the production $S \to \lambda$ to H without introducing any ambiguity. Therefore ultimately we will have $L(H) = L(G) \cap L(D)$ as desired.

The main construction of Bar-Hillel et al. [13] works on DFAs with a single accepting state: there is an initial preprocessing step which writes D as a union of $|F|$ DFAs of that form, so that the construction can be carried out on each and the resulting grammars combined. Doing this would contribute a factor of $|F|$ to our algorithm's runtime, so instead we modify D to produce an NFA $D' = (Q', \Sigma, \delta', q_0, F')$ as follows. We add two new states Accept and Reject, where $\delta(\textsc{Accept}, a) = \delta(\textsc{Reject}, a) = \textsc{Reject}$ for every $a \in \Sigma$. Also for any transition $\delta(x, a) = y$ where $y \in F$, we add a transition from x to Accept on input a — note that this makes D' nondeterministic. Finally, we make Accept the only accepting state of D'. Clearly we can construct D' in time linear in $|D|$.

Now consider any nonempty word $w \in L(D)$, which we may write $a_0 \dots a_n$ for some $n \geq 0$. Let w' be w without its last symbol a_n (so $w' = \epsilon$ if w has length 1). Since $w \in L(D)$ there is some $x \in Q$ and $y \in F$ such that $\delta(q_0, w') = x$ and $\delta(x, a_n) = y$. Therefore $w \in L(D')$, since D' on input w' can follow unmodified transitions from D to reach x, and then the new transition from x to Accept on input a_n. Conversely, if $w \in L(D')$ then executing D' on input w we must end in state Accept, and all transitions except the last must be transitions of D (since once we follow a new transition to Accept, any further transitions will end in Reject). Note that this means there is only one accepting path in D' corresponding to w, so D' is unambiguous — this will be important later. If the last transition in this path is from state x on input a, then by construction $\delta(x, a) \in F$ and so $w \in L(D)$. Therefore $L(D') = L(D) \setminus \{\lambda\}$, and since $\lambda \notin L(G')$ we have $L(G') \cap L(D) = L(G') \cap L(D')$.

Now, following Bar-Hillel et al. [13], we build the CFG $H = (\hat{V}, \Sigma, \hat{P}, \hat{S})$ where $\hat{V} = (Q' \times V \times Q') \cup \Sigma$ and $\hat{S} = (q_0, S, \text{ACCEPT})$. There are two[3] kinds of productions in $\hat{P}$:

1. For every production $A \to BC$ in P, we add the productions $(x, A, z) \to (x, B, y)(y, C, z)$ for all $x, y, z \in Q'$ to $\hat{P}$.

2. For every production $A \to b$ in P, we add the productions $(x, A, y) \to b$ for all $x, y \in Q'$ such that $(x, b, y) \in \delta'$ to $\hat{P}$.

For a proof that $L(H) = L(G') \cap L(D')$ (and thus that $L(H) = L(G') \cap L(D)$, as desired), see Bar-Hillel et al. [13] (note however that our type 2 productions are split into two separate productions in their presentation). Clearly, we can construct H in $O(|P| \cdot |Q'|^3) = O(|G||D|^3)$ time.

It remains to show that H is unambiguous. Take any word $w \in L(H)$, and any two derivation trees T_1 and T_2 for w. Observe that by the way we constructed the productions of H above, if we drop the first and third components of the labels on the non-leaf nodes of T_1 or T_2 we obtain a derivation tree for w from the grammar G'. Therefore since G' is unambiguous, T_1 and T_2 have the same tree structure.

Now we prove by induction on trees that every node in T_1 has the property that its label is identical to the corresponding node in T_2. For the leaf nodes this is immediate, since they spell out w in either tree. For nodes one level up, i.e. those to which a production of type 2 is applied, it is proved by Bar-Hillel et al. [13] that if these are written from left to right they form a sequence $(q_0, v_0, q_1), (q_1, v_1, q_2), \ldots, (q_{n-1}, v_{n-1}, q_n)$ where $q_0 q_1 \ldots q_n$ is an accepting path for w in D'. Since as shown above there is only one such path, these nodes also have the property. Finally, observe that under productions of type 1, the state labels of the parent node are uniquely determined by those of its children. So if all of a node's children satisfy the property, so does the node itself, and thus by induction all nodes have the property. Therefore T_1 and T_2 are identical, and so H is unambiguous. □

Using this algorithm, we can build a polynomial-time improvisation scheme for the case where the hard specification is a UCFG:

Theorem 3.7. *There is a polynomial-time improvisation scheme for* CI(UCFG,DFA).

Proof. We follow the procedure of Theorem 3.2. To take the intersection of $\mathcal{H}$ and $\mathcal{S}$ we apply Lemma 3.1, obtaining a UCFG $\mathcal{A}$ such that $L(\mathcal{A}) \cap \Sigma^{m:n} = A$. To take their difference, we apply Lemma 3.1 to $\mathcal{H}$ and the complement of $\mathcal{S}$ (computed with the usual DFA construction), obtaining a UCFG $\mathcal{B}$ such that $L(\mathcal{B}) \cap \Sigma^{m:n} = I \setminus A$. Next, applying the

[3] In fact there are four additional kinds, where the RHS is of the form bc, bC, Bc, or B. But these are easily handled along the same lines as types (1) and (2) above, so for simplicity we omit the details. The reason for dealing with all of these different types of productions instead of converting G to Chomsky normal form (which only has productions of types (1) and (2)) is that the conversion to CNF can square the size of the grammar. The conversion we use only expands the grammar by a linear amount at most, while keeping the construction of H efficient.

algorithms of Hickey and Cohen [86] or McKenzie [118] to $\mathcal{H}$ and $\mathcal{A}$, we compute $|I|$ and $|A|$ (adding up the counts for all lengths between m and n as in Theorem 3.4). Then we can proceed exactly as in Theorem 3.2, using the algorithms of Hickey and Cohen [86] or McKenzie [118] to sample from $L(\mathcal{A}) \cap \Sigma^{m:n} = A$ and $L(\mathcal{B}) \cap \Sigma^{m:n} = I \setminus A$. □

The opposite case, where the hard specification is a DFA and the soft specification is a UCFG, is significantly more involved. In the procedure above, we complemented the soft specification in order to construct the set $I \setminus A$. We cannot do this here because the complement of a context-free language is not necessarily context-free [87], and so we will not be able to apply the sampling algorithm for UCFGs. Fortunately, there is a workaround: note that we can easily *count* the complement of a UCFG G, since $|\overline{L(G)} \cap \Sigma^{m:n}| = |\Sigma^{m:n}| - |L(G) \cap \Sigma^{m:n}|$ and so counting the complement reduces to counting the original grammar. Then we can sample the complement by applying the general random-walk reduction of uniform sampling to counting [184].

Theorem 3.8. *There is a polynomial-time improvisation scheme for* CI(DFA,UCFG).

Proof. We again follow the procedure of Theorem 3.2. Applying Lemma 3.1 to $\mathcal{H}$ and $\mathcal{S}$, we obtain a UCFG $\mathcal{A}$ such that $L(\mathcal{A}) \cap \Sigma^{m:n} = A$. Then we can apply the algorithm of Hickey and Cohen [86] to $\mathcal{H}$ and $\mathcal{A}$ to compute $|I|$ and $|A|$. If we can uniformly sample from $\Delta = L(\mathcal{H}) \cap \overline{L(\mathcal{A})} \cap \Sigma^{m:n} = L(\mathcal{H}) \cap \Sigma^{m:n} \setminus L(\mathcal{A}) = I \setminus A$, we can then proceed exactly as in Theorem 3.2.

We will build up a word from Δ incrementally, starting from the empty word. Suppose we have generated the prefix σ so far. For each symbol $a \in \Sigma$, let $\Delta_{\sigma a} \subseteq \Delta$ contain all words starting with the prefix σa, i.e. $\Delta_{\sigma a} = \{w \in \Delta \mid \exists z \in \Sigma^* : w = \sigma a z\}$. Construct a DFA $P_{\sigma a}$ accepting all words in $\Sigma^{m:n}$ that start with the prefix σa (clearly we can take $P_{\sigma a}$ to have size polynomial in $|\Sigma|$ and n). Then $\Delta_{\sigma a} = \Delta \cap L(P_{\sigma a}) = \overline{\overline{\Delta} \cup \overline{L(P_{\sigma a})}} = \overline{\overline{L(\mathcal{H})} \cup L(\mathcal{A}) \cup \overline{L(P_{\sigma a})}} = \Sigma^{m:n} \setminus (\overline{L(\mathcal{H})} \cup L(\mathcal{A}) \cup \overline{L(P_{\sigma a})})$. Letting D be the complement of the product of $\mathcal{H}$ and $P_{\sigma a}$, we have $L(D) = \overline{L(\mathcal{H})} \cup \overline{L(P_{\sigma a})}$ and so $\Delta_{\sigma a} = \Sigma^{m:n} \setminus (L(\mathcal{A}) \cup L(D))$. Applying Lemma 3.1 to $\mathcal{A}$ and $P_{\sigma a}$, we can find a UCFG $\mathcal{A}_{\sigma a}$ such that $L(\mathcal{A}_{\sigma a}) = L(\mathcal{A}) \cap L(P_{\sigma a})$. Then we have

$$\begin{aligned}
|\Delta_{\sigma a}| &= |\Sigma^{m:n} \setminus (L(\mathcal{A}) \cup L(D))| \\
&= |\Sigma^{m:n}| - |(L(\mathcal{A}) \cup L(D)) \cap \Sigma^{m:n}| \\
&= |\Sigma^{m:n}| - \left|\left[(L(\mathcal{A}) \cap \overline{L(D)}) \cup L(D)\right] \cap \Sigma^{m:n}\right| \\
&= |\Sigma^{m:n}| - |L(\mathcal{A}) \cap \overline{L(D)} \cap \Sigma^{m:n}| - |L(D) \cap \Sigma^{m:n}| \\
&= |\Sigma^{m:n}| - |L(\mathcal{A}) \cap L(\mathcal{H}) \cap L(P_{\sigma a}) \cap \Sigma^{m:n}| - |L(D) \cap \Sigma^{m:n}| \\
&= |\Sigma^{m:n}| - |L(\mathcal{A}_{\sigma a}) \cap \Sigma^{m:n}| - |L(D) \cap \Sigma^{m:n}|.
\end{aligned}$$

$L(\mathcal{A}_{\sigma a})$ and D have polynomial size so using the algorithm of Hickey and Cohen [86] we can compute the last two terms of the formula above in polynomial time. Therefore we can compute $|\Delta_{\sigma a}|$ in polynomial time.

Now let Δ_σ consist of all words in Δ that start with the prefix σ. Observe that the sets $\Delta_{\sigma a}$ form a partition Π_σ of Δ_σ, unless $\sigma \in \Delta$ in which case we need to also add the set $\{\sigma\}$ to Π_σ. Select one of the sets in Π_σ randomly, with probability proportional to its size. If we pick $\{\sigma\}$, then we stop and return σ as our random sample. Otherwise we picked $\Delta_{\sigma a}$ for some $a \in \Sigma$, so we append a to σ and continue.

Since the procedure of Theorem 3.2 only samples from $I \setminus A = \Delta$ when it is nonempty, in the first iteration $\Delta_\sigma = \Delta$ is nonempty and so one of the sets in Π_σ must be nonempty. Since we assign a set in Π_σ probability zero if it is empty, we will not select such a set, and so by induction Δ_σ is nonempty at every iteration. We must terminate after at most n iterations, since σ gets longer in every iteration and all words in Δ have length at most n. We prove by induction on $n - |\sigma|$ that if Δ_σ is nonempty, our procedure starting from σ generates a uniform random sample from Δ_σ. In the base case $|\sigma| = n$, we have $\Delta_\sigma = \{\sigma\}$, since no word in Δ has length greater than n but Δ_σ is nonempty. Therefore $\Delta_{\sigma a} = \emptyset$ for all $a \in \Sigma$, so the procedure will return σ, and this is indeed a uniform random sample from Δ_σ. For $|\sigma| < n$, if $\sigma \in \Delta_\sigma$ then the probability of returning σ is the probability of picking the set $\{\sigma\}$ from Π_σ, which is $1/\sum_{S \in \Pi_\sigma} |S| = 1/|\Delta_\sigma|$. Any other $w \in \Delta_\sigma$ has length at least $|\sigma| + 1$ and so can be written $w = \sigma a w'$ for some $a \in \Sigma$ and $w' \in \Sigma^*$. Now to return w, our procedure must first pick the set $\Delta_{\sigma a}$ from Π_σ, and then return w in a later iteration. By the induction hypothesis, this happens with probability $(|\Delta_{\sigma a}|/\sum_{S \in \Pi_\sigma} |S|) \cdot (1/|\Delta_{\sigma a}|) = 1/|\Delta_\sigma|$. So all words in Δ_σ are returned with uniform probability, and by induction this holds for all σ. In particular it holds in the first iteration when σ is the empty word, in which case we sample uniformly from $\Delta_\sigma = \Delta$ as desired.

This allows us to uniformly sample from $I \setminus A$ and so finish the implementation of the procedure in Theorem 3.2. All of the operations we perform are polynomial-time, and the sampling procedure needs only polynomially-many iterations, so this yields a polynomial-time improvisation scheme. □

Together, Theorems 3.7 and 3.8 show that we can efficiently solve CI problems where one specification is a UCFG and the other is a DFA. However, when *both* the hard and soft specifications are unambiguous grammars, the complexity jumps all the way up to #P, the complexity class for general grammars. In order to show this, we introduce an intermediate problem that captures the essentially hard part of solving such CI problems:

Definition 3.7. #UCFG-Int is the problem of computing, given two UCFGs G_1 and G_2 over an alphabet Σ and $n \in \mathbb{N}$ in unary, the number of words in $L(G_1) \cap L(G_2) \cap \Sigma^{\leq n}$.

Without the length bound, checking emptiness of the intersection of two CFGs is undecidable, as shown by Bar-Hillel et al. [13] using a reduction to the Post correspondence problem [138]. We use a similar proof (in its details closer to that of Floyd [58]) to establish the complexity of the bounded version.

Lemma 3.2. #UCFG-Int *is #P-complete.*

Proof. All the words in $L(G_1) \cap L(G_2) \cap \Sigma^n$ have length polynomial in the size of the input, and checking whether a word is in the set can be done in polynomial time. So #UCFG-INT $\in$ #P.

To show hardness, we give a reduction to #UCFG-INT from #BOUNDED-PCP, the counting version of the bounded Post correspondence problem. Recall that BOUNDED-PCP instances are ordinary PCP instances together with a bound $m \in \mathbb{N}$ in unary on the total number of tiles that may be used in a solution. In the usual construction simulating a Turing machine M with PCP tiles (see for example Hopcroft et al. [87]), an accepting computation history for M leads to a PCP solution using a number of tiles linear in the size of the history. So the problem of checking whether M accepts a word in at most a number of steps given in unary reduces to BOUNDED-PCP, and thus the latter is NP-complete (as noted in a slightly different form by Constable et al. [39]). Furthermore, this reduction is almost parsimonious: the computation history uniquely determines the PCP solution until the accepting state is reached, when (at least in the reduction given in Hopcroft et al. [87]) the accepting state can "consume" tape symbols on either side of it in any order. However, we can easily modify the construction so that the symbols are consumed in a canonical order. In the terminology of Hopcroft et al. [87], we replace the "type-4" tiles with tiles of the form (qX, q) and $(Xq\#, q\#)$ for all accepting states q and tape symbols X. This forces tape symbols to be consumed from the right of q first, and only allows consumption from the left once there are no tape symbols to the right (so that q is adjacent to the # marking the end of the element of the history). Now there is a one-to-one correspondence between accepting computation histories of M and solutions to the BOUNDED-PCP instance, so the counting problem #BOUNDED-PCP is #P-complete.

Given an instance $\mathcal{P}$ of #BOUNDED-PCP with tiles $(x_1, y_1), \ldots, (x_k, y_k)$ over an alphabet Σ' and a bound m, we construct grammars X and Y along the lines of Bar-Hillel et al. [13] and Floyd [58]. The grammars use an alphabet Σ consisting of Σ' together with additional symbols $t_1, \ldots, t_k$, and have nonterminal symbols X_i and Y_i respectively for $1 \le i \le m$. Their start symbols are X_m and Y_m respectively, and they have the following productions:

$$\begin{aligned}
X_i &\to x_1 X_{i-1} t_1 \mid \cdots \mid x_k X_{i-1} t_k \mid X_1 && 2 \le i \le m \\
X_1 &\to x_1 t_1 \mid \cdots \mid x_k t_k \\
Y_i &\to y_1 Y_{i-1} t_1 \mid \cdots \mid y_k Y_{i-1} t_k \mid Y_1 && 2 \le i \le m \\
Y_1 &\to y_1 t_1 \mid \cdots \mid y_k t_k
\end{aligned}$$

An easy induction shows that the words derivable from X_i (respectively, Y_i) correspond to nonempty sequences of at most i tiles. Thus there is a one-to-one correspondence between words in $L(X) \cap L(Y)$ and solutions of $\mathcal{P}$. Furthermore, both grammars are unambiguous, since the sequence of t_i symbols uniquely determines the derivation tree. Finally, all words in $L(X)$ have length at most $n = m(1+\max(|x_1|, \ldots, |x_k|))$, so $|L(X) \cap L(Y)| = |L(X) \cap L(Y) \cap \Sigma^{\le n}|$. Therefore $(X, Y, 1^n)$ is a #UCFG-INT instance with the same number of solutions as $\mathcal{P}$. This reduction clearly can be done in polynomial time, so #UCFG-INT is #P-hard. □

Now we may establish the hardness of CI(UCFG,UCFG).

Theorem 3.9. CI(UCFG,UCFG) *is #P-hard.*

Proof. We reduce #UCFG-Int to checking feasibility of a CI(UCFG,UCFG) instance, along the same lines as Theorem 3.5. Given a #UCFG-Int instance $(\mathcal{H}, \mathcal{S}, 1^n)$ and fixing some $N \in \mathbb{N}$, consider the CI instance $\mathcal{C}_N = (\mathcal{H}, \mathcal{S}, 0, n, 0, 0, 1/N)$. For this instance we have $I = L(\mathcal{H}) \cap \Sigma^{\leq n}$ and $A = L(\mathcal{H}) \cap L(\mathcal{S}) \cap \Sigma^{\leq n}$. Since $\epsilon = \lambda = 0$ and $\rho = 1/N$, by Theorem 3.1 we have that $\mathcal{C}_N$ is feasible if and only if $|A| \geq N$, so we can determine if $|L(\mathcal{H}) \cap L(\mathcal{S}) \cap \Sigma^{\leq n}| \geq N$ with a feasibility query. Since $|L(\mathcal{H}) \cap L(\mathcal{S}) \cap \Sigma^{\leq n}| = O(|\Sigma|^n)$, we can compute $|L(\mathcal{H}) \cap L(\mathcal{S}) \cap \Sigma^{\leq n}|$ with polynomially-many such queries by binary search. This gives a polynomial-time (Cook) reduction from #UCFG-Int to CI(UCFG,UCFG), so by Lemma 3.2 the latter is #P-hard. □

3.4.3 Boolean Formulas

In this section we discuss an important class of specifications given by existentially-quantified Boolean formulas. We begin by defining the class and outline how it includes a widely-used succinct encoding of large finite-state automata. Then we show that although the CI problem is #P-hard with these specifications, it can be solved approximately using only an NP oracle. This means that some CI problems with very large automata can still be solved in practice using SAT solvers.

The specifications we will consider are Boolean formulas with auxiliary variables, where a word w (encoded in binary) is in the language if the formula is satisfiable after plugging in w.

Definition 3.8. Fixing an alphabet $\Sigma = \{0,1\}^k$ and a length $n \in \mathbb{N}$, a *symbolic specification* is a Boolean formula $\phi(\overline{w}, \overline{a})$ where $\overline{w}$ is a vector of nk variables and $\overline{a}$ is a vector of zero or more variables. The language of ϕ consists of all $\overline{w} \in \Sigma^n$ such that $\exists \overline{a}\, \phi(\overline{w}, \overline{a})$ is true. The class of symbolic specifications is denoted Symb.

Remark. The restriction that all words in the language of any symbolic specification have a fixed length n is for convenience: this allows us to intersect two specifications simply by conjoining their formulas. We can always pad shorter words out to length n with a dummy symbol added to the alphabet, altering the formula ϕ as appropriate.

Symbolic specifications arise in the analysis of transition systems, and in practice can be useful even for DFAs in situations where there is insufficient memory to store full transition tables. This is not uncommon in practice, because specifications are often built up as the conjunction of many small pieces. To use our earlier improvisation scheme for DFAs (Theorem 3.4), we would need to explicitly construct the product of these automata, which could be exponentially large. An implicit representation of the product by Boolean formulas, in contrast, will have linear size.

In bounded model checking [16], DFAs and NFAs can be represented by formulas as follows:

Definition 3.9. A *symbolic automaton* is a transition system over states $S \subseteq \{0,1\}^m$ and input alphabet $\Sigma \subseteq \{0,1\}^k$ represented by:

- a formula $\text{init}(\overline{x})$ which is true if and only if $\overline{x} \in \{0,1\}^m$ is an initial state,
- a formula $\text{acc}(\overline{x})$ which is true if and only if $\overline{x} \in \{0,1\}^m$ is an accepting state, and
- a formula $\delta(\overline{x}, \overline{c}, \overline{y})$ which is true if and only if there is a transition from $\overline{x} \in \{0,1\}^m$ to $\overline{y} \in \{0,1\}^m$ on input $\overline{c} \in \{0,1\}^k$.

A symbolic automaton accepts input words according to the usual definition for NFAs: it accepts $w \in \Sigma^*$ if and only if there is a path corresponding to w which leads from an initial state to an accepting state.

Given a symbolic automaton $\mathcal{D}$ and a length bound $n \in \mathbb{N}$, it is straightforward to generate a symbolic specification ϕ such that $L(\phi) = L(\mathcal{D}) \cap \Sigma^n$. The auxiliary variables of ϕ encode accepting paths of length n for a given word (see Biere et al. [16] for details), and existentially quantifying them yields a formula whose models correspond exactly to accepting words of length n. So for the rest of the section we focus on symbolic specifications, but our results apply to symbolic automata in particular[4].

Since symbolic specifications can be arbitrary Boolean formulas, for which counting is #P-complete, checking feasibility of CI problems involving them is obviously #P-hard. In the other direction, deciding membership of a word in a symbolic specification can be done with an NP oracle, so by Theorem 3.3 there is a polynomial-time improvisation scheme relative to a #P oracle (indeed, this holds even if we extend the definition of symbolic specification to allow a larger but bounded number of quantifiers). However, this scheme is not much use in practice, since #P-complete problems are very difficult to solve exactly.

On the other hand, as we saw in Chapter 2, it is possible to *approximately* solve #P problems using only an NP oracle, or in practice, a SAT solver. This does not let immediately let us solve CI problems with symbolic specifications: since feasibility checking is #P-hard, by Toda's theorem [171] we cannot solve such problems using only an NP oracle unless PH collapses. But algorithms for approximate model counting and sampling will allow us to solve symbolic CI problems in an *approximate* sense, achieving values of ϵ, λ, and ρ which are somewhat weaker than the optimal values. We will call an approximate improviser achieving given values of these parameters an $(\epsilon, \lambda, \rho)$-*improviser*.

Before we describe our approximate improvisation scheme, the next lemma makes precise how we can use uniform generation algorithms for SAT to sample the languages of symbolic specifications:

Lemma 3.3. *There is a probabilistic algorithm using an NP oracle that given a symbolic specification ϕ and any $\tau > 0$, returns a random sample from $L(\phi)$ that is uniform up to a*

[4] In fact, our results also apply to a more general notion of symbolic automaton where in Definition 3.9 we allow all three of the formulas to use existentially-quantified auxiliary variables. The BMC-style encoding of accepting paths works without change, the resulting formula simply having additional auxiliary variables.

multiplicative factor of $1+\tau$*. The algorithm runs in expected time polynomial in* $|\phi|$ *and* $1/\tau$ *relative to the oracle.*

Proof. By our definition above, words in the language of ϕ are assignments to the $\overline{w}$ variables of $\phi(\overline{w},\overline{a})$ that can be extended to a complete satisfying assignment. For an arbitrary formula ϕ, counting the number of such words is exactly the *projected model counting* problem described in Chapter 2. Sampling from $L(\phi)$ is therefore equivalent to (approximate) uniform sampling of the models of $\phi(\overline{w},\overline{a})$ projected onto the variables $\overline{w}$. This can be done with the algorithm of Chakraborty et al. [29], which will run in the time required relative to an NP oracle. Unfortunately for technical reasons the algorithm works only for $\tau > 6.84$. For general τ, we can instead modify the algorithm of Bellare et al. [15], which does sampling without projection. To add projection, we modify the algorithm as described in Chakraborty et al. [29]: apply the hash function only to the $\overline{w}$ variables, and when enumerating solutions using the NP oracle block all solutions that agree on the $\overline{w}$ with the solutions already enumerated. One minor complication is that this algorithm has a constant probability of failing by returning $\perp$ instead of a sample. If that happens, we simply retry until the algorithm succeeds: this only increases the expected runtime by a constant factor. We note that the algorithm of Bellare et al. [15] actually samples *exactly* uniformly at random, and so would allow us to get somewhat better performance in theory, but with current SAT solvers the algorithm does not scale (unlike the algorithm of Chakraborty et al. [29]). □

A natural approach to building an approximate improvisation scheme for symbolic CI problems is to use the generic approach of Theorem 3.2, simply using approximate counting and sampling in place of the exact operations. However, there are two problems. First, symbolic specifications are not closed under the difference operation, since negating a formula turns existential quantifiers into universal quantifiers. Second, with approximate counting the distribution constructed by Theorem 3.2 can be arbitrarily far from an improvising distribution. Consider a CI instance where $|A| = M$, $|I| = M+1$, $\epsilon = \rho = 1/(M+1)$, and $\lambda = 0$. This is clearly feasible, with the uniform distribution on I being an improvising distribution. Now suppose our estimated count for A is too small by a factor of $1+\tau$ for some $\tau > 0$, so that our algorithm thinks it has $M/(1+\tau)$ elements. Then our scheme will compute

$$\epsilon_{\text{opt}} = 1 - \rho|A| = 1 - \frac{1}{M+1} \cdot \frac{M}{1+\tau} = \frac{M\left(\frac{\tau}{1+\tau}\right)+1}{M+1}.$$

Now since $I \setminus A$ has only one element, we will generate that element with probability ϵ_{opt}, and the ratio $\epsilon_{\text{opt}}/\rho = M(\tau/(1+\tau))+1$ is arbitrarily large as $M \to \infty$. So for any constant-factor approximation of the size of A, our naïve algorithm could yield a distribution which is arbitrarily far from the maximum allowed probability ρ.

To avoid these problems, our approximate improvisation scheme avoids complementing $\mathcal{S}$, and in fact does not require counting at all. In exchange, for a desired ϵ the algorithm may return an improviser which does not achieve the best possible λ and ρ, but their suboptimality is bounded.

Theorem 3.10. *There is a procedure that given any $\tau > 0$ and a feasible CI problem $\mathcal{C} \in$ CI(SYMB,SYMB) with $\mathcal{C} = (\mathcal{H}, \mathcal{S}, n, n, \epsilon, \lambda, \rho)$, returns an $(\epsilon, \epsilon\lambda/(1+\tau), \rho(1+\epsilon)(1+\tau))$-improviser. Furthermore, the procedure and the improvisers it generates run in expected time given by some fixed polynomial in $|\mathcal{C}|$ and $1/\tau$ relative to an NP oracle.*

Proof. Conjoin $\mathcal{H}$ and $\mathcal{S}$ (renaming any shared auxiliary variables) to produce a symbolic specification $\mathcal{A}$ such that $L(\mathcal{A}) = L(\mathcal{H}) \cap L(\mathcal{S}) = A$. We build a probabilistic algorithm G^{NP} that samples approximately uniformly from A with probability $1-\epsilon$ and from I with probability ϵ. By Lemma 3.3, we can do this with tolerance τ in polynomial expected time relative to the NP oracle.

By definition, G always returns an element of I, and returns an element of A with probability at least $1-\epsilon$, so the hard and soft constraints are satisfied. Since $\mathcal{C}$ is feasible, by Theorem 3.1 we have $1/\rho \leq |I| \leq 1/\lambda$ and $(1-\epsilon)/\rho \leq |A|$. Now for any $w \in A$, we have $\Pr[w \leftarrow G] \geq (1-\epsilon)(1/|A|)/(1+\tau) + \epsilon(1/|I|)/(1+\tau) \geq \lambda/(1+\tau)$, while for any $w \in I \setminus A$ we have $\Pr[w \leftarrow G] \geq \epsilon(1/|I|)/(1+\tau) \geq \epsilon\lambda/(1+\tau)$. Similarly, for any $w \in A$ we have $\Pr[w \leftarrow G] \leq (1-\epsilon)(1/|A|)(1+\tau) + \epsilon(1/|I|)(1+\tau) \leq \rho(1+\epsilon)(1+\tau)$, while for any $w \in I \setminus A$ we have $\Pr[w \leftarrow G] \leq \epsilon(1/|I|)(1+\tau) \leq \epsilon\rho(1+\tau)$. So G is an $(\epsilon, \epsilon\lambda/(1+\tau), \rho(1+\epsilon)(1+\tau))$-improviser. □

Therefore, it is possible to *approximately* solve CI problems with specifications represented succinctly by Boolean formulas. This allows working with general NFAs, as well as deterministic automata that are too large to be stored explicitly (e.g. those arising as a product of many small automata), at the cost of using a SAT solver and having to relax the randomness requirement somewhat.

3.5 CI with Multiple Constraints

In this section, we discuss a generalized problem, *multi-constraint control improvisation (MCI)* where multiple soft constraints are allowed. We introduced an earlier form of this extension in Akkaya et al. [2] (for the lighting control application we will discuss in Chapter 7), with a complete definition and partial analysis following in Fremont et al. [62]. Compared to the basic CI problem, the conditions under which an instance of MCI is feasible are far more complex, and we do not have a concise form for them. As a result, we do not have a simple construction of an improviser, and are forced to fall back to the linear programming formulation of CI mentioned earlier, which at least allows us to give an exponential-time improvisation scheme. Finally, although we cannot pin down the complexity of MCI exactly, we show that it is significantly harder than basic CI: even with DFA specifications, checking MCI feasibility is #P-hard.

3.5.1 Introduction

In various applications of control improvisation, the natural formulation of the problem uses multiple constraints. We often have multiple requirements for a system to satisfy: for example, in robotic planning we might want to ensure the robot avoids collisions, visits several locations of interest, and returns to its home base. Hard constraints such as these can be combined into a single specification: requiring that several specifications $\mathcal{H}_1, \ldots, \mathcal{H}_k$ should all hold with probability 1 is equivalent to requiring that a single product specification $\mathcal{H}$ hold with probability 1. As noted above, this can cause a performance blowup if the product specification can be much larger than its components, but this is a problem of complexity and not expressivity: the definition of control improvisation we have used so far effectively permits multiple hard constraints.

However, this is not true for soft constraints. In general requiring that both $\Pr[w \in A_1] \geq 1/2$ and $\Pr[w \in A_2] \geq 1/2$ is not equivalent to $\Pr[w \in A] \geq p$ for *any* single property A and probability p. This is an issue, because there are natural problems requiring multiple soft constraints, such as the human-like lighting control application of Akkaya et al. [2].

Example (Human-Like Lighting Control). Suppose we want to automatically turn the lights in various rooms of a house on and off in a way that makes it look like the owner is at home. This task is naturally modeled as a CI problem, since we want a random policy for the lights (fixed timings repeated every day would be unnatural and suspicious), but could also want to put constraints on power consumption. In particular, we might want to be as human-like as possible, while still decreasing the power consumption during peak hours (i.e. the most expensive hours) by some desired amount. If there are multiple time periods during the day when we want to impose such limits, the most natural encoding would be to use one soft constraint for each period. We will discuss this application more formally in Chapter 7.

To handle applications like the one above, we remove the asymmetry in the CI definition by allowing multiple soft constraints. Specifically, the soft specification $\mathcal{S}$ and corresponding error probability ϵ are replaced by several specifications $\mathcal{S}_1, \ldots, \mathcal{S}_k$ and error probabilities $\epsilon_1, \ldots, \epsilon_k$.

Definition 3.10. Fix a *hard specification* $\mathcal{H}$, *soft specifications* $\mathcal{S}_1, \ldots, \mathcal{S}_k$, and length bounds $m, n \in \mathbb{N}$. An *improvisation* is any word $w \in L(\mathcal{H}) \cap \Sigma^{m:n}$, and we write I for the set of all improvisations as before. An improvisation $w \in I$ is *i-admissible* if $w \in L(\mathcal{S}_i)$, and we write A_i for the set of all i-admissible improvisations.

Definition 3.11. Given $\mathcal{C} = (\mathcal{H}, \mathcal{S}_1, \ldots, \mathcal{S}_k, m, n, \epsilon_1, \ldots, \epsilon_k, \lambda, \rho)$ with $\mathcal{H}$, $\mathcal{S}_i$, m, and n as above, $\epsilon_i \in [0,1] \cap \mathbb{Q}$ error probabilities, and $\lambda, \rho \in [0,1] \cap \mathbb{Q}$ probability bounds, a distribution $D : \Sigma^* \to [0,1]$ is an *improvising distribution* if it satisfies the following requirements:

Hard constraint: $\Pr[w \in I \mid w \leftarrow D] = 1$

Soft constraints: $\forall i \in [k],\ \Pr[w \in A_i \mid w \leftarrow D] \geq 1 - \epsilon_i$

Randomness: $\forall w \in I,\ \lambda \leq D(w) \leq \rho$

Feasibility, improvisers, and improvisation schemes are defined in terms of improvising distributions exactly as in Definitions 3.3 and 3.6.

Definition 3.12. Given $\mathcal{C} = (\mathcal{H}, \mathcal{S}_1, \ldots, \mathcal{S}_k, m, n, \epsilon_1, \ldots, \epsilon_k, \lambda, \rho)$, the *multi-constraint control improvisation (MCI)* problem is to decide whether $\mathcal{C}$ is feasible, and if so to generate an improviser for $\mathcal{C}$. The *size* $|\mathcal{C}|$ of an MCI instance is measured as for CI instances.

Definition 3.13. If $\mathcal{A}$ and $\mathcal{B}$ are classes of specifications, MCI($\mathcal{A}$,$\mathcal{B}$) is the class of MCI instances $\mathcal{C} = (\mathcal{H}, \mathcal{S}_1, \ldots, \mathcal{S}_k, m, n, \epsilon_1, \ldots, \epsilon_k, \lambda, \rho)$ where $\mathcal{H} \in \mathcal{A}$ and $\mathcal{S}_i \in \mathcal{B}$ for all $i \in [k]$. We write MCI$_k$($\mathcal{A}$,$\mathcal{B}$) for the subset of MCI($\mathcal{A}$,$\mathcal{B}$) with the given value of k. When discussing decision problems, we use the same notation for the feasibility problem associated with the class (i.e. given $\mathcal{C} \in$ MCI($\mathcal{A}$,$\mathcal{B}$), decide whether it is feasible).

Note that while the MCI definition treats the multiple soft constraints conjunctively (i.e. *every* constraint must hold), if the type of specification used supports Boolean operations then other types of soft constraint can be brought to this form. For example, the requirement

$$\Pr[w \in A_1] < 3/4 \implies \Pr[w \in A_2 \wedge w \in A_3] \geq 1/5$$

can be rewritten

$$\Pr[w \in \overline{A_1}] \geq 1/4 \vee \Pr[w \in (A_2 \cap A_3)] \geq 1/5,$$

and then each disjunct tested for feasibility separately. In general, we can reduce Boolean combinations of specifications inside probabilities to single specifications and write the resulting constraint in disjunctive normal form. Each disjunct is then an MCI instance (ignoring the very minor issue of strict vs. non-strict inequalities), and the original problem with a complex soft constraint is feasible if and only if one of the disjuncts is. This transformation could of course blow up the size of the problem exponentially — the point is that slightly more complex soft constraints can be handled within the MCI framework.

3.5.2 Feasibility and the Linear Programming Formulation

Unlike the case of basic CI, the feasibility conditions for MCI do not fall out of a simple intuition for how to build an improvising distribution. Whereas for CI we only need bounds on the sizes of I and A (Theorem 3.1), for MCI feasibility requires bounds not just on the sizes of the sets A_i individually but also on the sizes of their intersections, pairwise, 3-wise, and so forth. For example, if $\mathcal{S}_1$ and $\mathcal{S}_2$ must each be satisfied with probability at least 3/4, then A_1 and A_2 cannot be disjoint, since we only have a total probability of 1 to distribute between them. Along these lines it is straightforward to derive various inequalities which are *necessary* for an MCI instance to be feasible; however, we do not know any explicit set of inequalities which is *sufficient*. Instead, we will derive an implicit feasibility condition by viewing the MCI problem from the perspective of *linear programming*.

As we already noted for basic CI, the requirements on an improvising distribution in Definition 3.11 form a linear program over variables representing the probability assigned to

each word in I. Although this program is exponentially-large in general (with I having up to $|\Sigma|^n$ elements), observe that with respect to the hard, soft, and randomness constraints, the only differences between words in I are which soft specifications they satisfy. Exploiting this symmetry, we can reduce the program to one variable for every possible combination of satisfied/violated soft specifications. For example, basic CI with $k = 1$ would yield 2 variables, representing the probabilities of admissible and inadmissible improvisations respectively. In general the program will have 2^k variables, which is still exponential but can be practical for small numbers of soft constraints. To define the program formally, we introduce some notation.

Definition 3.14. Fix an MCI instance $\mathcal{C}$. For any $M \subseteq [k]$, define

$$A'_M = I \cap \bigcap_{i \in M} A_i \setminus \bigcup_{i \notin M} A_i,$$

the improvisations that are i-admissible for exactly those i which are elements of M.

For example, $A'_\emptyset$ consists of the improvisations satisfying no soft specifications, and $A'_{\{1,2\}}$ consists of the improvisations satisfying $\mathcal{S}_1$ and $\mathcal{S}_2$ but no other soft specifications. Clearly, the sets A'_M are disjoint and partition I, and $A_i = \cup_{M \ni i} A'_M$ for all $i \in [k]$.

Definition 3.15. For any MCI instance $\mathcal{C}$, the linear program $Lin(\mathcal{C})$ is defined over the variables p_M for $M \subseteq [k]$ by the following equations:

$$\forall M \subseteq [k], \quad p_M \geq 0 \tag{3.1}$$

$$\sum_{M \subseteq [k]} p_M = 1 \tag{3.2}$$

$$\forall i \in [k], \quad \sum_{\substack{M \subseteq [k] \\ M \ni i}} p_M \geq 1 - \epsilon_i \tag{3.3}$$

$$\forall M \subseteq [k], \quad \lambda |A'_M| \leq p_M \leq \rho |A'_M| \tag{3.4}$$

Lemma 3.4. *An MCI instance $\mathcal{C}$ is feasible if and only if $Lin(\mathcal{C})$ is.*

Proof. Suppose $\mathcal{C}$ is feasible, with an improvising distribution D. We claim that $p_M = \sum_{w \in A'_M} D(w)$ is a solution to $Lin(\mathcal{C})$. Since D is a probability distribution, equations (3.1) are trivially satisfied. By the hard constraint, $\sum_{w \in I} D(w) = 1$, and since the sets A'_M partition I we have $\sum_{M \subseteq [k]} p_M = \sum_{M \subseteq [k]} \sum_{w \in A'_M} D(w) = \sum_{w \in I} D(w) = 1$, so equation (3.2) is also satisfied. Similarly, by the soft constraints we have $\sum_{w \in A_i} D(w) \geq 1 - \epsilon_i$ for each $i \in [k]$, and since the sets A'_M with $M \ni i$ partition A_i, we have

$$\sum_{\substack{M \subseteq [k] \\ M \ni i}} p_M = \sum_{\substack{M \subseteq [k] \\ M \ni i}} \sum_{w \in A'_M} D(w) = \sum_{w \in A_i} D(w) \geq 1 - \epsilon_i,$$

satisfying equations (3.3). Finally, by the randomness constraint we have $\lambda \leq D(w) \leq \rho$ for every $w \in I$, so $|A'_M|\lambda \leq p_M \leq |A'_M|\rho$ and equations (3.4) are also satisfied.

Conversely, if $Lin(\mathcal{C})$ has a solution $(p_M)_{M\subseteq[k]}$, for every $w \in I$ define $D(w) = p_{M_w}/|A'_{M_w}|$ where $M_w \subseteq [k]$ is the unique set such that $w \in A'_{M_w}$ (notice this guarantees we do not divide by zero). Since $p_{M_w} \geq 0$ by equations (3.1), we have $D(w) \geq 0$. Furthermore, since the sets A'_M partition I,

$$\sum_{w\in I} D(w) = \sum_{w\in I} \frac{p_{M_w}}{|A'_{M_w}|} = \sum_{M\subseteq[k]} \sum_{w\in A'_M} \frac{p_M}{|A'_M|} = \sum_{M\subseteq[k]} p_M = 1$$

by equation (3.2). So D is a probability distribution, and since its domain is I it satisfies the hard constraint. Similarly, since the sets A'_M with $M \ni i$ partition A_i for each $i \in [k]$, we have

$$\sum_{w\in A_i} D(w) = \sum_{w\in A_i} \frac{p_{M_w}}{|A'_{M_w}|} = \sum_{\substack{M\subseteq[k]\\ M\ni i}} \sum_{w\in A'_M} \frac{p_M}{|A'_M|} = \sum_{\substack{M\subseteq[k]\\ M\ni i}} p_M \geq 1 - \epsilon_i$$

by equations (3.3). So D satisfies the soft constraints. Finally, for any $w \in I$ we have $\lambda \leq p_{M_w}/|A'_{M_w}| \leq \rho$ by equations (3.4), so D also satisfies the randomness constraint and is thus an improvising distribution. □

Thus, although we do not have closed-form conditions for the feasibility of an MCI instance $\mathcal{C}$, we can determine feasibility by constructing $Lin(\mathcal{C})$ and applying linear programming algorithms.

3.5.3 Complexity

Since the linear program $Lin(\mathcal{C})$ is in general exponentially large, using linear programming algorithms to solve MCI problems can take exponential time even for tractable specifications. Therefore the best analogue we can give of our generic scheme for CI (Theorem 3.2) is an exponential-time scheme:

Theorem 3.11. *If* Spec *is a class of specifications for which membership can be decided in exponential time, then there is an exponential-time improvisation scheme for* MCI(Spec,Spec).

Proof. To compute the quantities needed to formulate $Lin(\mathcal{C})$, we enumerate every $w \in I = L(\mathcal{H}) \cap \Sigma^{m:n}$ and check for each one which of the specifications $\mathcal{S}_i$ it satisfies. In the process we record the unique $M_w \subseteq [k]$ such that $w \in A'_{M_w}$, and keep track of how many words are in each A'_M. This can be done in exponential time, since there are exponentially-many words in $\Sigma^{m:n}$ and checking each one against $\mathcal{H}$ and every $\mathcal{S}_i$ takes exponential time. Then we can construct $Lin(\mathcal{C})$, and since linear programming is polynomial-time [97, 94], we can solve the exponentially-large program in exponential time. If it is infeasible, by Lemma 3.4 so is $\mathcal{C}$ and we return $\perp$. Otherwise, we obtain a solution $\vec{p_M}$ to $Lin(\mathcal{C})$, and generating $w \in I$ with probability $p_{M_w}/|A'_{M_w}|$ is an improvising distribution as in the Lemma. □

On the other hand, if we consider the number of soft constraints k to be fixed, then we can solve MCI problems efficiently whenever the specifications support the intersection, difference, counting, and uniform sampling operations we used in our generic scheme for CI (Theorem 3.2):

Theorem 3.12. *Suppose* Spec *is a class of specifications for which the operations in Section 3.3 can be done in polynomial time. Then for any fixed k, there is a polynomial-time improvisation scheme for* $\mathrm{MCI}_k(\textsc{Spec},\textsc{Spec})$.

Proof. Using the intersection and difference operations we can construct specifications $\mathcal{A}_M$ for the sets A'_M for each $M \subseteq [k]$. Since k is fixed, there are only a constant number of these specifications. Furthermore, we can construct them using a constant number of intersection and difference operations, and since these operations take polynomial time, the specifications have polynomial size. Next, applying the counting operation to each $\mathcal{A}_M$, we can find the sizes $|A'_M|$ and then build the linear program $Lin(\mathcal{C})$. The program has polynomial size, since k is fixed and the sizes $|A'_M|$ have polynomial bitwidth (as $A'_M \subseteq I \subseteq \Sigma^n$). So we can solve $Lin(\mathcal{C})$ using a polynomial-time linear programming algorithm.

Now by Lemma 3.4, if $Lin(\mathcal{C})$ is infeasible neither is $\mathcal{C}$ and we return $\perp$. Otherwise, we obtain a solution $(p_M)_{M\subseteq[k]}$ to $Lin(\mathcal{C})$, and the Lemma shows that assigning each $w \in A'_M$ probability $p_M/|A'_M|$ yields an improvising distribution. We can sample from this distribution by picking $M \subseteq [k]$ with probability p_M, then applying the uniform sampling operation to $\mathcal{A}_M$. Since the number of $M \subseteq [k]$ is constant and each $\mathcal{A}_M$ has polynomial size as noted above, this improviser will run in polynomial time. □

Remark. This result shows that, in a sense, MCI is *fixed-parameter tractable* in k: we have an improvisation scheme running in time on the order of $2^k \cdot poly(|\mathcal{C}|)$. For problems with very few soft constraints, this algorithm could indeed be practical. However, for even moderate k, the runtime explodes. The lighting control example we will discuss in Chapter 7 uses $k = 24$, and even if solving a linear program with 2^{24} (∼17 million) variables is reasonable, constructing 2^{24} automata (to compute the size of each set A'_M) is not.

Towards a lower bound, recall that even though multiple hard constraints can be combined into a single hard constraint by taking their product, this can cause a blowup in the size of the specification. For example, while finding a word in the language of a DFA is easy, finding one in the intersection of an arbitrary number of DFAs is PSPACE-hard [100]. In our setting, we only want to find words of bounded length, but we also need to count them, which makes the problem #P-hard.

Definition 3.16. #Bounded-DFA-Int is the problem of computing, given DFAs $D_1, \ldots, D_k$ over an alphabet Σ and $n \in \mathbb{N}$ in unary, the number of words in $L(D_1) \cap \cdots \cap L(D_k) \cap \Sigma^n$.

Lemma 3.5. #Bounded-DFA-Int *is #P-complete.*

Proof. The problem is clearly in #P, since in polynomial time we can nondeterministically pick a word in Σ^n and check if every D_i accepts it. In the other direction, we give a

reduction from #SAT similar to that used by Kannan et al. [93] to prove hardness of counting the language of an NFA. Suppose we are given a formula F in conjunctive normal form, i.e. $F = c_1 \wedge \cdots \wedge c_k$ where each c_i is a disjunction of variables and their negations. If F has n variables V, putting them in some order we can view assignments to V as words in Σ^n where $\Sigma = \{0, 1\}$. Then for each c_i we can build a DFA D_i accepting assignments that satisfy c_i. We have one state for each $v \in V$, and start from the first variable under the order. If in state v we read a 1 and v occurs positively in c_i, or we read a 0 and v occurs negatively, then the assignment satisfies c_i and we transition to a chain of states that ensure we accept if and only if the word has length exactly n. Otherwise c_i is not yet satisfied, so we move to the state for the next variable in the order. If we are already at the last variable, then c_i is not satisfied by the assignment and we reject. Clearly, each D_i has size polynomial in $|F|$, and the intersection $\cap_i L(D_i)$ contains precisely the satisfying assignments of F. □

We can use this to lower bound the complexity of MCI with DFA specifications:

Theorem 3.13. MCI(DFA,DFA) *is #P-hard.*

Proof. We reduce #Bounded-DFA-Int to MCI(DFA,DFA) along similar lines to Theorem 3.5. For any $N \in \mathbb{N}$, consider the MCI instance $\mathcal{C}_N = (\mathcal{T}, D_1, \ldots, D_k, n, n, 0, \ldots, 0, 0, 0, 1/N)$ where $\mathcal{T}$ is the trivial DFA accepting all of Σ^*. For this instance we have $I = \Sigma^n$ and $A_i = L(D_i)$ for every $i \in \{1, \ldots, k\}$. Since $\epsilon_i = 0$ for every i, an improvising distribution for $\mathcal{C}_N$ must assign probability zero to all words not in $\cap_i L(D_i)$. Therefore since no word can be assigned probability greater than ρ, if an improvising distribution exists then $|\cap_i L(D_i)| \geq 1/\rho = N$. Conversely, if $|\cap_i L(D_i)| \geq N$ then a uniform distribution on $\cap_i L(D_i)$ is clearly an improvising distribution for $\mathcal{C}_N$. So $\mathcal{C}_N$ is feasible if and only if $|\cap_i L(D_i)| \geq N$. Finally, since $|\cap_i L(D_i)| \leq |\Sigma|^n = 2^n$, by binary search we can determine $|\cap_i L(D_i)|$ with polynomially-many MCI(DFA,DFA) queries, and thereby count the satisfying assignments of F. □

Recall that by Theorem 3.4, there is a polynomial-time improvisation scheme for CI(DFA,DFA). Thus, allowing multiple constraints increases the complexity from P to at least #P. On the other hand, note that we only used hard constraints, i.e. soft constraints with $\epsilon = 0$, in the proof of Theorem 3.13. For MCI problems with only hard constraints, there is a polynomial-time improvisation scheme relative to a #P oracle by Theorem 3.3, since we can check all hard specifications at once with a single polynomial-time algorithm (which simply runs the algorithms for each specification in sequence). Therefore, having multiple hard constraints increases the complexity to #P, but no further. This raises the natural question of whether multiple *soft* constraints can raise the complexity even higher: theoretically, it could reach as high as EXP before being limited by Theorem 3.11. Closing this gap between #P and EXP is a clear direction for future work: a multi-constraint version of Theorem 3.1 establishing closed-form feasibility conditions for MCI would be helpful here.

Table 3.1: Complexity of the control improvisation problem for various types of hard and soft specifications $\mathcal{H}$ and $\mathcal{S}$. Here #P indicates that feasibility checking is #P-hard, and that there is a polynomial-time improvisation scheme relative to a #P oracle.

$\mathcal{H}$ \ $\mathcal{S}$		**DFA**	**CFG** unamb.	**CFG** amb.	**NFA**
DFA		poly-time	poly-time		
CFG	unambiguous	poly-time	#P		
	ambiguous			#P	
NFA					#P

3.6 Summary and Future Work

In this chapter, we introduced control improvisation, the problem of synthesizing improvisers which generate finite words subject to hard, soft, and randomness constraints. The CI problem provides a firm theoretical footing for the concept of algorithmic improvisation discussed in the Introduction, and is the basis for more sophisticated forms of algorithmic improvisation we will study in later chapters. We thoroughly studied the theory of CI, giving precise conditions for when improvisers exist and a general framework for solving CI problems. We used this framework to give polynomial-time improvisation schemes for CI problems with deterministic finite automata and unambiguous context-free grammar specifications. We also showed that for nondeterministic automata and general context-free grammars, CI is #P-hard and so polynomial-time improvisation schemes are unlikely to exist. Our complexity results are summarized in Table 3.1. For specifications given symbolically by Boolean formulas, we showed that it is possible to approximately solve CI problems using SAT solvers. Finally, we explored a generalized form of CI allowing multiple soft constraints, and gave evidence suggesting that it is substantially more difficult than the basic problem: even for DFA specifications it is #P-hard, and our best improviser construction is only in EXP.

As we saw in several examples above, control improvisation is useful in a number of applications including robotic planning and human modeling, which we will study in Chapters 6 and 7. There are also several interesting directions for additional theoretical work:

More Tractable Cases of CI. So far we have only studied broad classes of specifications based on different formalisms, like finite automata or context-free grammars. There might well be other practically-useful classes of CI problems which are tractable because of finer structural properties of the automata involved, for example, rather than just determinism.

Continuous Signals. Our definition of CI is discrete both in time and in space, with an improviser choosing symbols from a finite set at each discrete step. Even the rela-

tively simple extensions to continuous alphabets (e.g. real numbers), or to sequences of time-stamped events over continuous time, would already be highly interesting (see Chapter 8 for one potential application). The further extension to continuous signals would enable test generation for a broad class of cyber-physical systems taking such signals as input.

Quantitative Soft Constraints. We studied a particular type of soft constraint, namely one requiring that a Boolean property hold with at least some given probability. Many other types of soft constraints are possible, for example requiring that some objective function be close to its optimal value in expectation. Such constraints could be useful in robotic planning, where we might want a generated route to be as short as possible on average while still allowing a certain amount of randomness.

Other Randomness Constraints. The randomness constraint we used above is essentially the simplest possible one (other than just requiring a uniform distribution). Various other types of randomness constraints would be useful in applications: for example, when generating synthetic data to train or test a machine learning-based system, we need much finer control over the distribution of the data than simply requiring it to be close to uniform. We will discuss this application in depth in Chapters 5 and 8. Another useful type of randomness constraint would be to directly bound a measure of unpredictability like the entropy in the patrolling robot application. Finally, for applications like robotic planning and music improvisation where there are natural metrics on the space of improvisations, we could more directly ensure diversity by imposing requirements on the distance between successive improvisations or on the overall coverage of the space.

Chapter 4

Reactive Control Improvisation

The control improvisation problem we studied in Chapter 3 is an *offline* synthesis problem in the sense that an improviser generates a word all at once. However, many interesting systems, including protocol handlers, robotic task planners, and concurrent software generally, are *open* systems that interact over time with an external environment. Synthesis of such *reactive systems* requires finding an implementation that chooses actions *online* in response to the environment, ensuring the desired specification is satisfied no matter what the environment does. This problem, *reactive synthesis*, has a long history going back to Church [33] (see Finkbeiner [56] for a survey). Reactive synthesis from temporal logic specifications [136] has been particularly well-studied and is being increasingly used in applications such as hardware synthesis [18] and robotic task planning [101].

In this chapter, we introduce a *reactive* version of control improvisation [67] which allows synthesizing reactive systems with *random behavior*: in fact, systems where *being random in a prescribed way is part of their specification*. This is in contrast to prior work on stochastic games where randomness is used to model uncertain environments or randomized strategies are merely allowed, not required. Solvers for stochastic games may incidentally produce randomized strategies to satisfy a functional specification (and some types of specification, e.g. multi-objective queries [31], may only be realizable by randomized strategies), but do not provide a general way to *enforce* randomness. Unlike most specifications used in reactive synthesis, our randomness requirement is a property of a system's *distribution* of behaviors, not of an individual behavior. While probabilistic specification languages like PCTL [85] can capture some such properties, the simple and natural randomness requirement of control improvisation cannot be concisely expressed by existing languages (even those as powerful as Stochastic Game Logic [11], which can quantify over strategies). Thus, *randomized reactive synthesis* in our sense requires significantly different methods than those previously studied.

However, we argue that this type of synthesis is quite useful: as we have argued above, introducing randomness into the behavior of a system can often be beneficial, enhancing *variety*, *robustness*, and *unpredictability*. Example applications include:

- Synthesizing a black-box fuzz tester for a network service, we want a client program

that not only conforms to the protocol (perhaps only most of the time) but can generate many different sequences of packets even if the server being tested responds in a fixed way: randomness ensures this.

- Synthesizing a controller for a robot exploring an unknown environment, randomness provides a low-memory way to increase coverage of the space. It can also help to reduce systematic bias in the exploration procedure.

- Synthesizing a controller for a patrolling surveillance robot, introducing randomness in planning makes the robot's future location harder to predict.

We will discuss applications to robotic planning in more detail in Chapter 6.

Our *reactive control improvisation (RCI)* problem is defined similarly to the CI problem of Chapter 3: we again have hard, soft, and randomness constraints. The primary difference is that an improviser now generates a word incrementally, alternating adding symbols with a potentially-adversarial environment. This allows us to perform reactive synthesis in a finite window, encoding functional specifications and environment assumptions into the hard constraint, while the soft and randomness constraints allow us to tune how randomness is added to the system. The improviser obtained by solving the RCI problem is then a solution to the original synthesis problem. We define the RCI problem formally in Section 4.1.

We begin our study of the theory of RCI in Section 4.2 by analyzing when RCI problems are solvable, or *realizable* (in the terminology of reactive synthesis). To do this, we introduce the notion of the *width* of a game, which generalizes the concept of a "winning" position by *counting* how many ways a player can win from the position. Using width, we give precise conditions for an RCI instance to be realizable. Although these conditions are quite similar to those we gave in Chapter 3 for CI, the construction of an improviser is significantly more complex in the reactive setting. Notably, our improviser requires *memory*, and we show this is necessary even for simple reachability and safety games where memoryless strategies suffice for ordinary reactive synthesis.

As was the case for CI, our constructive proof of the feasibility conditions allows us to give a generic procedure for building RCI improvisers in Section 4.3. Whenever it is possible to efficiently intersect specifications and compute their widths, we obtain an efficient improvisation scheme. We also give a general recursive algorithm for computing widths, which yields a polynomial-space improvisation scheme for any specifications testable in polynomial space.

Next, in Section 4.4 we study the complexity of RCI in more detail for several interesting classes of specifications. We develop a polynomial-time improvisation scheme for deterministic finite automata, which also applies to reachability and safety games since these can be encoded as DFAs. For nondeterministic automata, the complexity increases to PSPACE, as we show that even finding a single winning strategy for a game whose winning set is given by an NFA is PSPACE-hard. This hardness carries over to context-free grammars, Boolean formulas, and temporal logic formulas.

Finally, we conclude in Section 4.5 with a summary and directions for future work.

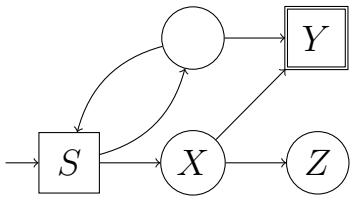


Figure 4.1: An example of a reachability game.

4.1 Problem Definition

4.1.1 Synthesis Games

Reactive control improvisation will be formalized in terms of a 2-player game which is essentially the standard *synthesis game* used in reactive synthesis [56]. However, our formulation is slightly different for compatibility with the definition of control improvisation, so we give a self-contained presentation here.

Our definitions will be general enough to capture many types of games used in practice for reactive synthesis (modulo the restriction to finite words, which we discuss further below). In particular, as examples we will often use *reachability games* [117], where players' actions cause transitions in a state space and the goal is to reach a target state. We group these games, *safety games* where the goal is to *avoid* a set of states, and *reach-avoid* games combining reachability and safety goals [172] together as *reachability/safety games* (RSGs). We draw reachability games in the usual way, shown in Figure 4.1: squares are adversary-controlled states, and states with a double border are target states (mirroring the accepting state notation we use for automata). In this game, if the adversary decides to move from the initial position S to X, then we can move to Y and win, or to Z and lose.

Now we proceed to define our formalism for games. Fix a finite alphabet Σ. The players of the game will alternate picking symbols from Σ, building up a word. We can then specify the set of winning plays with a language over Σ. To simplify our presentation we assume that players strictly alternate turns and that any symbol from Σ is a legal move. These assumptions can be relaxed in the usual way by modifying the winning set appropriately.

Remark. To model a game where the allowed actions can change each turn (e.g. reachability games, where the set of states we can move to depends on the current state), we can use the standard method of modifying the winning set of that any player who makes an illegal move automatically loses. In particular, to view a reachability game G with a set of states Σ as a language over Σ, we let our language L of winning plays consist of all paths in G ending in a target state (the original winning plays for the reachability game), together with all words in Σ^* which are not paths in G and such that it is the adversary's turn during the first move causing the word to fail to be a path (an example in Figure 4.1 would be the word SZX, since the moves from S to Z and from Z to X are both illegal, but the adversary makes an illegal move first). Then if we synthesize a strategy ensuring we get a play in L against

any adversary, it is a winning strategy for the original game G: an adversary for G always follows edges in G (unless they are at a state with no outgoing edges, but then the game ends), so in order to yield a word in L our strategy must also. In fact, the same type of procedure allows us to encode arbitrary assumptions on the adversary, which will be useful in cases like our robotic planning example in Chapter 6: such planning problems typically do not admit solutions without making assumptions on the environment.

Likewise, we can relax the assumption that players strictly alternate, since it is straightforward to modify our algorithms to work with a function specifying the current player as a function of what has been played so far. Alternatively we can use games like reachability games, which do not have strict alternation, as specifications for our algorithms below by inserting extra states in the usual way to produce strict alternation. Note that there is a one-to-one correspondence between plays before and after this transformation, so it will not affect the randomness requirement of RCI.

Finite words: While reactive synthesis is usually considered over infinite words, here we focus on synthesis in a finite window, as it is unclear how best to generalize our randomness requirement to the infinite case. This assumption is not too restrictive, as solutions of bounded length are adequate for many applications. In fuzz testing, for example, we do not want to generate arbitrarily long files or sequences of packets. In robotic planning, we often want a plan that accomplishes a task within a certain amount of time. Furthermore, planning problems with liveness specifications can often be segmented into finite pieces: we do not need an infinite route for a patrolling robot, but can plan within a finite horizon and replan periodically. Replanning may even be *necessary* when environment assumptions become invalid. At any rate, we will see that the bounded case of reactive control improvisation is already highly nontrivial.

As a final simplification, we require that all plays have length exactly $n \in \mathbb{N}$. To allow a range $[m, n]$ we can simply add a new padding symbol to Σ and extend all shorter words to length n, modifying the winning set appropriately.

Definition 4.1. A *history* h is an element of $\Sigma^{\leq n}$, representing the moves of the game played so far. At the start of the game, the history is the empty word, which we write λ. We say the game has *ended* after h if $|h| = n$; otherwise it is *our turn* after h if $|h|$ is even, and *the adversary's turn* if $|h|$ is odd.

Definition 4.2. A *strategy* is a function $\sigma : \Sigma^{\leq n} \times \Sigma \to [0, 1]$ such that for any history $h \in \Sigma^{\leq n}$ with $|h| < n$, $\sigma(h, \cdot)$ is a probability distribution over Σ. We write $x \leftarrow \sigma(h)$ to indicate that x is a symbol randomly drawn from $\sigma(h, \cdot)$.

Since strategies are randomized, fixing strategies for both players does not uniquely determine a play of the game, but defines a *distribution* over plays:

Definition 4.3. Given a pair of strategies (σ, τ), we can generate a random *play* $\pi \in \Sigma^n$ as follows. Pick $\pi_0 \leftarrow \sigma(\lambda)$, then for i from 1 to $n - 1$ pick $\pi_i \leftarrow \tau(\pi_0 \dots \pi_{i-1})$ if i is odd and $\pi_i \leftarrow \sigma(\pi_0 \dots \pi_{i-1})$ otherwise. Finally, put $\pi = \pi_0 \dots \pi_{n-1}$. We write $P_{\sigma,\tau}(\pi)$ for the

probability of obtaining the play π. This extends to a *set* of plays $X \subseteq \Sigma^n$ in the natural way: $P_{\sigma,\tau}(X) = \sum_{\pi \in X} P_{\sigma,\tau}(\pi)$. Finally, the set of *possible* plays is $\Pi_{\sigma,\tau} = \{\pi \in \Sigma^n \mid P_{\sigma,\tau}(\pi) > 0\}$.

The next definition is just the conditional probability of a play given a history, but works for histories with probability zero, simplifying our presentation.

Definition 4.4. For any history $h = h_0 \dots h_{k-1} \in \Sigma^{\leq n}$ and word $\rho \in \Sigma^{n-k}$, we write $P_{\sigma,\tau}(\rho|h)$ for the probability that if we assign $\pi_i = h_i$ for $i < k$ and sample $\pi_k, \dots, \pi_{n-1}$ by the process above, then $\pi_k \dots \pi_{n-1} = \rho$.

We use without comment several basic facts about $P_{\sigma,\tau}(\rho|h)$, all immediate from its definition:

Lemma 4.1. *For any history $h \in \Sigma^{\leq n}$, word $\rho \in \Sigma^{n-|h|}$, and strategies σ, τ:*

(1) if $|h| = 0$, then $P_{\sigma,\tau}(\rho|h) = P_{\sigma,\tau}(\rho)$;

(2) if $|h| = n$, then $P_{\sigma,\tau}(\rho|h) = 1$;

(3) if $|h| < n$, then $\rho = u\rho'$ for some $u \in \Sigma$, and:

- *a) if it is our turn after h, then $P_{\sigma,\tau}(\rho|h) = \sigma(h,u) \cdot P_{\sigma,\tau}(\rho'|hu)$;*
- *b) if it is the adversary's turn after h, then $P_{\sigma,\tau}(\rho|h) = \tau(h,u) \cdot P_{\sigma,\tau}(\rho'|hu)$.*

4.1.2 The Reactive Control Improvisation Problem

Reactive control improvisation is a direct generalization of the control improvisation problem we studied in Chapter 3, being a randomized synthesis problem subject to hard, soft, and randomness constraints. The main difference in RCI is that these constraints are interpreted not over just the output of the improviser, but over a *play* of the synthesis game described above; i.e., the combined trace $\pi \in \Sigma^n$ including the actions of both the improviser and the adversary. We will use the same terminology as Chapter 3 to describe the specifications and languages defining the hard and soft constraints:

Definition 4.5. Given *hard* and *soft* specifications $\mathcal{H}$ and $\mathcal{S}$ of languages over Σ, an *improvisation* is a word $w \in L(\mathcal{H}) \cap \Sigma^n$. It is *admissible* if $w \in L(\mathcal{S})$. The set of all improvisations is denoted I, and admissible improvisations A.

The constraints of RCI are then defined analogously to CI: the hard constraint requires that we only generate improvisations, the soft constraint requires that the generated improvisation be admissible with probability at least $1 - \epsilon$ for some error probability ϵ, and the randomness constraint requires that no improvisation be generated with probability greater than a specified ρ.

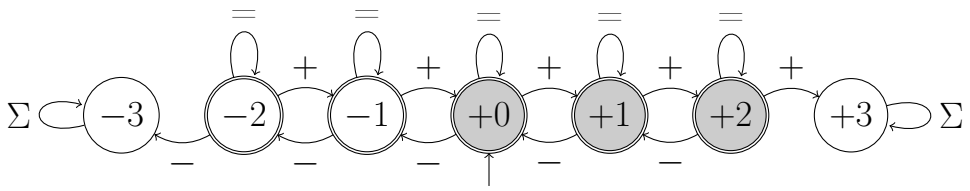


Figure 4.2: The hard specification DFA $\mathcal{H}$ in our running example. The soft specification $\mathcal{S}$ is the same but with only the shaded states accepting.

Running Example. We will use the following simple example throughout the chapter: each player may increment ($+$), decrement ($-$), or leave unchanged ($=$) a counter which is initially zero. We require that the counter must always stay within $[-2, 2]$, and furthermore that it end at a nonnegative value at least 3/4 of the time. Finally, we want the improviser to generate at least 16 different behaviors, all of length 4. We can formalize this as an RCI problem as follows:

Alphabet. $\Sigma = \{+, -, =\}$, representing the 3 actions the players may take.

Length Bound. $n = 4$, so that the game lasts for 4 moves.

Hard Constraint. $\mathcal{H}$ is a DFA, shown in Figure 4.2, encoding the property that the counter stay within $[-2, 2]$.

Soft Constraint. $\mathcal{S}$ is a similar DFA, also shown in Figure 4.2, encoding the property that the counter end at a nonnegative value (given $\mathcal{H}$). Since we want this property to hold 3/4 of the time, we put the error probability $\epsilon = 1/4$.

Randomness Constraint. $\rho = 1/16$, which implies the improviser can generate at least 16 words.

Then for example the word $++==$ is an admissible improvisation, satisfying both hard and soft constraints, and so is in A. The word $+-=-$ on the other hand satisfies $\mathcal{H}$ but not $\mathcal{S}$ (the counter ends at -1), so it is in I but not A. Finally, $+++-$ does not satisfy $\mathcal{H}$ (the counter exceeds 2 at some point), so it is not an improvisation at all and is not in I.

A reactive control improvisation problem is defined by $\mathcal{H}$, $\mathcal{S}$, ϵ, and ρ, and a solution is a strategy which ensures that the hard, soft, and randomness constraints hold against every adversary. Formally, following our definition in Chapter 3:

Definition 4.6. Given an *RCI instance* $\mathcal{C} = (\mathcal{H}, \mathcal{S}, n, \epsilon, \rho)$ with $\mathcal{H}$, $\mathcal{S}$, and n as above and $\epsilon, \rho \in [0, 1] \cap \mathbb{Q}$, a strategy σ is an *improvising strategy* if it satisfies the following requirements for every adversary τ:

Hard constraint: $P_{\sigma,\tau}(I) = 1$

Soft constraint: $P_{\sigma,\tau}(A) \geq 1 - \epsilon$

Randomness: $\forall \pi \in I,\ P_{\sigma,\tau}(\pi) \leq \rho$.

If there is an improvising strategy σ, we say that $\mathcal{C}$ is *realizable*. An *improviser* for $\mathcal{C}$ is then an expected-finite time probabilistic algorithm implementing such a strategy σ, i.e. whose output distribution on input $h \in \Sigma^{\leq n}$ is $\sigma(h, \cdot)$.

Definition 4.7. Given an RCI instance $\mathcal{C} = (\mathcal{H}, \mathcal{S}, n, \epsilon, \rho)$, the *reactive control improvisation* (RCI) problem is to decide whether $\mathcal{C}$ is realizable, and if so to generate an improviser for $\mathcal{C}$.

Running Example. Suppose we set $\epsilon = 1/2$ and $\rho = 1/2$. Let σ be the strategy which picks $+$ or $-$ with equal probability in the first move, and thenceforth picks the action which moves the counter closest to ± 1 respectively. This satisfies the hard constraint, since if the adversary ever moves the counter to ± 2 we immediately move it back. The strategy also satisfies the soft constraint, since with probability $1/2$ we set the counter to $+1$ on the first move, and if the adversary moves to 0 we move back to $+1$ and remain nonnegative. Finally, σ also satisfies the randomness constraint, since each choice of first move happens with probability $1/2$ and so no play can be generated with higher probability. So σ is an improvising strategy and this RCI instance is realizable.

Remark. Notice that the randomness constraint in RCI only imposes an upper bound ρ on the probability of a trace, whereas in CI we also allowed a lower bound λ. This is because in the reactive setting, lower bounds can only be satisfied trivially. Specifically, in any game where the adversary can ever choose between multiple actions, they can ensure some traces have probability zero by never taking one of the actions. So RCI problems with $\lambda > 0$ are always unrealizable except in cases where the improviser need not be reactive at all.

As in Chapter 3, we will study classes of RCI problems with different types of specifications:

Definition 4.8. If HSPEC and SSPEC are classes of specifications, then the class of RCI instances $\mathcal{C} = (\mathcal{H}, \mathcal{S}, n, \epsilon, \rho)$ where $\mathcal{H} \in$ HSPEC and $\mathcal{S} \in$ SSPEC is denoted RCI (HSPEC, SSPEC). We use the same notation for the decision problem associated with the class, i.e., given $\mathcal{C} \in$ RCI (HSPEC, SSPEC), decide whether $\mathcal{C}$ is realizable. The *size* $|\mathcal{C}|$ of an RCI instance is the total size of the bit representations of its parameters, with n represented in unary and ϵ, ρ in binary.

Finally, a *synthesis algorithm* in our context takes a specification in the form of an RCI instance and produces an implementation in the form of an improviser. This corresponds exactly to the notion of an improvisation scheme from Chapter 3, simply using the RCI definitions of improvisers and realizability:

Definition 4.9. A *polynomial-time improvisation scheme* for a class $\mathcal{P}$ of RCI instances is an algorithm S with the following properties:

Correctness: For any $\mathcal{C} \in \mathcal{P}$, if $\mathcal{C}$ is realizable then $S(\mathcal{C})$ is an improviser for $\mathcal{C}$, and otherwise $S(\mathcal{C}) = \perp$.

Scheme efficiency: There is a polynomial $p : \mathbb{R} \to \mathbb{R}$ such that the runtime of S on any $\mathcal{C} \in \mathcal{P}$ is at most $p(|\mathcal{C}|)$.

Improviser efficiency: There is a polynomial $q : \mathbb{R} \to \mathbb{R}$ such that for every $\mathcal{C} \in \mathcal{P}$, if $G = S(\mathcal{C}) \neq \perp$ then G has expected runtime at most $q(|\mathcal{C}|)$.

As before, the first two requirements simply say that the scheme produces valid improvisers in polynomial time. The third is necessary to ensure that the improvisers themselves are efficient: otherwise, the scheme might for example produce improvisers running in time exponential in the size of the specification.

A main goal of our paper is to determine for which types of specifications there exist polynomial-time improvisation schemes. While we do find such algorithms for important classes of specifications, we will also see that determining the realizability of an RCI instance is often PSPACE-hard. Therefore we also consider *polynomial-space improvisation schemes*, defined as above but replacing time with space.

4.2 Existence of Improvisers

The most basic question in reactive synthesis is whether a specification is realizable. In *randomized* reactive synthesis, the question is more delicate because the randomness requirement means that it is no longer enough to ensure some property regardless of what the adversary does: there must be *many ways* to do so. Specifically, the same argument we used for CI in Chapter 3 applies: there must be at least $1/\rho$ improvisations if we are to generate each of them with probability at most ρ. However, in the reactive setting there is an additional complication: at least this many improvisations must be *possible* given an unknown adversary: even if many exist, the adversary may be able to force us to use only a single one.

To address this issue, we introduce a new notion of the size of a set of plays that takes the adversary into account: *width*. Below, we define width, establish its main properties, and use it to formulate the realizability conditions for RCI. As for CI, our proof of these conditions will be constructive; however, building an improviser is much more involved in the reactive case, so we present a general overview in Section 4.2.2 before giving the details in Section 4.2.3.

4.2.1 Width and Realizability

The *width* of a set of plays simply measures how many of the plays are actually possible, assuming the adversary tries to minimize this number:

Definition 4.10. The *width* of $X \subseteq \Sigma^n$ is $W(X) = \max_\sigma \min_\tau |X \cap \Pi_{\sigma,\tau}|$.

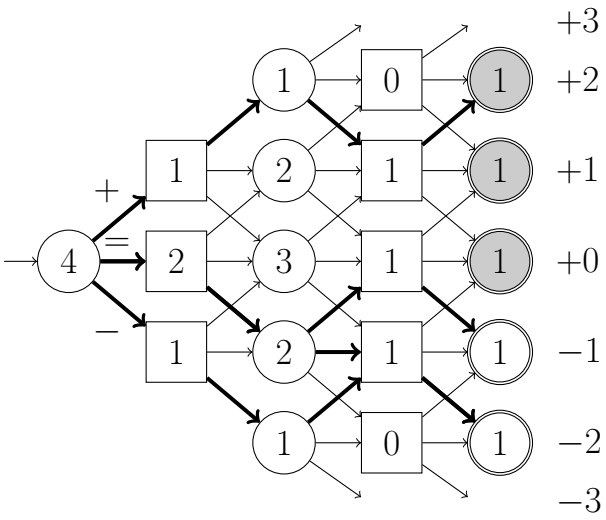


Figure 4.3: Synthesis game for our running example. States are labeled with the widths of I (at left) and A (right) given a history ending at that state.

Thus, the width counts how many distinct plays can be generated regardless of what the adversary does. Intuitively, a "narrow" game — one whose set of winning plays has small width — is one in which the adversary can force us to choose among only a few winning plays, while in a "wide" one we always have many safe choices available. Note that *which* particular plays can be generated depends on the adversary: the width only measures *how many* can be generated. For example, $W(X) = 1$ means that a play in X can always be generated, but it could be a different element of X for different adversaries.

Running Example. Figure 4.3 shows the synthesis game for our running example: paths ending in circled or shaded states are plays in I or A respectively (ignore the state labels for now). At left, the bold arrows show the 4 plays in I possible against the adversary that moves away from 0, and down at 0. This shows $W(I) \leq 4$, and in fact 4 plays are possible against any adversary, so $W(I) = 4$. Similarly, at right we see that $W(A) = 1$.

It will be useful later to have a *relative* version of width that counts how many plays are possible *from a given position*:

Definition 4.11. Given a set of plays $X \subseteq \Sigma^n$ and a history $h \in \Sigma^{\leq n}$, the *width of* X *given* h is $W(X|h) = \max_\sigma \min_\tau |\{\pi \mid h\pi \in X \wedge P_{\sigma,\tau}(\pi|h) > 0\}|$.

This is a direct generalization of the concept of "winning" positions: if X is the set of winning plays, then $W(X|h)$ counts the number of ways to win from h.

The next lemma establishes several fundamental properties of $W(X|h)$ which we will use frequently throughout the chapter without further comment. Note in particular that properties (d)–(f) provide a recursive way to compute widths. These recursive relations, illustrated by the state labels in Figure 4.3, will be critical in our improviser construction.

Lemma 4.2. *For any set of plays $X \subseteq \Sigma^n$ and history $h \in \Sigma^{\leq n}$:*

(a) ***(Bounds)*** $0 \leq W(X|h) \leq |\Sigma|^{n-|h|}$*;*

(b) ***(Monotonicity)*** *if $Y \subseteq X$, then $W(Y|h) \leq W(X|h)$;*

(c) ***(Game Start)*** $W(X|\lambda) = W(X)$*;*

(d) ***(Game End)*** *if $|h| = n$, then $W(X|h) = \mathbf{1}_{h \in X}$;*

(e) ***(Our Turn)*** *if it is our turn after h, then $W(X|h) = \sum_{u \in \Sigma} W(X|hu)$;*

(f) ***(Adversary Turn)*** *if it is the adversary's turn after h, then $W(X|h) = \min_{u \in \Sigma} W(X|hu)$.*

Proof.

(a) By definition, $W(X|h) = \max_\sigma \min_\tau |\{\pi \mid h\pi \in X \wedge P_{\sigma,\tau}(\pi|h) > 0\}|$, so $W(X|h) \geq 0$ trivially. Since $X \subseteq \Sigma^n$, if $h\pi \in X$ then $\pi \in \Sigma^{n-|h|}$. So $W(X|h) \leq |\Sigma|^{n-|h|}$.

(b) Let $\hat{\sigma}$ be a strategy witnessing $W(Y|h)$ (i.e., which achieves the maximum in the definition of $W(Y|h)$). By the definition of $W(X|h)$, there is some strategy $\hat{\tau}$ such that $|\{\pi \mid h\pi \in X \wedge P_{\hat{\sigma},\hat{\tau}}(\pi|h) > 0\}| \leq W(X|h)$. Then $W(Y|h) \leq |\{\pi \mid h\pi \in Y \wedge P_{\hat{\sigma},\hat{\tau}}(\pi|h) > 0\}| \leq |\{\pi \mid h\pi \in X \wedge P_{\hat{\sigma},\hat{\tau}}(\pi|h) > 0\}| \leq W(X|h)$.

(c) $W(X|\lambda) = \max_\sigma \min_\tau |\{\pi \mid \pi \in X \wedge P_{\sigma,\tau}(\pi) > 0\}| = \max_\sigma \min_\tau |X \cap \Pi_{\sigma,\tau}| = W(X)$.

(d) If $h \in X$, then the only word of the form $h\pi$ in X is h, with $\pi = \lambda$ (and $P_{\sigma,\tau}(\lambda|h) = 1 > 0$). Otherwise there is no word of the form $h\pi$ in X.

(e) Since it is our turn, and in particular the game has not ended, every play $h\pi$ that can be generated given history h has the form $hu\pi'$ for some $u \in \Sigma$. So for any strategies σ and τ we have $|\{\pi \mid h\pi \in X \wedge P_{\sigma,\tau}(\pi|h) > 0\}| = \sum_{u \in \Sigma} |\{\pi' \mid hu\pi' \in X \wedge P_{\sigma,\tau}(u\pi'|h) > 0\}| = \sum_{u \in \Sigma} |\{\pi' \mid hu\pi' \in X \wedge \sigma(h,u) \cdot P_{\sigma,\tau}(\pi'|hu) > 0\}|$.

For each $u \in \Sigma$, let σ_u be a strategy witnessing $W(X|hu)$. Let $\tilde{\sigma}$ be a strategy which on history h picks $u \in \Sigma$ uniformly at random, and on histories prefixed by hu follows

σ_u (otherwise picking arbitrarily). Then

$$\begin{aligned}
W(X|h) &\geq \min_\tau |\{\pi \mid h\pi \in X \wedge P_{\tilde{\sigma},\tau}(\pi|h) > 0\}| \\
&= \min_\tau \sum_{u\in\Sigma} |\{\pi' \mid hu\pi' \in X \wedge \tilde{\sigma}(h,u) \cdot P_{\tilde{\sigma},\tau}(\pi'|hu) > 0\}| \\
&= \min_\tau \sum_{u\in\Sigma} |\{\pi' \mid hu\pi' \in X \wedge P_{\tilde{\sigma},\tau}(\pi'|hu) > 0\}| \\
&\geq \sum_{u\in\Sigma} \min_\tau |\{\pi' \mid hu\pi' \in X \wedge P_{\tilde{\sigma},\tau}(\pi'|hu) > 0\}| \\
&= \sum_{u\in\Sigma} \min_\tau |\{\pi' \mid hu\pi' \in X \wedge P_{\sigma_u,\tau}(\pi'|hu) > 0\}| \\
&= \sum_{u\in\Sigma} W(X|hu).
\end{aligned}$$

For the other direction, let $\tilde{\sigma}$ be a strategy witnessing $W(X|h)$, and for each $u \in \Sigma$ let $\tau_u = \arg\min_\tau |\{\pi' \mid hu\pi' \in X \wedge P_{\tilde{\sigma},\tau}(\pi'|hu) > 0\}|$. Let $\tilde{\tau}$ be a strategy which on histories prefixed by hu follows τ_u (otherwise picking arbitrarily). Then

$$\begin{aligned}
W(X|h) &\leq |\{\pi \mid h\pi \in X \wedge P_{\tilde{\sigma},\tilde{\tau}}(\pi|h) > 0\}| \\
&= \sum_{u\in\Sigma} |\{\pi' \mid hu\pi' \in X \wedge \tilde{\sigma}(h,u) \cdot P_{\tilde{\sigma},\tilde{\tau}}(\pi'|hu) > 0\}| \\
&\leq \sum_{u\in\Sigma} |\{\pi' \mid hu\pi' \in X \wedge P_{\tilde{\sigma},\tilde{\tau}}(\pi'|hu) > 0\}| \\
&= \sum_{u\in\Sigma} |\{\pi' \mid hu\pi' \in X \wedge P_{\tilde{\sigma},\tau_u}(\pi'|hu) > 0\}| \\
&\leq \sum_{u\in\Sigma} W(X|hu).
\end{aligned}$$

(f) Since it is the adversary's turn (and in particular the game has not ended), for any strategies σ and τ we have $|\{\pi \mid h\pi \in X \wedge P_{\sigma,\tau}(\pi|h) > 0\}| = \sum_{u\in\Sigma} |\{\pi' \mid hu\pi' \in X \wedge P_{\sigma,\tau}(u\pi'|h) > 0\}| = \sum_{u\in\Sigma} |\{\pi' \mid hu\pi' \in X \wedge \tau(h,u) \cdot P_{\sigma,\tau}(\pi'|hu) > 0\}|$.

For each $u \in \Sigma$, let σ_u be a strategy witnessing $W(X|hu)$. Let $\tilde{\sigma}$ be a strategy which

on histories prefixed by hu follows σ_u (otherwise picking arbitrarily). Then

$$\begin{aligned}
W(X|h) &\geq \min_{\tau} |\{\pi \mid h\pi \in X \wedge P_{\tilde{\sigma},\tau}(\pi|h) > 0\}| \\
&= \min_{\tau} \sum_{u \in \Sigma} |\{\pi' \mid hu\pi' \in X \wedge \tau(h,u) \cdot P_{\tilde{\sigma},\tau}(\pi'|hu) > 0\}| \\
&\geq \min_{\tau} \min_{u \in \Sigma} |\{\pi' \mid hu\pi' \in X \wedge P_{\tilde{\sigma},\tau}(\pi'|hu) > 0\}| \\
&= \min_{u \in \Sigma} \min_{\tau} |\{\pi' \mid hu\pi' \in X \wedge P_{\sigma_u,\tau}(\pi'|hu) > 0\}| \\
&= \min_{u \in \Sigma} W(X|hu).
\end{aligned}$$

For the other direction, let $\tilde{\sigma}$ be a strategy witnessing $W(X|h)$, and define $\tilde{u} = \arg\min_{u \in \Sigma} W(X|hu)$, and $\tau_{\tilde{u}} = \arg\min_{\tau} |\{\pi' \mid h\tilde{u}\pi' \in X \wedge P_{\tilde{\sigma},\tau}(\pi'|h\tilde{u}) > 0\}|$. Let $\tilde{\tau}$ be a strategy which on history h picks $\tilde{u}$ and on histories prefixed with $h\tilde{u}$ follows $\tau_{\tilde{u}}$ (otherwise picking arbitrarily). Then

$$\begin{aligned}
W(X|h) &\leq |\{\pi \mid h\pi \in X \wedge P_{\tilde{\sigma},\tilde{\tau}}(\pi|h) > 0\}| \\
&= \sum_{u \in \Sigma} |\{\pi' \mid hu\pi' \in X \wedge \tau(h,u) \cdot P_{\tilde{\sigma},\tilde{\tau}}(\pi'|hu) > 0\}| \\
&= |\{\pi' \mid h\tilde{u}\pi' \in X \wedge P_{\tilde{\sigma},\tilde{\tau}}(\pi'|h\tilde{u}) > 0\}| \\
&= |\{\pi' \mid h\tilde{u}\pi' \in X \wedge P_{\tilde{\sigma},\tau_{\tilde{u}}}(\pi'|h\tilde{u}) > 0\}| \\
&\leq W(X|h\tilde{u}) \\
&= \min_{u \in \Sigma} W(X|hu).
\end{aligned}$$

□

Now we can state the realizability conditions, which are simply that I and A have sufficiently large width. In fact, the conditions turn out to be exactly the same as those for non-reactive CI (Theorem 3.1) with $\lambda = 0$, except that width takes the place of size.

Theorem 4.1. *The following are equivalent:*

(1) $\mathcal{C}$ is realizable.

(2) $W(I) \geq 1/\rho$ and $W(A) \geq (1-\epsilon)/\rho$.

(3) There is an improviser for $\mathcal{C}$.

Running Example. We saw above that our example was realizable with $\epsilon = \rho = 1/2$, and indeed $4 = W(I) \geq 1/\rho = 2$ and $1 = W(A) \geq (1-\epsilon)/\rho = 1$. However, if we put $\rho = 1/3$ we violate the second inequality and the instance is not realizable: essentially, we need to distribute probability $1 - \epsilon = 1/2$ among plays in A (to satisfy the soft constraint), but since $W(A) = 1$, against some adversaries we can only generate one play in A and would have to give it the whole $1/2$ (violating the randomness requirement).

The difficult part of the Theorem is constructing an improviser when the inequalities (2) hold. Despite the similarity in these conditions to the non-reactive case, the construction is much more involved. We begin with a general overview.

4.2.2 Constructing an Improviser: Overview

Recall that in Chapter 3 we built improvisers by uniformly sampling from the sets of admissible and inadmissible improvisations. Uniform sampling is also at the core of our improviser for RCI, which can be viewed as an extension of the classical random-walk reduction of uniform sampling to counting [184]. In this reduction (which also underlies the algorithms for sampling from DFAs and UCFGs used earlier [86, 118]), a uniform distribution over paths in a DAG is obtained by moving to the next vertex with probability proportional to the number of paths originating at it. In our case, which plays are possible depends on the adversary, but the width still tells us *how many* plays are possible. So we could try a random walk using widths as weights: e.g., on the first turn in Figure 4.3, picking $+$, $-$, and $=$ with probabilities $1/4$, $2/4$, and $1/4$ respectively. Against the adversary shown in Figure 4.3, this would indeed yield a uniform distribution over the four possible plays in I.

However, the soft constraint may require a non-uniform distribution. In the running example with $\epsilon = \rho = 1/2$, we need to generate the single possible play in A with probability $1/2$, not just the uniform probability $1/4$. This is easily fixed by doing the random walk with a *weighted average* of the widths of I and A: specifically, move to position h with probability proportional to $\alpha W(A|h) + \beta(W(I|h) - W(A|h))$. In the example, this would result in plays in A getting probability α and those in $I \setminus A$ getting probability β. Taking α sufficiently large, we can ensure the soft constraint is satisfied. This is again analogous to our construction for CI, where we used a biased coin to pick between uniformly sampling from A or $I \setminus A$.

Unfortunately, this strategy can fail if the adversary makes *more* plays available than the width guarantees. Consider the game on the left of Figure 4.4, where $W(I) = 3$ and $W(A) = 2$. This is realizable with $\epsilon = \rho = 1/3$, but no values of α and β yield improvising strategies, essentially because an adversary moving from X to Z breaks the worst-case assumption that the adversary will minimize the number of possible plays by moving to Y. In fact, this instance is realizable but not by any memoryless strategy. To see this, note that all such strategies can be parametrized by the probabilities p and q in Figure 4.4. To satisfy the randomness constraint against the adversary that moves from X to Y, both p and $(1-p)q$ must be at most $1/3$. To satisfy the soft constraint against the adversary that moves from X to Z we must have $pq + (1-p)q \geq 2/3$, so $q \geq 2/3$. But then $(1-p)q \geq (1-1/3)(2/3) = 4/9 > 1/3$, a contradiction.

Note that this result is in strong contrast to the situation for ordinary 2-player games: finite 2-player games with reachability and safety objectives always admit memoryless strategies, even if the game is stochastic [117, 54]. The fact that RCI cannot always be solved by memoryless strategies illustrates the fundamental difference between allowing the environment to be random, as in stochastic games, and requiring the strategy to be random, as in

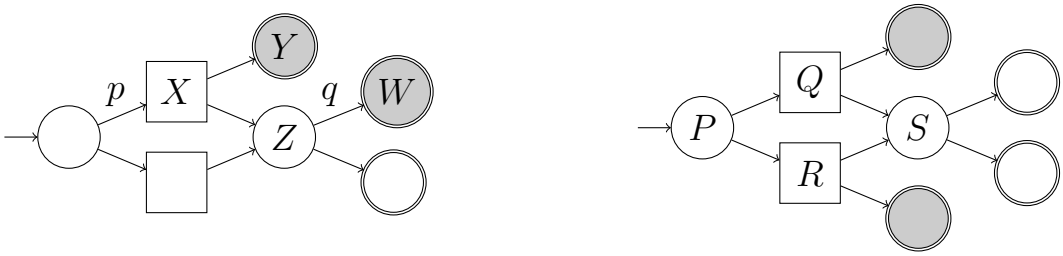


Figure 4.4: Reachability games where a naïve random walk, and all memoryless strategies, fail (left) and where no strategy can optimize either ϵ or ρ against every adversary simultaneously (right).

control improvisation. In fact, both the soft and randomness constraints in RCI are necessary for this phenomenon to occur: it is not hard to see that a simple random walk using $W(I|h)$ as weights as suggested above will satisfy the hard and randomness constraints, and any strategy for winning set A will satisfy the hard and soft constraints.

To fix the problem of the adversary possibly making more plays available than expected, our improvising strategy $\hat{\sigma}$ (which we will fully specify in Algorithm 4.1 below) takes a simplistic approach: it tracks how many plays in A and I are expected to be possible based on their widths, and if more are available it ignores them. For example, entering state Z from X in Figure 4.4, there are 2 ways to produce a play in I, but since $W(I|X) = 1$ we ignore the play in $I \setminus A$ and always move to state W. Extra plays in A are similarly ignored by being treated as members of $I \setminus A$. Ignoring unneeded plays may seem wasteful, but the proof of Theorem 4.1 will show that $\hat{\sigma}$ nevertheless achieves the best possible ϵ:

Corollary 4.1. *$\mathcal{C}$ is realizable if and only if $W(I) \geq 1/\rho$ and $\epsilon \geq \epsilon_{\text{opt}} \equiv \max(1 - \rho W(A), 0)$. Against any adversary, the error probability of Algorithm 4.1 is at most ϵ_{opt}.*

Thus, if *any* improviser can achieve an error probability ϵ, ours does. We could ask for a stronger property, namely that against each adversary the improviser achieves the smallest possible error probability *for that adversary*. Unfortunately, this is impossible in general. Consider the game on the right in Figure 4.4, with $\rho = 1$. Against the adversary which moves to state S we can never generate an admissible improvisation, so we cannot do better than $\epsilon = 1$ (agreeing with the Corollary, since $W(A) = 0$ and so $\epsilon_{\text{opt}} = 1$). However, we can do better against other adversaries: against the adversary which always moves up, we can achieve $\epsilon = 0$ with the strategy that at P moves to Q. We can also achieve $\epsilon = 0$ against the adversary that always moves down, but only with a *different* strategy, namely the one that at P moves to R. So there is no single strategy that achieves the optimal ϵ for every adversary. A similar argument shows that if we fix $\epsilon = 1$, there is also no strategy achieving the smallest possible ρ for every adversary. In essence, optimizing ϵ or ρ in every case would require the strategy to depend on the adversary.

4.2.3 Constructing an Improviser: Details

Our improvising strategy, as outlined in the previous section, is shown in Algorithm 4.1. We first compute α and β, the (maximum) probabilities for generating elements of A and $I \setminus A$ respectively. As in the CI improviser construction (Theorem 3.1), we take α as large as possible given $\alpha \leq \rho$, and determine β from the probability left over (modulo a couple corner cases).

Algorithm 4.1 The RCI improvising strategy $\hat{\sigma}$.

$\alpha \leftarrow \min(\rho, 1/W(A))$ (or 0 instead if $W(A) = 0$)
$\beta \leftarrow (1 - \alpha W(A))/(W(I) - W(A))$ (or 0 instead if $W(I) - W(A) = 0$)
$m^A \leftarrow W(A)$, $m^I \leftarrow W(I)$
$h \leftarrow \lambda$
while the game is not over after h **do**
    if it is our turn after h **then**
        $m^A_u, m^I_u \leftarrow$ PARTITION(m^A, m^I, h) ▷ returns values for each $u \in \Sigma$
        for each $u \in \Sigma$, put $t_u \leftarrow \alpha m^A_u + \beta(m^I_u - m^A_u)$
        pick $u \in \Sigma$ with probability proportional to t_u and append it to h
        $m^A \leftarrow m^A_u$, $m^I \leftarrow m^I_u$
    else
        the adversary picks $u \in \Sigma$ given the history h; append it to h
return h

Next we initialize m^A and m^I, our expectations for how many plays in A and I respectively are still possible to generate. Initially these are given by $W(A)$ and $W(I)$, but as we saw above it is possible for more plays to become available. The function PARTITION handles this, deciding which m^A (resp., m^I) out of the available $W(A|h)$ ($W(I|h)$) plays we will use. The behavior of PARTITION is defined by the following lemma; its proof greedily takes the first m^A possible plays in A under some canonical order and the first $m^I - m^A$ of the remaining plays in I.

Lemma 4.3. *If it is our turn after $h \in \Sigma^{\leq n}$, and $m^A, m^I \in \mathbb{Z}$ satisfy:*

- $0 \leq m^A \leq m^I \leq W(I|h)$ *and*
- $m^A \leq W(A|h)$,

there are integer partitions $\sum_{u\in\Sigma} m^A_u$ and $\sum_{u\in\Sigma} m^I_u$ of m^A and m^I respectively such that

- $0 \leq m^A_u \leq m^I_u \leq W(I|hu)$ *and*
- $m^A_u \leq W(A|hu)$

for all $u \in \Sigma$. Canonical such partitions are computable in polynomial time given oracles for $W(I|\cdot)$ and $W(A|\cdot)$.

Proof. Index the elements of Σ via some canonical order as $(u_j)_{0\le j<\ell}$ for some $\ell \ge 1$. We first construct the partition $\sum_{j<\ell} m_j^A$ of m^A. Find the greatest $k \le \ell$ such that $\sum_{j<k} W(A|hu_j) \le m^A$. This is well-defined, since if $k = 0$ then the sum is zero and the condition is satisfied. If $\sum_{j<k} W(A|hu_j) = m^A$, we put

$$m_j^A = \begin{cases} W(A|hu_j) & j < k \\ 0 & j \ge k. \end{cases}$$

If instead $\sum_{j<k} W(A|hu_j) < m^A$ we must have $k < \ell$, since $\sum_{j<\ell} W(A|hu_j) = \sum_{u\in\Sigma} W(A|hu) = W(A|h) \ge m^A$. Then by the definition of k we have $\sum_{j\le k} W(A|hu_j) > m^A$, so $W(A|hu_k) > m^A - \sum_{j<k} W(A|hu_j)$. Therefore we put

$$m_j^A = \begin{cases} W(A|hu_j) & j < k \\ m^A - \sum_{i<k} W(A|hu_i) & j = k \\ 0 & j > k. \end{cases}$$

Now we construct the partition $\sum_{j<\ell} m_j^I$ of m^I. We do this by partitioning the difference $m^I - m^A$ along the same lines as above, then adding back m_j^A to ensure $m_j^I \ge m_j^A$. Let $d_j = W(I|hu_j) - m_j^A$. Since $m_j^A \le W(A|hu_j) \le W(I|hu_j)$, we have $d_j \ge 0$. Find the greatest $k \le \ell$ such that $\sum_{j<k} d_j \le m^I - m^A$. This is well-defined since if $k = 0$ the sum is zero, and $m^I - m^A \ge 0$ by assumption. If $\sum_{j<k} d_j = m^I - m^A$ we put

$$m_j^I = \begin{cases} m_j^A + d_j & j < k \\ m_j^A & j \ge k. \end{cases}$$

This clearly satisfies $m_j^A \le m_j^I \le W(I|hu_j)$, and $\sum_{j<\ell} m_j^I = \sum_{j<k}(m_j^A + d_j) + \sum_{j\ge k} m_j^A = \sum_{j<\ell} m_j^A + \sum_{j<k} d_j = m^A + (m^I - m^A) = m^I$ as desired. If instead $\sum_{j<k} d_j < m^I - m^A$ we must have $k < \ell$, since $\sum_{j<\ell} d_j = \sum_{j<\ell}(W(I|hu_j) - m_j^A) = \sum_{u\in\Sigma} W(I|hu) - \sum_{j<\ell} m_j^A = W(I|h) - m^A \ge m^I - m^A$. Then by the definition of k we have $\sum_{j\le k} d_j > m^I - m^A$, so $d_k > m^I - m^A - \sum_{j<k} d_j$. Therefore we put

$$m_j^I = \begin{cases} m_j^A + d_j & j < k \\ m_k^A + (m^I - m^A - \sum_{i<k} d_i) & j = k \\ m_j^A & j > k. \end{cases}$$

Again this satisfies $m_j^A \le m_j^I \le W(I|hu_j)$, and $\sum_{j<\ell} m_j^I = \sum_{j<k}(m_j^A + d_j) + (m_k^A + (m^I - m^A - \sum_{j<k} d_j)) + \sum_{j>k} m_j^A = \sum_{j<\ell} m_j^A + (m^I - m^A) = m^A + (m^I - m^A) = m^I$ as desired.

These partitions are canonical since the values of k used in each construction are uniquely determined (and the ordering of Σ is fixed). Also, k may be found by a linear search from 0 up to ℓ, which has value at most $|\Sigma|$. The quantities $W(I|hu_j)$ all have polynomial bitwidth

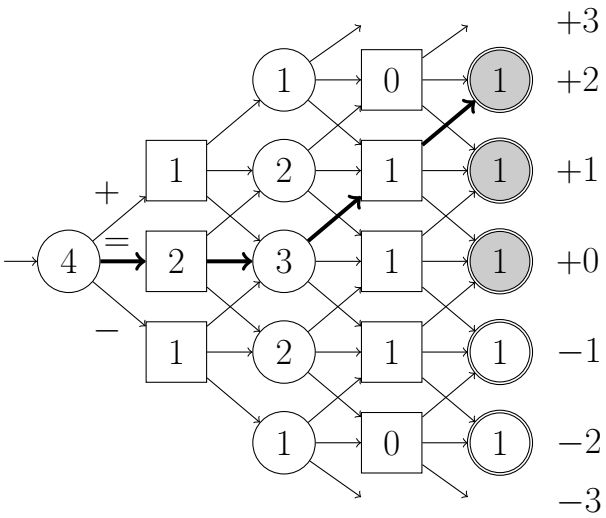


Figure 4.5: A run of Algorithm 4.1, labeling states with the corresponding widths of I (left) and A (right).

(they are bounded above by $|\Sigma|^n$), so the arithmetic above can be done in polynomial time. Therefore the total time needed to construct the partitions is polynomial relative to oracles for $W(I|\cdot)$ and $W(A|\cdot)$. □

Finally, we perform the random walk, moving from position h to hu with (unnormalized) probability t_u, the weighted average described above.

Running Example. With $\epsilon = \rho = 1/2$, as before $W(A) = 1$ and $W(I) = 4$ so $\alpha = 1/2$ and $\beta = 1/6$. On the first move, m^A and m^I match $W(A|h)$ and $W(I|h)$, so all plays are used and PARTITION returns $(W(A|hu), W(I|hu))$ for each $u \in \Sigma$. Looking up these values in Figure 4.5, we see $(m^A_=, m^I_=) = (0, 2)$ and so $t(=) = 2\beta = 1/3$. Similarly $t(+) = \alpha = 1/2$ and $t(-) = \beta = 1/6$. We choose an action according to these weights; suppose $=$, so that we update $m^A \leftarrow 0$ and $m^I \leftarrow 2$, and suppose the adversary responds with $=$. From Figure 4.5, $W(A|\ ==) = 1$ and $W(I|\ ==) = 3$, whereas $m^A = 0$ and $m^I = 2$. So PARTITION discards a play, say returning $(m^A_u, m^I_u) = (0, 1)$ for $u \in \{+, =\}$ and $(0, 0)$ for $u \in \{-\}$. Then $t(+) = t(=) = \beta = 1/6$ and $t(-) = 0$. So we pick $+$ or $=$ with equal probability, say $+$. If the adversary responds with $+$, we get the play $==++$, shown in bold on Figure 4.5. As desired, it satisfies the hard constraint.

The next few lemmas establish the properties of $\hat{\sigma}$ we need to prove Theorem 4.1. Throughout, we write $m^A(h)$ (respectively, $m^I(h)$) for the value of m^A (m^I) at the start of the iteration for history h. We also define $t(h) = \alpha m^A(h) + \beta(m^I(h) - m^A(h))$: note that on line (9) of Algorithm 4.1, the probability of picking u is then $t_u / \sum_v t_v = t(hu)/t(h)$.

We start by proving that $\hat{\sigma}$ is actually well-defined and satisfies the hard constraint. For the former, we show by induction on the history h that the conditions of Lemma 4.3 are always satisfied and that $t(h) > 0$ so we do not divide by zero. This also establishes the hard

constraint, since $t(h) > 0$ implies $m^I(h) > 0$, and when the game ends this can only happen if $h \in I$.

Lemma 4.4. *If $W(I) \geq 1/\rho$, then $\hat{\sigma}$ is a well-defined strategy and $P_{\hat{\sigma},\tau}(I) = 1$ for every adversary τ.*

Proof. First we show by induction on i that for all plays $h\pi \in \Pi_{\hat{\sigma},\tau}$ with $|h| = i$, we have:

- $0 \leq m^A(h) \leq m^I(h) \leq W(I|h)$ and $m^A(h) \leq W(A|h)$;
- $t(h), m^I(h) > 0$.

In the base case $i = 0$, we must have $h = \lambda$. Then $m^A(\lambda) = W(A) \geq 0$ and $m^I(\lambda) = W(I) \geq 1/\rho > 0$. To show $t(\lambda) = 1$, there are three cases. If $W(A) = 0$, then $\alpha = 0$ and $W(I) - W(A) = W(I) \geq 1/\rho > 0$, so $\beta = 1/W(I)$ and $t(\lambda) = \beta W(I) = 1$. If $W(I) - W(A) = 0$, then $\beta = 0$ and $W(A) = W(I) \geq 1/\rho$, so $\alpha = 1/W(A)$ and $t(\lambda) = \alpha W(A) = 1$. Otherwise $\alpha = \min(\rho, 1/W(A))$ and $\beta = (1 - \alpha W(A))/(W(I) - W(A))$, so $t(\lambda) = \alpha W(A) + (1 - \alpha W(A)) = 1$. Therefore we always have $t(\lambda) = 1$.

Now take any play $h\pi \in \Pi_{\hat{\sigma},\tau}$ with $|h| = i < n$ and suppose the hypothesis holds. If it is the adversary's turn after h, then if the adversary outputs $u \in \Sigma$ we have $m^A(h) = m^A(hu)$ and $m^I(h) = m^I(hu)$. So since $m^A(h) \leq W(A|h) = \min_{v\in\Sigma} W(A|hv) \leq W(A|hu)$ and $m^I(h) \leq W(I|h) = \min_{v\in\Sigma} W(I|hv) \leq W(I|hu)$, the hypothesis holds in the next step. If instead it is our turn after h and we output $u \in \Sigma$, then $m^A(hu)$ and $m^I(hu)$ are given by Lemma 4.3 and $0 \leq m^A(hu) \leq m^I(hu) \leq W(I|hu)$ and $m^A(hu) \leq W(A|hu)$ by construction. Furthermore $t(hu) > 0$, since if $t(hu) = 0$ then $\hat{\sigma}$ has probability zero to output u, a contradiction. This implies $m^I(hu) > 0$, since if $m^I(hu) = 0$ then $m^A(hu) = 0$ and so $t(hu) = 0$. Therefore by induction we always have $0 \leq m^A(h) \leq m^I(h) \leq W(I|h)$, $m^A(h) \leq W(A|h)$, and $t(h), m^I(h) > 0$.

Now for any history $h \in \Sigma^{\leq n}$ after which it is our turn, by construction the quantities $m^A(hu)$ and $m^I(hu)$ for $u \in \Sigma$ form partitions of $m^A(h)$ and $m^I(h)$ respectively. So $\sum_{u\in\Sigma} t(hu) = \sum_{u\in\Sigma} \alpha m^A(hu) + \beta(m^I(hu) - m^A(hu)) = \alpha m^A(h) + \beta(m^I(h) - m^A(h)) = t(h) > 0$. So $\hat{\sigma}(h, \cdot)$ is a probability distribution over Σ, and $\hat{\sigma}$ is a well-defined strategy.

Finally, take any play $\pi \in \Pi_{\hat{\sigma},\tau}$. As shown above we have $W(I|\pi) \geq m^I(\pi) > 0$, and since $|\pi| = n$ this implies $\pi \in I$. Therefore $P_{\hat{\sigma},\tau}(I) = 1$. □

Next, we show that because of the term $\alpha m^A(h)$ term in the weights $t(h)$, our strategy generates a member of A with probability at least $\alpha W(A)$.

Lemma 4.5. *If $W(I) \geq 1/\rho$, then $P_{\hat{\sigma},\tau}(A) \geq \min(\rho W(A), 1)$ for every adversary τ.*

Proof. We prove by induction on i in decreasing order that for all plays $h\pi \in \Pi_{\hat{\sigma},\tau}$ with $|h| = i$, $\sum_{\rho \,|\, h\rho\in A} P_{\hat{\sigma},\tau}(\rho|h) \geq \alpha m^A(h)/t(h)$. In the base case $i = n$, by Lemma 4.4 we must have $h \in I$, so $m^A(h) \leq m^I(h) \leq W(I|h) = 1$. If $m^A(h) = 0$ the hypothesis holds trivially. Otherwise $m^A(h) = 1$, so $t(h) = \alpha$ and since $m^A(h) \leq W(A|h)$ we must have

$h \in A$. Therefore, letting $\rho = \lambda$ we have $h\rho \in A$, so $P_{\hat{\sigma},\tau}(\rho|h) = 1 = \alpha m^A(h)/t(h)$ and the hypothesis again holds.

Now take any play $h\pi \in \Pi_{\hat{\sigma},\tau}$ with $|h| = i < n$. If $h \in I$ then the hypothesis holds as above. Otherwise if it is the adversary's turn after h, then $m^A(h) = m^A(hu)$, $m^I(h) = m^I(hu)$, and $t(h) = t(hu)$ for any $u \in \Sigma$. By hypothesis, for any play $hu\pi' \in \Pi_{\hat{\sigma},\tau}$ we have $\sum_{\rho \mid hu\rho \in A} P_{\hat{\sigma},\tau}(\rho|hu) \geq \alpha m^A(hu)/t(hu) = \alpha m^A(h)/t(h)$. So

$$\begin{aligned}
\sum_{\rho \mid h\rho \in A} P_{\hat{\sigma},\tau}(\rho|h) &= \sum_{u \in \Sigma} \sum_{\rho' \mid hu\rho' \in A} \tau(h,u) \cdot P_{\hat{\sigma},\tau}(\rho'|hu) \\
&= \sum_{u \in \Sigma} \tau(h,u) \sum_{\rho' \mid hu\rho' \in A} P_{\hat{\sigma},\tau}(\rho'|hu) \\
&\geq \sum_{u \in \Sigma} \tau(h,u) \cdot \frac{\alpha m^A(h)}{t(h)} \\
&= \frac{\alpha m^A(h)}{t(h)} \sum_{u \in \Sigma} \tau(h,u) \\
&= \frac{\alpha m^A(h)}{t(h)}
\end{aligned}$$

as desired. If instead it is our turn after h, then if we output $u \in \Sigma$ we update m^A to m^A_u, so $m^A(hu) = m^A_u(h)$. Then by hypothesis we have $\sum_{\rho \mid hu\rho \in A} P_{\hat{\sigma},\tau}(\rho|hu) \geq \alpha m^A(hu)/t(hu) = \alpha m^A_u(h)/t(hu)$. So

$$\begin{aligned}
\sum_{\rho \mid h\rho \in A} P_{\hat{\sigma},\tau}(\rho|h) &= \sum_{u \in \Sigma} \sum_{\rho' \mid hu\rho' \in A} \hat{\sigma}(h,u) \cdot P_{\hat{\sigma},\tau}(\rho'|hu) \\
&= \sum_{u \in \Sigma} \hat{\sigma}(h,u) \sum_{\rho' \mid hu\rho' \in A} P_{\hat{\sigma},\tau}(\rho'|hu) \\
&\geq \sum_{u \in \Sigma} \hat{\sigma}(h,u) \cdot \frac{\alpha m^A_u(h)}{t(hu)} \\
&= \alpha \sum_{u \in \Sigma} \frac{t(hu)}{t(h)} \cdot \frac{m^A_u(h)}{t(hu)} \\
&= \frac{\alpha}{t(h)} \sum_{u \in \Sigma} m^A_u(h) \\
&= \frac{\alpha m^A(h)}{t(h)}
\end{aligned}$$

again as desired. Therefore by induction this holds for every i, and in particular for $i = 0$.

Since every play $\pi \in \Pi_{\hat{\sigma},\tau}$ is of the form $\lambda\pi$, noting that $t(\lambda) = 1$ as shown in Lemma 4.4 we have $P_{\hat{\sigma},\tau}(A) = \sum_{\pi \mid \lambda\pi \in A} P_{\hat{\sigma},\tau}(\pi|\lambda) \geq \alpha m^A(\lambda)/t(\lambda) = \alpha W(A) = \min(\rho W(A), 1)$. □

Finally, we show that our strategy never generates an individual trace with probability greater than ρ. This essentially follows from the fact that if the adversary is deterministic, the weights of our random walk yield a distribution where each play π has probability either α or β (depending on whether $m^A(\pi) = 1$ or 0), and these are both at most ρ. If the adversary instead assigns nonzero probability to multiple actions, this only decreases the probability of individual plays.

Lemma 4.6. *If $W(I) \geq 1/\rho$, then $P_{\hat{\sigma},\tau}(\pi) \leq \rho$ for every $\pi \in \Sigma^n$ and adversary τ.*

Proof. We prove by induction on i in decreasing order that for all plays $h\pi \in \Pi_{\hat{\sigma},\tau}$ with $|h| = i$, $P_{\hat{\sigma},\tau}(\pi|h) \leq \max(\alpha, \beta)/t(h)$. In the base case $i = n$, by Lemma 4.4 we must have $h \in I$. Then $m^I(h) = 1$, since $m^I(h) \leq W(I|h) = 1$ and $m^I(h) > 0$ if h can be generated by $\hat{\sigma}$ (as shown in Lemma 4.4). So $t(h) = \alpha m^A(h) + \beta(1 - m^A(h))$, and thus either $t(h) = \alpha$ or $t(h) = \beta$ (depending on whether $m^A(h) = 1$ or $m^A(h) = 0$). In either case $\max(\alpha, \beta)/t(h) \geq 1$, so $P_{\hat{\sigma},\tau}(\pi|h) \leq \max(\alpha, \beta)/t(h)$ as desired.

Now take any play $h\pi \in \Pi_{\hat{\sigma},\tau}$ with $|h| = i < n$. Since $|h| < n$ the play is of the form $hu\pi'$ for some $u \in \Sigma$, and by hypothesis $P_{\hat{\sigma},\tau}(\pi'|hu) \leq \max(\alpha, \beta)/t(hu)$. Now if it is the adversary's turn after h, then $m^A(hu) = m^A(h)$ and $m^I(hu) = m^I(h)$, so $t(hu) = t(h)$ and therefore $P_{\hat{\sigma},\tau}(\pi|h) = \tau(h, u) \cdot P_{\hat{\sigma},\tau}(\pi'|hu) \leq P_{\hat{\sigma},\tau}(\pi'|hu) \leq \max(\alpha, \beta)/t(hu) = \max(\alpha, \beta)/t(h)$ as desired. If instead it is our turn after h, then

$$P_{\hat{\sigma},\tau}(\pi|h) = \hat{\sigma}(h, u) \cdot P_{\hat{\sigma},\tau}(\pi'|hu) = \frac{t(hu)}{t(h)} \cdot P_{\hat{\sigma},\tau}(\pi'|hu) \leq \frac{t(hu)}{t(h)} \cdot \frac{\max(\alpha, \beta)}{t(hu)} = \frac{\max(\alpha, \beta)}{t(h)}$$

again as desired. So by induction the hypothesis holds for every $i \in \{0, \ldots, n\}$, and in particular for $i = 0$.

Since every play $\pi \in \Pi_{\hat{\sigma},\tau}$ is of the form $\lambda\pi$, we have $P_{\hat{\sigma},\tau}(\pi) = P_{\hat{\sigma},\tau}(\pi|\lambda) \leq \max(\alpha, \beta)/t(\lambda) = \max(\alpha, \beta)$ (as $t(\lambda) = 1$). Recall that $\alpha = \min(\rho, 1/W(A)) \leq \rho$ and $\beta = (1-\alpha W(A))/(W(I)-W(A))$, with the convention that $\alpha = 0$ if $W(A) = 0$ and $\beta = 0$ if $W(I) - W(A) = 0$. If $\alpha = \rho$, then $\beta = (1 - \rho W(A))/(W(I) - W(A)) \leq (1 - \rho W(A))/((1/\rho) - W(A)) = \rho$. If instead $\alpha = 1/W(A)$, then $\beta = 0$. Finally, if $\alpha = 0$ then $W(A) = 0$ and $\beta = 1/W(I) \leq \rho$. So $\max(\alpha, \beta) \leq \rho$, and therefore $P_{\hat{\sigma},\tau}(\pi) \leq \rho$ for every $\pi \in \Pi_{\hat{\sigma},\tau}$. In fact this holds for all plays $\pi \in \Sigma^n$, since if $\pi \notin \Pi_{\hat{\sigma},\tau}$ then $P_{\hat{\sigma},\tau}(\pi) = 0$. □

Now we have all the pieces ready to prove the realizability conditions.

Proof of Theorem 4.1. We use a similar argument to that of Theorem 3.1.

(1)⇒(2) Suppose σ is an improvising strategy, and fix any adversary τ. Then $\rho|\Pi_{\sigma,\tau} \cap I| = \sum_{\pi \in \Pi_{\sigma,\tau} \cap I} \rho \geq \sum_{\pi \in I} P_{\sigma,\tau}(\pi) = P_{\sigma,\tau}(I) = 1$, so $|\Pi_{\sigma,\tau} \cap I| \geq 1/\rho$. Since τ is arbitrary, this implies $W(I) \geq 1/\rho$. Since $A \subseteq I$, we also have $\rho|\Pi_{\sigma,\tau} \cap A| = \sum_{\pi \in \Pi_{\sigma,\tau} \cap A} \rho \geq \sum_{\pi \in A} P_{\sigma,\tau}(\pi) = P_{\sigma,\tau}(A) \geq 1 - \epsilon$, so $|\Pi_{\sigma,\tau} \cap A| \geq (1 - \epsilon)/\rho$ and thus $W(A) \geq (1 - \epsilon)/\rho$.

(2)⇒(3) By Lemmas 4.4 and 4.6, $\hat{\sigma}$ is well-defined and satisfies the hard and randomness constraints. By Lemma 4.5, $P_{\hat{\sigma},\tau}(A) \geq \min(\rho W(A), 1) \geq 1 - \epsilon$, so $\hat{\sigma}$ also satisfies the soft constraint and thus is an improvising strategy. Its transition probabilities are rational, so it can be implemented by an expected finite-time probabilistic algorithm, which is then an improviser for $\mathcal{C}$.

(3)⇒(1) Immediate. □

Proof of Corollary 4.1. The inequalities in the statement are equivalent to those of Theorem 4.1(2). By Lemma 4.5, we have $P_{\hat{\sigma},\tau}(A) \geq \min(\rho W(A), 1)$. So the error probability is at most $1 - \min(\rho W(A), 1) = \epsilon_{\text{opt}}$. □

4.3 A Generic Improvisation Scheme

As we did for CI, we can now use our improviser construction to develop a generic improvisation scheme usable with any class of specifications Spec supporting a certain set of operations:

Intersection: Given specs $\mathcal{X}$ and $\mathcal{Y}$, find $\mathcal{Z}$ such that $L(\mathcal{Z}) = L(\mathcal{X}) \cap L(\mathcal{Y})$.

Width Measurement: Given a specification $\mathcal{X}$, a length $n \in \mathbb{N}$ in unary, and a history $h \in \Sigma^{\leq n}$, compute $W(X|h)$ where $X = L(\mathcal{X}) \cap \Sigma^n$.

Recall that our CI scheme, Theorem 3.2, used *intersection*, *difference*, *counting*, and *uniform sampling*. Here, because we need to generate a word in an online manner, we need a more general version of counting: width measurement is essentially the adversarial version of counting the strings in a specification which extend a given prefix. However, given this operation we can dispense with uniform sampling, reducing it to counting partial solutions and performing a random walk, as in Algorithm 4.1. This approach also eliminates the need for the difference operation.

Theorem 4.2. *If the operations on* Spec *above take polynomial time (respectively, space), then* RCI(Spec, Spec) *has a polynomial-time (space) improvisation scheme.*

Proof. Given an instance $\mathcal{C} = (\mathcal{H}, \mathcal{S}, n, \epsilon, \rho)$ in RCI(Spec, Spec), we first apply intersection to $\mathcal{H}$ and $\mathcal{S}$ to obtain $\mathcal{A} \in$ Spec such that $L(\mathcal{A}) \cap \Sigma^n = A$. Since intersection takes polynomial time (space), $\mathcal{A}$ has size polynomial in $|\mathcal{C}|$. Next we use width measurement to compute $W(I) = W(L(\mathcal{H}) \cap \Sigma^n|\lambda)$ and $W(A) = W(L(\mathcal{A}) \cap \Sigma^n|\lambda)$. If these violate the inequalities in Theorem 4.1, then $\mathcal{C}$ is not realizable and we return $\perp$. Otherwise $\mathcal{C}$ is realizable, and $\hat{\sigma}$ in Algorithm 4.1 is an improvising strategy. Furthermore, we can construct an expected finite-time probabilistic algorithm implementing $\hat{\sigma}$, using width measurement to instantiate the oracles needed by Lemma 4.3. Determining $m^A(h)$ and $m^I(h)$ takes $O(n)$

invocations of PARTITION, each of which is poly-time relative to the width measurements[1]. These take time (space) polynomial in $|\mathcal{C}|$, since $\mathcal{H}$ and $\mathcal{A}$ have size polynomial in $|\mathcal{C}|$. As $m^A, m^I \leq |\Sigma|^n$, they have polynomial bitwidth and so the arithmetic required to compute t_u for each $u \in \Sigma$ takes polynomial time. Therefore the total expected runtime (space) of the improviser is polynomial. □

Remark. Note that as a byproduct of testing the inequalities in Theorem 4.1, our algorithm can compute the best possible error probability ϵ_{opt} given $\mathcal{H}$, $\mathcal{S}$, and ρ (see Corollary 4.1). Alternatively, given ϵ, we can compute the best possible ρ.

Recall that the complexities of a very wide range of CI problems lie between P and #P, due to the close connection between CI and counting problems (Theorem 3.3). A similar phenomenon occurs for RCI, except that its nature as a bounded 2-player game makes its natural complexity class PSPACE rather than #P. Specifically, there is a polynomial-space improvisation scheme for any specifications which can be tested in polynomial space:

Theorem 4.3. RCI(PSA, PSA) *has a polynomial-space improvisation scheme, where* PSA *is the class of polynomial-space decision algorithms.*

Proof. We implement the operations required by Theorem 4.2. Intersection is simple: we run the algorithms $\mathcal{X}$ and $\mathcal{Y}$ and accept if and only if both do. The resulting algorithm runs in polynomial space, since $\mathcal{X}$ and $\mathcal{Y}$ do, and can be constructed in polynomial time.

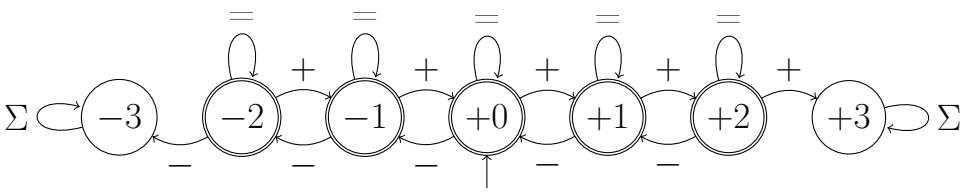
For width measurement, we compute $W(X|h)$ using an arithmetization of the usual PSPACE algorithm for QBF, replacing $\vee$ by $+$ at $\exists$ nodes in the recursive tree and $\wedge$ by min at $\forall$ nodes. Specifically, if it is our turn after h (corresponding to an $\exists$ quantifier in a QBF), we recursively compute $W(X|hu)$ for each $u \in \Sigma$ and return $\sum_{u\in\Sigma} W(X|hu) = W(X|h)$. If instead it is the adversary's turn, we again recursively compute $W(X|hu)$ for each $u \in \Sigma$ but now return $\min_{u\in\Sigma} W(X|hu) = W(X|h)$. Finally, in the base case $|h| = n$ we have $W(X|h) = \mathbf{1}_{h\in X}$ and so simply invoke $\mathcal{X}$ to determine in polynomial space whether $h \in X = L(\mathcal{X}) \cap \Sigma^n$. As in the QBF algorithm, the recursive tree has polynomial depth, and since $W(X|h) \leq |\Sigma|^n$ we need only polynomial space to remember partial results along the current path through the tree. So we can compute $W(X|h)$ in polynomial space. □

4.4 Complexity of Reactive Control Improvisation

Next, we study the complexity of RCI for different types of specifications. Since RCI is a strict generalization of CI[2], we first examine RCI for all the types of specifications we studied in Chapter 3. We find that for DFAs, the complexity remains polynomial-time when adding

[1] In practice, we would not recompute $m^A(h)$ and $m^I(h)$ each iteration but update them with a single call to PARTITION, as shown in Algorithm 4.1.

[2] We can encode a CI problem by having our specifications encode the assumption that the adversary always outputs a constant symbol, so that there is a one-to-one correspondence between improviser behaviors and plays of the 2-player game.



$i=4:$	0	1	1	1	1	1	0
$i=3:$	0	0	1	1	1	0	0
$i=2:$		1	2	3	2	1	
$i=1:$			1	2	1		
$i=0:$				4			

Figure 4.6: The hard specification DFA $\mathcal{H}$ in our running example, showing how $W(I|h)$ is computed.

reactivity, while for NFAs, CFGs, and Boolean formulas the complexity increases from #P to PSPACE. We also consider two additional types of specifications popular in reactive synthesis, namely safety/reachability games and temporal logic formulas. The former admit a polynomial-time improvisation scheme, being reducible to DFAs, while temporal logic is at least as expressive as Boolean formulas and therefore we can only give a polynomial-space scheme.

4.4.1 Finite Automata and Safety/Reachability Games

Now we develop a polynomial-time improvisation scheme for RCI instances with DFA specifications. This also provides a scheme for reachability/safety games, whose winning conditions can be straightforwardly encoded as DFAs.

Suppose D is a DFA with states V, accepting states T, and transition function $\delta : V \times \Sigma \to V$. Our scheme is based on the fact that $W(L(D)|h)$ depends only on the state of D reached on input h, allowing these widths to be computed by dynamic programming (in a similar way to the DFA counting algorithm [86]). Specifically, for all $v \in V$ and $i \in \{0, \ldots, n\}$ we define:

$$C(v,i) = \begin{cases} \mathbf{1}_{v \in T} & i = n \\ \min_{u \in \Sigma} \; C(\delta(v,u), i+1) & i < n \wedge i \text{ odd} \\ \sum_{u \in \Sigma} \; C(\delta(v,u), i+1) & \text{otherwise.} \end{cases}$$

Running Example. Figure 4.6 shows the values $C(v,i)$ in rows from $i = n$ downward. For example, $i = 2$ is our turn, so $C(1,2) = C(0,3) + C(1,3) + C(2,3) = 1 + 1 + 0 = 2$, while $i = 3$ is the adversary's turn, so $C(-3,3) = \min\{C(-3,4)\} = \min\{0\} = 0$. Note that the values in Figure 4.6 agree with the widths $W(I|h)$ shown in Figure 4.5.

Lemma 4.7. *For any history $h \in \Sigma^{\leq n}$, writing $X = L(D) \cap \Sigma^n$ we have $W(X|h) = C(D(h), |h|)$, where $D(h)$ is the state reached by running D on h.*

Proof. We prove this by induction on $i = |h|$ in decreasing order. In the base case $i = n$, we have $W(X|h) = \mathbf{1}_{h \in X} = \mathbf{1}_{D(h) \in T} = C(D(h), n)$. Now take any history $h \in \Sigma^{\leq n}$ with $|h| = i < n$. By hypothesis, for any $u \in \Sigma$ we have $W(X|hu) = C(D(hu), i+1)$. If it is our turn after h, then $W(X|h) = \sum_{u \in \Sigma} W(X|hu) = \sum_{u \in \Sigma} C(D(hu), i+1) = C(D(h), i)$ as desired. If instead it is the adversary's turn after h, then $W(X|h) = \min_{u \in \Sigma} W(X|hu) = \min_{u \in \Sigma} C(D(hu), i+1) = C(D(h), i)$ again as desired. So by induction the hypothesis holds for any i. □

Theorem 4.4. RCI (DFA, DFA) *has a polynomial-time improvisation scheme.*

Proof. We implement Theorem 4.2. Intersection can be done with the standard product construction. For width measurement we compute the quantities $C(v, i)$ by dynamic programming (from $i = n$ down to $i = 0$) and apply Lemma 4.7. □

As in the case of CI, the RCI problem becomes much more difficult for nondeterministic automata. The dynamic programming approach above fails for the same reason as before, namely that accepting paths no longer correspond one-to-one with words in the language. In fact, an argument quite similar to the one we used for the #P-hardness result for CI with NFAs shows that even finding a single winning strategy is PSPACE-hard:

Theorem 4.5. *Finite-window reactive synthesis for* NFA*s is PSPACE-hard.*

Proof. We show hardness by reduction from QBF. Given a QBF ϕ with variables numbered $1, \ldots, n$, without loss of generality we may assume the quantifiers strictly alternate starting with $\exists$ and that the matrix ψ is in disjunctive normal form (by negating the formula as needed). We can view an assignment to the variables as a word in $\{0,1\}^n$, where each element indicates the truth of the variable corresponding to its position. Now using the method of Kannan et al. [93], we can construct in polynomial time an NFA N which accepts exactly the satisfying assignments of ψ. Then we have a winning strategy to generate a play in $L(N)$ if and only if ϕ is true. □

Corollary 4.2. RCI (NFA, Σ^*) *and* RCI (Σ^*, NFA) *are PSPACE-hard.*

4.4.2 Context-Free Grammars and Boolean Formulas

To compare the complexity of RCI to that of CI for all types of specifications we considered in Chapter 3, we also consider specifications given by grammars or Boolean formulas. While grammars are not usually used in reactive synthesis, they are potentially useful in *randomized* reactive synthesis for applications like fuzz testing: the set of allowed messages exchanged between a client and server following a protocol with recursive structure can conveniently be expressed using a CFG [165].

Since NFAs can be converted to CFGs in polynomial time, Theorem 4.5 immediately shows that RCI with CFG specifications is PSPACE-hard. The same is true for Boolean formula specifications (and the more general symbolic specifications SYMB we defined in Section 3.4.3), since the problem of finding a single winning strategy is just QBF, which is PSPACE-hard. However, our generic polynomial-space scheme applies to both types of specifications:

Theorem 4.6. RCI (CFG, CFG) *and* RCI (SYMB, SYMB) *have polynomial-space improvisation schemes.*

Proof. By Theorem 4.3, since CFG parsing can be done in polynomial time and quantified Boolean formulas (including symbolic specifications) can be checked in polynomial space. □

4.4.3 Temporal Logic Formulas

Finally, we consider reactive control improvisation with specifications given by formulas of linear temporal logic (LTL) [135], a popular formalism for reactive synthesis, and the more expressive linear dynamic logic (LDL) [41]. For either logic we use its natural semantics on finite words (see De Giacomo and Vardi [41] for details).

For LTL specifications, RCI is PSPACE-hard because this is already true of ordinary reactive synthesis in a finite window[3].

Theorem 4.7. *Finite-window reactive synthesis for* LTL *is PSPACE-hard.*

Proof. We show hardness by reduction from QBF, along similar lines to Theorem 4.5. Given a QBF ϕ with variables numbered $1, \ldots, n$, without loss of generality we may assume the quantifiers strictly alternate starting with $\exists$ and that the matrix is in conjunctive normal form, consisting of clauses $c_1, \ldots, c_m$. We can view an assignment to the variables as a length-n trace over a single proposition p indicating the truth of the variable corresponding to the position. Then for each clause c_i we can construct an LTL formula ψ_c whose models of length n are exactly the assignments satisfying the clause. Specifically, if c_i contains the variables V^+ positively and V^- negatively, we put

$$\psi_c = \bigvee_{v \in V^+} X^{v-1} p \vee \bigvee_{v \in V^-} X^{v-1}(\neg p).$$

Then putting $\psi = \bigwedge_i \psi_{c_i}$, the length-$n$ models of ψ are exactly the assignments satisfying the matrix of ϕ. So we have a winning strategy to generate a play satisfying ψ if and only if ϕ is true. Finally, this construction clearly can be done in polynomial time. □

Corollary 4.3. RCI (LTL, Σ^*) *and* RCI (Σ^*, LTL) *are PSPACE-hard.*

[3] We suspect this has been observed but could not find a proof in the literature. See Torfah and Zimmermann [173] for a related type of result.

Table 4.1: Complexity of the reactive control improvisation problem for various types of hard and soft specifications $\mathcal{H}$ and $\mathcal{S}$. Here PSPACE indicates that checking realizability is PSPACE-hard, and that there is a polynomial-space improvisation scheme.

$\mathcal{H}\backslash\mathcal{S}$	**RSG**	**DFA**	**NFA**	**CFG**	**LTL**	**LDL**
RSG	poly-time		PSPACE			
DFA						
NFA						
CFG						
LTL						
LDL						

This is perhaps disappointing, but is an inevitable consequence of LTL subsuming Boolean formulas. On the other hand, our general polynomial-space scheme applies to LTL and its much more expressive generalization LDL:

Theorem 4.8. RCI (LDL, LDL) *has a polynomial-space improvisation scheme.*

Proof. This follows from Theorem 4.3, since satisfaction of an LDL formula by a finite word can be checked in polynomial time (e.g. by combining dynamic programming on subformulas with a regular expression parser). □

Thus for temporal logics polynomial-time algorithms are unlikely, but adding randomization to reactive synthesis does not increase its complexity.

4.5 Summary and Future Work

In this chapter we introduced *reactive control improvisation* as a framework for modeling reactive synthesis problems where random but controlled behavior is desired. RCI provides a natural way to tune the amount of randomness while ensuring that safety or other constraints remain satisfied. We showed that RCI problems can be efficiently solved in many cases occurring in practice, giving a polynomial-time improvisation scheme for reachability/safety or DFA specifications. We also showed that RCI problems with specifications in LTL or LDL, popularly used in planning, have the PSPACE-hardness typical of bounded games, and gave a matching polynomial-space improvisation scheme. This scheme generalizes to any specification checkable in polynomial space, including NFAs, CFGs, and many more expressive formalisms. Table 4.1 summarizes these results.

These results show that, at a high level, finding a maximally-randomized strategy using RCI is no harder than finding any winning strategy at all: for specifications yielding games solvable in polynomial time (respectively, space), we gave polynomial-time (space) improvisation schemes. We therefore hope that in applications where ordinary reactive synthesis

has proved tractable, our notion of randomized reactive synthesis will also. In particular, we expect our DFA scheme to be quite practical, and have experimented with applications in robotic planning (see Chapter 6). In addition to pursuing such applications, there are a number of interesting directions for further theoretical work:

Using Constraint Solvers. We saw in Chapter 3 that SAT solvers can be helpful in approximately solving difficult CI problems. In the reactive, 2-player game setting, the natural analogs of such algorithms are QBF solvers. Thus, it would be very interesting to see if QBF solvers can be extended to solve the kind of counting problems needed for RCI with Boolean or temporal logic formula specifications. Although this would not improve the theoretical complexity, since we already gave a polynomial-space improvisation scheme for such problems, it could provide schemes which are useful in practice. In fact, such symbolic methods could be helpful even for DFA specifications, since our automata-theoretic algorithm suffers from the problem (which we saw earlier in Chapter 3) that conjoining many simple properties can lead to exponentially-large automata.

Multiple Soft Constraints. We did not attempt to extend RCI with multiple soft constraints, as we did for CI in Chapter 3. Although the complexity of RCI is likely greater than that of CI in some cases (modulo complexity theory hypotheses), our PSPACE upper bounds are significantly lower than the EXP upper bound for multi-constraint CI from Theorem 3.11. So it is possible that adding reactivity to multi-constraint CI does not in fact change its complexity.

Infinite Words. A more interesting theoretical extension of RCI would be to allow unbounded or infinite words, as typically used in reactive synthesis. These extensions would be useful in robotic planning, as we discuss further in Chapter 6, as well as in other applications. However, it is unclear how best to adapt our randomness constraint to settings where the improviser can generate infinitely many words. In such settings the improviser could assign arbitrarily small or even zero probability to every word, rendering the randomness constraint trivial.

Generalized Randomness Constraints. Even in the bounded case, RCI extensions with more complex randomness constraints than a simple upper bound on individual word probabilities would be worthy of study. One possibility would be to more directly control diversity and/or unpredictability by requiring the distribution of the improviser's output to be close to uniform after transformation by a given function. This would allow, for example, requiring the order in which a set of designated locations is visited by a patrolling robot to be close to uniform, even if there are many more feasible routes for one order over another.

More General Games. Finally, we used the simplest type of 2-player game, where the underlying dynamics are deterministic and fully-observable. Extending RCI to stochastic

games and games of partial information would allow more accurate modeling of a variety of robotic planning problems where the results of actions and the state of the environment are uncertain but not necessarily adversarial.

Chapter 5

Language-Based Improvisation

5.1 Introduction

In Chapters 3 and 4, we focused on specification formalisms from logic, like Boolean and temporal logic formulas, or from formal language theory, like finite automata and context-free grammars. In this chapter, we consider a very different type of formalism: a *programming language*. At first it might seem that a program might be too low-level to be a convenient specification, but in fact there are a number of applications where this is quite useful. For example, in superoptimization [116, 154], one seeks to synthesize a program that is functionally equivalent to a reference program, but faster.

An example closer to algorithmic improvisation is that of *property-based testing*, popularized by the QuickCheck tool [34]. Here, one writes programs testing various desired properties of a system, as well as generator programs producing random inputs on which to run the system. For example, to test an API for manipulating heaps we could write a generator producing random heaps, and a program checking that the heap property is maintained after inserting a random value. These programs form a specification in that they define *which tests* are allowed: we should only invoke the API on valid heaps, for instance, and we should only check the heap property in between API calls.

This basic idea — using a program to encode a class of tests — has been used in a number of different forms. *Parametrized unit tests* [169] as used in Pex [168] are quite similar to property-based testing: one writes unit tests which should succeed for all possible values of some unspecified parameters; each assignment to the parameters yields a test case. Another example is CONCURRIT [52], a domain-specific language for describing thread schedules for testing a concurrent program. In all of these cases, a nondeterministic program defines a class of tests, which is then explored by some kind of search method: symbolic execution in Pex, random sampling in QuickCheck, and configurable random/systematic exploration in CONCURRIT.

Algorithmic improvisation with program specifications offers a potential way to generalize these techniques, at least for those based on random testing. However, the theory we studied

in Chapter 3 is too restrictive in that it does not give any detailed control over the distribution of an improviser. While near-uniform distributions are sensible if we know nothing about the input space (after restricting it with hard constraints), often we want highly non-uniform distributions. An extreme example is that of training machine learning algorithms: if we are generating synthetic training images, for example, its distribution must be close to that of the images the algorithm will actually be used on. For such applications, we want to allow complete control over the distribution of the generated data, while still being able to impose hard and soft constraints declaratively. This suggests that we consider another foundation for improvising from programs: *probabilistic programming languages*.

Probabilistic programming languages (PPLs) augment ordinary programming languages with probabilistic constructs, so that a single program defines a distribution over its outputs [79]. A large variety of PPLs have been developed, including imperative languages like PROB [79], functional languages like Church [77], and declarative languages like BLOG [123]. Although all PPLs define generative processes, where running a program yields a sample from its distribution, many PPLs also allow *conditioning* this distribution. Such languages have an *observe* statement which conditions the distribution on a given predicate, eliminating all executions of the program where the predicate does not hold [134, 79]. This is exactly what we need to implement the hard and soft constraints in algorithmic improvisation.

To our knowledge, probabilistic programming languages have not previously been used for test generation, probably because the design goals for PPLs have typically been to maximize expressivity and provide a general-purpose tool for Bayesian inference[1]. However, we argue that a *domain-specific* PPL can be a highly flexible and practical way to specify tests for a particular domain. Here, we focus on systems like autonomous cars and robots, whose environment is a *scene*, a configuration of physical objects and agents. Scenes are a complex and heterogeneous domain, with intricate geometric constraints: a typical traffic scene, like that in Figure 5.1, for example, has significant randomness but also significant structure. This makes scenes an ideal candidate for a domain where a DSL can dramatically decrease the effort required to specify and generate useful tests.

In this chapter, we design a domain-specific PPL, SCENIC, for defining distributions over scenes, and study *scene improvisation*, the problem of generating scenes from a SCENIC program [64, 65]. Scene improvisation has a number of applications, including not only testing but also training and debugging cyber-physical systems using synthetic data. We will discuss these applications in depth in Chapter 8, where we use SCENIC to analyze and improve the performance of a practical deep neural network for autonomous driving beyond what is achieved by state-of-the-art synthetic data generation methods. More generally, SCENIC can be used to give a formal specification of the distribution of environments under which a system is expected to operate correctly with high probability. Such environment models are essential for any formal analysis: in particular, any attempt to prove the correctness of a system requires a precise definition of the assumptions it makes about its environment. For

[1] Although PPLs are more commonly used for inference, they have been used for generation in computer graphics. See Chapter 8 for a discussion.

Figure 5.1: A scene of bumper-to-bumper traffic, generated from a ∼20-line SCENIC program and rendered in the video game Grand Theft Auto V [68].

this reason, SCENIC is a crucial component of VERIFAI, a toolkit for the design and analysis of AI-based systems [47], which uses SCENIC as its environment modeling language.

We will begin in Section 5.2 with an overview of SCENIC, highlighting its major features and motivating various choices in its design. Section 5.3 describes the syntax of the language in detail, illustrating the semantics of its constructs with examples, while Section 5.4 gives a formal operational semantics. Section 5.5 discusses the scene improvisation problem, and our specialized sampling algorithms for SCENIC. Finally, we conclude in Section 5.6 with a summary and directions for future work.

5.2 The Design of SCENIC

In this section, we motivate and illustrate the main features of SCENIC, focusing on aspects of the language that make it particularly well-suited for describing geometric scenarios. Throughout, we use examples from our case study using SCENIC to generate traffic scenes to test and train autonomous cars, which will be detailed in Chapter 8.

5.2.1 High-Level Design Choices

We begin by discussing some high-level choices in the design of SCENIC, before moving on to specific language features. SCENIC is:

Imperative: SCENIC uses an imperative syntax familiar to users of languages like Python. Constructs such as loops and functions give SCENIC a great deal of flexibility compared to purely declarative formalisms like XML and VRML, which are the basis of input formats for various simulators (e.g. the Gazebo [99] and Webots [122] robotics simulators). For example, we can use a loop to create a line of cars following each other, then wrap this into a function for later use. However, SCENIC also allows *constraints* to be defined declaratively, as we will see later.

Object-oriented: Object-orientation provides a natural organizational principle for scenarios involving different types of physical objects, which can be modeled as classes. For example, we could define a class `Car` defining general characteristics of cars, e.g. their typical color distribution, and the fact that they are normally found on roads. We could then make subclasses for particular car models defining their sizes, etc., or (as we will see below) override the superclass for a particular instance to place a car on the sidewalk. This use of classes also improves compositionality, since we can define generic models like `Car` in a library for a particular application and reuse it in many different scenarios.

Probabilistic: As a probabilistic programming language, SCENIC allows sampling from distributions: `x = (-10, 10)` assigns `x` to a random number in the interval $(-10, 10)$. The ability to sample random values is useful not simply for generating random tests: it is necessary for modeling real-world stochasticity, for example encoding a distribution for the distance between successive cars in traffic, learned from data. This in turn is essential for using SCENIC to train systems based on machine learning: using randomness, we can generate training data matching the distribution the system will be used under. More generally, being a PPL makes it possible to use SCENIC as an environment specification language for cyber-physical systems which operate in an inherently noisy and unpredictable world.

Domain-specific: SCENIC provides readable, concise syntax for common geometric relationships that would otherwise require complex non-linear expressions and constraints. It has native support for geometric concepts like points, local coordinate systems, lines of sight, regions of space, and vector fields, including a variety of operators for manipulating these. For example, we can pick a uniformly random point in the region `road` by simply writing `Point on road`. As we will see, much of SCENIC's domain-specific syntax is not merely syntactic sugar; rather, it enables flexible, natural language-like ways of describing positions and orientations that would be difficult to implement as a library on top of an existing language. Furthermore, while SCENIC could be translated

into existing PPLs, using a new language allows us to impose restrictions enabling specialized sampling techniques not possible with general-purpose PPLs.

Integrated with Python: SCENIC is implemented by translation to Python and therefore allows full use of Python syntax for writing classes, functions, and so forth (except of course for the few Python constructs to which SCENIC gives new semantics). More importantly, SCENIC programs can invoke external Python code, making the vast ecosystem of Python packages available for use. This made it possible, for example, to create SCENIC libraries which generate workspaces by parsing Webots world files [122] using ANTLR [132], or OpenStreetMap data [59] using Python XML parsing packages [112].

Generic: Although SCENIC is a domain-specific language specialized for describing scenes, it is also generic in the sense of not being restricted to any particular application domain or simulator. While most of our examples will be based on the domain of visual perception for autonomous driving, this being our main case study in Chapter 8, we also give an example of a different domain (robotic motion planning) and simulator (Webots [122]) at the end of this section. Furthermore, we have also used SCENIC with the CARLA driving simulator [45] and the X-Plane flight simulator [141]. This was made possible by designing SCENIC to have a generic interface and only building in general geometric concepts.

In fact, SCENIC is also not restricted to generating visual data, but can produce data of any desired type — RGB+depth images, LIDAR point clouds, or trajectories from dynamical simulations — by interfacing it to an appropriate simulator. This requires only two steps:

1. writing a small SCENIC library defining the types of objects supported by the simulator, as well as the geometry of the workspace (e.g. any fixed obstacles).
2. writing an interface layer converting the scenes output by SCENIC (as simple lists of objects with XY coordinates, etc.) into the simulator's input format.

While the current version of SCENIC is primarily concerned with geometry, leaving the details of rendering and physics up to the simulator, the language allows putting distributions on any parameters the simulator exposes. For example, in GTA V the meshes of the various car models are fixed, but we can control their overall color. When performing simulations over time, we can also use SCENIC to specify distributions over parameters on the system dynamics.

5.2.2 Main Language Features

Classes, Objects, and Geometry

To start, suppose we want scenes of one car viewed from another on the road. We can write this very concisely in SCENIC:

```
1 from gta import Car
2 ego = Car
3 Car
```

Line 1 imports `gta`, a SCENIC library containing everything specific to our autonomous car case study: in particular, the class `Car` (in future examples we suppress the `import` line). Line 2 then creates a `Car` and assigns it to the special variable `ego` specifying the *ego object*. This is the reference point for the scenario: rendered images from the scenario will be from its perspective, and many of SCENIC's geometric operators use `ego` by default when a position is left implicit. Finally, line 3 creates a second `Car`. Note that we have not specified the position or any other properties of the two cars: this means they are inherited from the *default values* defined in the class `Car`. Slightly simplified, that definition begins:

```
1 class Car:
2     position: Point on road
3     heading: roadDirection at self.position
```

Here `road` is a *region*, one of SCENIC's primitive types, defined in the `gta` module to specify which points in the workspace are on a road. Similarly, `roadDirection` is a *vector field* specifying the nominal traffic direction at such points. The operator F `at` X simply gets the direction of the field F at point X, so line 3 sets a `Car`'s default `heading` to be the road direction at its `position`. The default `position`, in turn, is a `Point on road` (we will explain this syntax shortly), which means a uniformly random point on the road. Thus, in our simple scenario above both cars will be placed on the road and facing a reasonable direction, without our having to specify this explicitly.

We can of course override the class-provided defaults and define the position of a car more specifically. For example,

```
1 Car offset by (-10, 10) @ (20, 40)
```

creates a car that is 20–40 meters ahead of the camera, and up to 10 meters to the left or right, while still using the default heading (namely, being aligned with the road). Here the interval notation `(`X`, `Y`)` creates a uniform distribution on the interval, and X `@` Y creates a vector from xy coordinates (as in Smalltalk [73]).

Local Coordinate Systems

SCENIC provides a number of constructs for working with local coordinate systems, which are often helpful when building a scene incrementally out of component parts. Above, we saw how `offset by` could be used to position an object in the coordinate system of the `ego`, for instance placing a car a certain distance away from the camera. In fact, `ego` is a variable and can be reassigned, so we can set `ego` to one object, build a part of the scene around it, then reassign `ego` and build another part of the scene.

It is equally easy in SCENIC to use local coordinate systems around other objects or even arbitrary points. For example, suppose we want to make the scenario above more realistic by not requiring the car to be *exactly* aligned with the road, but to be within say 5°. We could write

```
Car offset by (-10, 10) @ (20, 40),
    facing (-5, 5) deg
```

but this is not quite what we want, since this sets the orientation of the car in *global* coordinates. Thus the car will end up facing within 5° of North, rather than within 5° of the road direction. Instead, we can use SCENIC's general operator X `relative to` Y, which can interpret vectors and headings as being in a variety of local coordinate systems:

```
Car offset by (-10, 10) @ (20, 40),
    facing (-5, 5) deg relative to roadDirection
```

If instead we want the heading to be relative to that of the ego car, so that the two cars are (roughly) aligned, we simply write `(-5, 5) deg relative to ego`.

Notice that since `roadDirection` is a vector field, it defines a local coordinate system at each point in space: at different points on the map, roads point different directions! Thus an expression like `15 deg relative to field` does not define a unique heading. The example above works because SCENIC knows that `(-5, 5) deg relative to roadDirection` depends on a reference position, and automatically uses the `position` of the `Car` being defined. This is a feature of SCENIC's system of *specifiers*, which we explain next.

Readable, Flexible Specifiers

The syntax `offset by` X and `facing` Y for specifying positions and orientations may seem unusual compared to typical constructors in object-oriented languages. There are two reasons why SCENIC uses this kind of syntax: first, readability. The second is more subtle and based on the fact that in natural language there are many ways to specify positions and other properties, some of which interact with each other. Consider the following ways one might describe the location of an object:

1. "is at position X" (absolute position);
2. "is just left of position X" (position based on orientation);
3. "is 3 m West of the taxi" (relative position);
4. "is 3 m left of the taxi" (a local coordinate system);
5. "is one lane left of the taxi" (another local coordinate system);
6. "appears to be 10 m behind the taxi" (relative to the line of sight);

7. "is 10 m along the road from the taxi" (following a potentially curving vector field).

These are all fundamentally different from each other: e.g., (4) and (5) differ if the taxi is not parallel to the lane.

Furthermore, these specifications combine other properties of the object in different ways: to place the object "just left of" a position, we must first know the object's `heading`; whereas if we wanted to face the object "towards" a location, we must instead know its `position`. There can be chains of such *dependencies*: "the car is 0.5 m left of the curb" means that the *right edge* of the car is 0.5 m away from the curb, not the car's `position`, which is its center. So the car's `position` depends on its `width`, which in turn depends on its `model`. In a typical object-oriented language, this might be handled by computing values for `position` and other properties and passing them to a constructor. For "a car is 0.5 m left of the curb" we might write:

```
model = Car.defaultModelDistribution.sample()
pos = curb.offsetLeft(0.5 + model.width / 2)
car = Car(pos, model=model)
```

Notice how `model` must be used twice, because `model` determines both the model of the car and (indirectly) its position. This is inelegant, and breaks encapsulation because the default model distribution is used outside of the `Car` constructor. The latter problem could be fixed by having a specialized constructor or factory function,

```
car = CarLeftOfBy(curb, 0.5)
```

but these would proliferate since we would need to handle all possible combinations of ways to specify different properties (e.g. do we want to require a specific model? Are we overriding the width provided by the model for this specific car?). Instead of having a multitude of such monolithic constructors, Scenic factors the definition of objects into potentially-interacting but syntactically-independent parts:

```
Car left of spot by 0.5, with model BUS
```

Here `left of` X `by` D and `with model` M are *specifiers* which do not have an order, but which *together* specify the properties of the car. Scenic works out the dependencies between properties (here, `position` is provided by `left of`, which depends on `width`, whose default value depends on `model`) and evaluates them in the correct order. To use the default model distribution we would simply leave off `with model BUS`; keeping it affects the `position` appropriately without having to specify `BUS` more than once.

Specifying Multiple Properties Together

Recall that we defined the default `position` for a `Car` to be a `Point on road`: this is an example of another specifier, `on` *region*, which specifies `position` to be a uniformly random point in the given region. This specifier illustrates another feature of Scenic, namely that

specifiers can specify multiple properties simultaneously. Consider the following scenario, which creates a parked car given a region `curb` defined in the `gta` library:

```
spot = OrientedPoint on visible curb
Car left of spot by 0.25
```

The function `visible` *region* returns the part of the region that is visible from the ego object. The specifier `on visible curb` will then set `position` to be a uniformly random visible point on the curb. We create `spot` as an `OrientedPoint`, which is a built-in class that defines a local coordinate system by having both a `position` and a `heading`. The `on` *region* specifier can also specify `heading` if the region has a preferred orientation (a vector field) associated with it: in our example, `curb` is oriented by `roadDirection`. So `spot` is, in fact, a uniformly random visible point on the curb, oriented along the road. That orientation then causes the car to be placed 0.25 m left of `spot` in `spot`'s local coordinate system, i.e. away from the curb, as desired.

In fact, Scenic makes it easy to elaborate the scenario without needing to alter the code above. Most simply, we could specify a particular model or non-default distribution over models by just adding `with model` M to the definition of the `Car`. More interestingly, we could produce a scenario for *badly*-parked cars by adding two lines:

```
spot = OrientedPoint on visible curb
badAngle = Uniform(1.0, -1.0) * (10, 20) deg
Car left of spot by 0.5,
    facing badAngle relative to roadDirection
```

This will yield cars parked 10–20° off from the direction of the curb, as seen in Figure 5.2. This illustrates how specifiers greatly enhance Scenic's flexibility and modularity.

Declarative Hard and Soft Constraints

Notice that in the scenarios above we never explicitly ensured that two cars will not intersect each other. Despite this, Scenic will never generate such scenes. This is because Scenic enforces several *default requirements*:

- All objects must be contained in the workspace, or a particular region if specified. For example, we can define the `Car` class so that all of its instances must be contained in the region `road` by default.
- Objects must not intersect each other (unless explicitly allowed).
- Objects must be visible from the ego object (so that they affect the rendered image; this requirement can be also disabled, for example if generating non-visual data).

Scenic also allows the user to define custom requirements checking arbitrary conditions built from various geometric predicates. For example, the following scenario produces a car headed roughly towards the camera, while still facing the nominal road direction:

Figure 5.2: A scene of a badly-parked car.

```
ego = Car on road
car2 = Car offset by (-10, 10) @ (20, 40), with viewAngle 30 deg
require car2 can see ego
```

Here we have used the *X* `can see` *Y* predicate, which in this case is checking that the ego car is inside the 30° view cone of the second car.

Requirements, called *observations* in other probabilistic programming languages (see, e.g., [79, 134]), are very convenient for defining scenarios because they make it easy to restrict attention to particular cases of interest. Note how difficult it would be to write the scenario above without the `require` statement: when defining the ego car, we would have to somehow specify those positions where it is possible to put a roughly-oncoming car 20–40 meters ahead (for example, this is not possible on a one-way road). Instead, we can simply place `ego` uniformly over all roads and let Scenic work out how to condition the distribution so that the requirement is satisfied. As this example illustrates, the ability to declaratively impose constraints gives Scenic much greater versatility than purely-generative formalisms like shape grammars (see e.g. [125]). Requirements also improve encapsulation, since we can restrict an existing scenario without altering it.

The constraint in our example above is a *hard requirement* which must always be satisfied.

SCENIC also allows imposing *soft requirements* that need only be true with some minimum probability:

```
require[0.5] car2 can see ego
```

As we will see later, soft requirements can be implemented in general-purpose PPLs using only observations and conditionals (and in fact some PPLs provide similar constructs, or allow the likelihood of executions to be arbitrarily adjusted, e.g. Stan [23]). However, soft requirements are not simply syntactic sugar in SCENIC because the language does not allow probabilistic control flow. They are useful, for example, in ensuring adequate representation of a particular condition when generating a training set: for instance, we could require that at least 90% of the images have a car driving on the right side of the road.

Mutations

Finally, a common testing paradigm is to randomly generate *variations* of existing tests. For example, in mutational fuzz testing, one might test an audio player by applying random *mutations* to a set of ordinary MP3 files, flipping bits and deleting fields [165]. The same idea has been used for testing autonomous cars in simulation, generating many variations on complex or problematic scenarios observed in real-world testing [114].

SCENIC supports this paradigm by providing syntax for performing mutations in a compositional manner, adding variety to a scenario without changing its code. For example, given a complex scenario involving a `taxi`, we can add one additional line:

```
taxi = Car at 120@300, facing 37 deg, ...
...
mutate taxi
```

The `mutate` statement will add Gaussian noise to the `position` and `heading` of taxi, while still enforcing all built-in and custom requirements. The standard deviation of the noise can be scaled by writing, for example, `mutate taxi by 2` (which adds twice as much noise), and we will see later that it can be controlled separately for `position` and `heading`.

5.2.3 A Worked Example

We conclude this section with a larger example of a SCENIC program which also illustrates the language's utility across domains and simulators. Specifically, we consider the problem of testing a motion planning algorithm for a Mars rover able to climb over rocks. Such robots can have very complex dynamics, with the feasibility of a motion plan depending on exact details of the robot's hardware and the geometry of the terrain. We can use SCENIC to write a scenario generating challenging cases for a planner to solve in simulation.

We will write a scenario representing a rubble field of rocks and pipes with a bottleneck between the rover and its goal that forces the path planner to consider climbing over a rock.

First, we import a small library `mars` defining the (empty) workspace and several types of objects: the `Rover` itself, the `Goal` (represented by a flag), and debris classes `Rock`, `BigRock`, and `Pipe`. `Rock` and `BigRock` have fixed sizes, and the rover can climb over them; `Pipe` cannot be climbed over, and can represent a pipe of arbitrary length, controlled by the `height` property (`height` corresponding to SCENIC's y axis).

```
1  import mars
```

Then we create the rover at a fixed position and the goal at a random position on the other side of the workspace:

```
2  ego = Rover at 0 @ -2
3  goal = Goal at (-2, 2) @ (2, 2.5)
```

Next, we pick a position for the bottleneck, requiring it to lie roughly on the way from the robot to its goal, and place a rock there.

```
4  bottleneck = OrientedPoint offset by (-1.5, 1.5) @ (0.5, 1.5),
5                              facing (-30, 30) deg
6  require abs((angle to goal) - (angle to bottleneck)) <= 10 deg
7  BigRock at bottleneck
```

Note how we define `bottleneck` as an `OrientedPoint`, with a range of possible orientations: this is to set up a local coordinate system for positioning the pipes making up the bottleneck. Specifically, we position two pipes of varying lengths on either side of the bottleneck, with their ends far enough apart for the robot to be able to pass between:

```
8  halfGapWidth = (1.2 * ego.width) / 2
9  leftEnd = OrientedPoint left of bottleneck by halfGapWidth,
10                         facing (60, 120) deg relative to bottleneck
11 rightEnd = OrientedPoint right of bottleneck by halfGapWidth,
12                          facing (-120, -60) deg relative to bottleneck
13 Pipe ahead of leftEnd, with height (1, 2)
14 Pipe ahead of rightEnd, with height (1, 2)
```

Finally, to make the scenario slightly more interesting we add several additional obstacles, positioned either on the far side of the bottleneck or anywhere at random.

```
15 BigRock beyond bottleneck by (-0.5, 0.5) @ (0.5, 1)
16 BigRock beyond bottleneck by (-0.5, 0.5) @ (0.5, 1)
17 Pipe
18 Rock
19 Rock
20 Rock
```

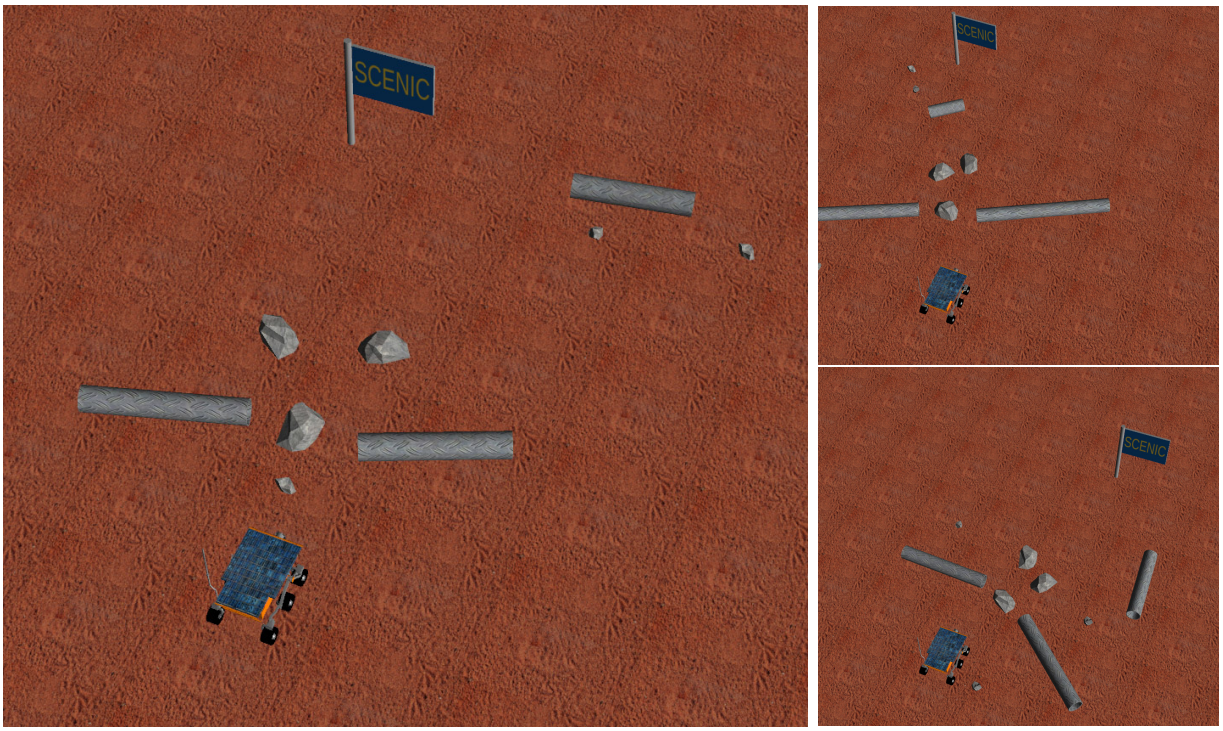


Figure 5.3: Debris fields generated with SCENIC for testing a robotic motion planner.

This completes the scenario. Several scenes generated from it, visualized in the Webots simulator [122], are shown in Figure 5.3. This example, the badly-parked car scenario of Figure 5.2, and the bumper-to-bumper traffic scenario of Figure 5.1 illustrate the versatility of SCENIC in constructing a wide range of interesting scenarios. Many more examples can be found in Chapter 8 and in the SCENIC distribution [65].

5.3 Syntax of SCENIC

Having now outlined the main design choices and constructs of SCENIC, we proceed to define the language's syntax in detail. SCENIC inherits without change the Python syntax for conditionals, loops, functions, and methods, which we do not describe further (see the Python documentation [60] for details), focusing on the new elements. The novel syntax, as we saw above, is largely devoted to expressing geometric relationships in a concise and flexible manner. Figure 5.4 gives a formal grammar for this syntax, which will be elaborated below. For now we will describe the meaning of SCENIC's constructs informally, leaving formal semantics to Section 5.4.

```
scenario       := (statement)*
boolean        := True | False | booleanOperator
scalar         := number | distribution | scalarOperator
distribution   := baseDistribution | resample(distribution)
vector         := scalar @ scalar | Point | vectorOperator
heading        := scalar | OrientedPoint | headingOperator
direction      := heading | vectorField
value          := boolean | scalar | vector | direction | region
                | instance | instance.property
classDefinition := class class[(superclass)]:
                       (property: defaultValueExpression)*
instance       := class specifier, ...
specifier      := with property value | positionSpecifier | headingSpecifier
```

Figure 5.4: Simplified top-level SCENIC grammar. *Point* and *OrientedPoint* are instances of the corresponding classes. See Table 5.5 for statements, Figure 5.6 for operators, Table 5.1 for *baseDistribution*, and Tables 5.3 and 5.4 for *positionSpecifier* and *headingSpecifier*.

5.3.1 Primitive Data Types

SCENIC provides several primitive data types:

Booleans expressing truth values.

Scalars representing distances, angles, etc. as floating-point numbers, which can be sampled from various distributions (see Section 5.3.2).

Vectors representing positions and offsets in space, constructed from coordinates with the syntax `X @ Y` (inspired by Smalltalk [73]). By convention, coordinates are in meters, although the semantics of SCENIC does not depend on this. More significantly, the vector syntax is specialized for 2-dimensional space. The 2D assumption dramatically simplifies much of SCENIC's syntax (particularly that dealing with orientations, as we will see below), while still being adequate for a variety of applications. However, it is important to note that the fundamental ideas of SCENIC are not specific to 2D, and it would be easy to extend our implementation of the language to support 3D space.

Headings representing orientations in space. Conveniently, in 2D these can be expressed using a single angle (rather than Euler angles or a quaternion). SCENIC represents headings in radians, measured anticlockwise from North, so that a heading of 0 is due North and a heading of $\pi/2$ is due West. We use the convention that the heading of a local coordinate system is the heading of its y-axis, so that, for example, `-2 @ 3` means 2 meters left and 3 ahead.

Table 5.1: Distributions built into SCENIC. All parameters are *scalars* except for *value*, which can be any value.

Syntax	Distribution
`(`*low*`, `*high*`)`	uniform on an interval
`Normal(`*mean*`, `*stdDev*`)`	normal with given mean and standard deviation
`Uniform(`*value*`, ...)`	uniform over a finite set of values
`Discrete({`*value*`: `*wt*`, ...})`	discrete with given values and weights

Vector Fields associating an orientation (i.e. a heading) to each point in space. For example, a vector field could represent the shortest paths to a destination, or the nominal traffic direction on a road.

Regions representing sets of points in space. SCENIC provides a variety of ways to define Regions: rectangles, circular sectors, line segments, polygons, occupancy grids, and explicit lists of points. These particular types of Regions are all implemented using Python classes, so we do not discuss them further here (see the SCENIC implementation for details [65]).

Regions can have an associated vector field giving points in the region preferred orientations. For example, a Region representing a lane of traffic could have a preferred orientation aligned with the lane, so that (as we will see later) we can easily talk about distances *along* the lane, even if it curves. Another possible use of preferred orientations is to give the surface of an object normal vectors, so that other objects placed on the surface face outward by default.

In addition to these basic types, SCENIC provides a system of classes and objects, which will be described shortly in Section 5.3.3. First, we discuss SCENIC's syntax for distributions.

5.3.2 Distributions

SCENIC provides 4 basic types of distributions, shown in Table 5.1. There are two continuous distributions, uniform and normal, with uniform distributions over an interval being written simply as that interval: thus `(-1, 1)` represents a uniform distribution over $(-1, 1)$.

SCENIC also supports arbitrary discrete distributions, which can be over any type of value. For example, if we have Regions `leftLane` and `rightLane`, we could write

```
lane = Discrete({
    leftLane: 2,
    rightLane: 1
})
```

to pick the left lane twice as often as the right lane. SCENIC allows the shorthand `Uniform(...)` for a uniform discrete distribution over a given set of values.

There is in fact a 5th type of distribution built into SCENIC: a uniform distribution over points in a Region. It is not shown in Table 5.1, since it is only available using the specifier syntax we will discuss in Section 5.3.4. As we will see there, we can for example write `Point on road` to generate a uniformly-random point in the Region `road`, or `Point visible from taxi` for a uniformly-random point visible from the object `taxi`.

As is typical in an imperative probabilistic programming language, syntax denoting a distribution really refers to a *sample* from that distribution. For example, the program

```
x = (0, 1)
y = x @ x
```

should be read as assigning `x` a sample from the unit interval $(0, 1)$. Thus, the distribution of `y` is not uniform over the unit box, but rather over its diagonal. This choice makes SCENIC programs have the semantics one would expect from an ordinary, non-probabilistic programming language, where reading twice from the same variable yields the same value each time.

However, for cases where multiple samples from the same distribution are desired, SCENIC provides a `resample(`D`)` function defined on primitive distributions D (not general expressions involving distributions, which would lead to confusion about the semantics). This function returns an independent sample from D, conditioned on the values of its parameters, if any. For example, in the program

```
x = Uniform(0, 5)
y = (x, x+1)
z = resample(y)
```

with probability $1/2$ both `y` and `z` are independent uniform samples from $(0, 1)$, and with probability $1/2$ they are independent uniform samples from $(5, 6)$. It is never the case that `y` $\in (0, 1)$ and `z` $\in (5, 6)$ or vice versa, which would require inconsistent assignments to `x`.

5.3.3 Objects

SCENIC provides a simple system of objects, organized into single-inheritance classes. Class definitions specify a set of *properties* their instances have, together with *default values* for these properties (see Figure 5.4). Default values can be distributions, which allows defining a prior distribution on a property of an entire class of object. For example, we can define the prior distribution of the `position` of a `Pedestrian` to be uniformly at random on a sidewalk, but override this in particular cases, e.g. to place a pedestrian in the middle of the road. To support this usage, default value expressions are evaluated each time an object is created; thus, if we define

```
class Rock:
    weight: (50, 300)
```

then each `Rock` will have a property `weight` with value drawn *independently* from `(50, 300)`, as we would expect.

Default values may use the special syntax `self.`*property* to refer to one of the other properties of the object, which is then a *dependency* of the default value. For example, writing

```
class Pedestrian:
    height: Normal(1.8, 0.1)
    weight: self.height * 100
```

the default value of `weight` depends on `height`, and so the latter is automatically sampled before the former. The procedure which SCENIC uses to determine the sampling order, including detecting cyclic dependencies, will be described in Section 5.4.3.

The ability to use an arbitrary expression as a default value, even one which depends on other properties of the object, makes object creation in SCENIC concise and modular. We can see this using a simplified version of the class `Car` from our case study in Chapter 8:

```
class Car:
    heading: roadDirection at self.position
    width: self.model.width
    height: self.model.height
```

Here, the dimensions of a `Car` are automatically inferred from its `model`, but can be easily overridden if we want an extra-long truck, for example. More interestingly, the `heading` is defined to be the nominal traffic direction (given by the vector field `roadDirection`) at the car's location, and this holds regardless of how we specify the latter.

There are 3 classes built into SCENIC: `Point`, `OrientedPoint`, and `Object`. `Point` represents a location in space and provides the `position` property, while its subclass `OrientedPoint` additionally provides an orientation via the `heading` property and so defines a local coordinate system. `Object` represents a physical object, extending `OrientedPoint` by defining a bounding box via the properties `width` and `height` (for the X and Y axes respectively). `Object` is also the default superclass for user-defined classes when none is specified. The properties provided by the built-in classes, along with their default values, are shown in Table 5.2.

To simplify notation, `Point` and `OrientedPoint` are automatically interpreted as vectors or headings in contexts expecting these, as shown in Figure 5.4. A `Point` is interpreted as a vector by reading its `position` property, and an `OrientedPoint` is interpreted as a heading by reading its `heading` property. For example, given an `Object` called `taxi`, we can simplify the two expressions

Table 5.2: Properties of `Point`, `OrientedPoint`, and `Object`.

Property	Default	Meaning
`position`	0 @ 0	position in global coordinates
`viewDistance`	50	distance for the '`can see`' operator (Section 5.3.5)
`mutationScale`	0	overall scale of mutations (see Section 5.3.6)
`positionStdDev`	1	mutation standard deviation for `position`
`heading`	0	heading in global coordinates
`viewAngle`	360°	angle for the '`can see`' operator
`headingStdDev`	5°	mutation standard deviation for `heading`
`width`	1	width of bounding box (X axis)
`height`	1	height of bounding box (Y axis)
`regionContainedIn`	`workspace`	`Region` the object must lie within
`allowCollisions`	false	whether collisions are allowed
`requireVisible`	true	whether object must be visible from `ego`

```
taxi.position offset by 1 @ 2
30 deg relative to taxi.heading
```

to the equivalent

```
taxi offset by 1 @ 2
30 deg relative to taxi
```

Cases where this would be ambiguous, e.g. `taxi relative to limo` where both objects provide both `position` and `heading`, are illegal (being caught by a simple type system); the more verbose syntax must be used instead.

Finally, creation of objects in SCENIC has two unusual aspects. First, definitions of instances of `Object` are the only expressions in SCENIC with a side effect, namely creating an object in the generated scene. Second, the properties of an object are defined using system of *specifiers* outlined above, which we now discuss in detail.

5.3.4 Specifiers

As seen in Figure 5.4, instances of classes are created by writing the name of the class followed by a (possibly empty) comma-separated list of specifiers. The specifiers are combined, possibly adding default specifiers from the class definition, to form a complete specification of all properties of the object. Arbitrary properties, including user-defined properties with no meaning in SCENIC, can be specified with the generic specifier `with` *property value*, while SCENIC provides many other specifiers for the built-in properties `position` and `heading`. These are shown in Tables 5.3 and 5.4 respectively (we will describe their semantics shortly).

Table 5.3: Specifiers for `position`. Those in the second group also optionally specify `heading`.

Specifier	Dependencies
`at` *vector*	—
`offset by` *vector*	—
`offset along` *direction* `by` *vector*	—
(`left` | `right`) `of` *vector* [`by` *scalar*]	`heading`, `width`
(`ahead of` | `behind`) *vector* [`by` *scalar*]	`heading`, `height`
`beyond` *vector* `by` *vector* [`from` *vector*]	—
`visible` [`from` (*Point* | *OrientedPoint*)]	—
(`in` | `on`) *region*	—
(`left` | `right`) `of` (*OrientedPoint* | *Object*) [`by` *scalar*]	`width`
(`ahead of` | `behind`) (*OrientedPoint* | *Object*) [`by` *scalar*]	`height`
`following` *vectorField* [`from` *vector*] `for` *scalar*	—

Table 5.4: Specifiers for `heading`.

Specifier	Dependencies
`facing` *heading*	—
`facing` *vectorField*	`position`
`facing` (`toward` | `away from`) *vector*	`position`
`apparently facing` *heading* [`from` *vector*]	`position`

In general, a specifier is a function mapping one assignment of properties to another. As input, it takes in values for zero or more properties, its *dependencies*. As output, it *specifies* values for one or more other properties. Consider two specifiers from Table 5.3:

```
1 Car at 1 @ 2              # no dependencies; specifies 'position'
2 Car left of taxi by 5     # depends on 'width'; specifies 'position'
```

The `at 1 @ 2` specifier has no dependencies and specifies the constant value `1 @ 2` for `position`. On the other hand, `left of taxi by 5` depends on `width`, since in order to place the `Car` 5 meters to the left of the `taxi` we need to know how wide the `Car` is.

The `left of` specifier also illustrates another feature of specifiers, namely that they can specify properties *optionally*. This means that other specifiers which specify the same property, but not optionally, will override them. In the example above, `left of` optionally specifies `heading` to be the same as `taxi`, so that the `Car` ends up 5 meters to the left of `taxi` and parallel with it. But since the specification is optional, we can override it:

```
1 Car left of taxi by 5, facing toward taxi
```

The `facing toward` specifier, from Table 5.4, specifies `heading` non-optionally, so it takes precedence.

Another interesting example of optional specification is the `in/on` *region* specifier. It always specifies `position` to be a uniformly random point in the given Region. However, it can also optionally specify `heading`, namely when the Region has a preferred orientation. If `road` is such a region, then we can write for example

```
1 Car on road                    # use heading from orientation of road
2 Car on road, facing 30 deg     # override with specific heading
```

If `road` did not have a preferred orientation, the behavior of line 2 would remain the same, while line 1 would use the default `heading` defined in the `Car` class.

Specifiers are combined to determine the properties of an object by evaluating them in an order which ensures that their dependencies are always already assigned. If there is no such order or a single property is specified twice, the scenario is ill-formed. The procedure by which the order is found, taking into account properties that are optionally specified and default values, will be described in Section 5.4.3.

Finally, we informally describe the semantics of all of the specifiers. For formal definitions, see Section 5.4.4. Several specifiers are illustrated in Figure 5.5.

Specifiers for `position`

`at` *vector*: Positions the object at the given global coordinates.

`offset by` *vector*: Positions the object at the given coordinates in the local coordinate system of `ego` (which must already be defined). See Figure 5.5 for an example.

`offset along` *direction* `by` *vector*: Positions the object at the given coordinates, in a local coordinate system centered at `ego` and oriented along the given direction (which, if a vector field, is evaluated at `ego` to obtain a heading).

(`left` | `right`) `of` *vector* [`by` *scalar*]: Depends on `heading` and `width`. Without the optional `by` *scalar*, positions the object immediately to the left/right of the given position; i.e., so that the midpoint of the object's right/left edge is at that position. If `by` *scalar* is used, the object is placed further to the left/right by the given distance.

(`ahead of` | `behind`) *vector* [`by` *scalar*]: As above, except placing the object ahead of or behind the given position (so that the midpoint of the object's back/front edge is at that position); thereby depending on `heading` and `height`.

`beyond` *vector* `by` *vector* [`from` *vector*]: Positions the object at coordinates given by the second vector, in a local coordinate system centered at the first vector and oriented along the line of sight to there from the `ego`. For example, `beyond taxi by 0 @ 3` means 3 meters directly behind the taxi as viewed by the camera. Another example

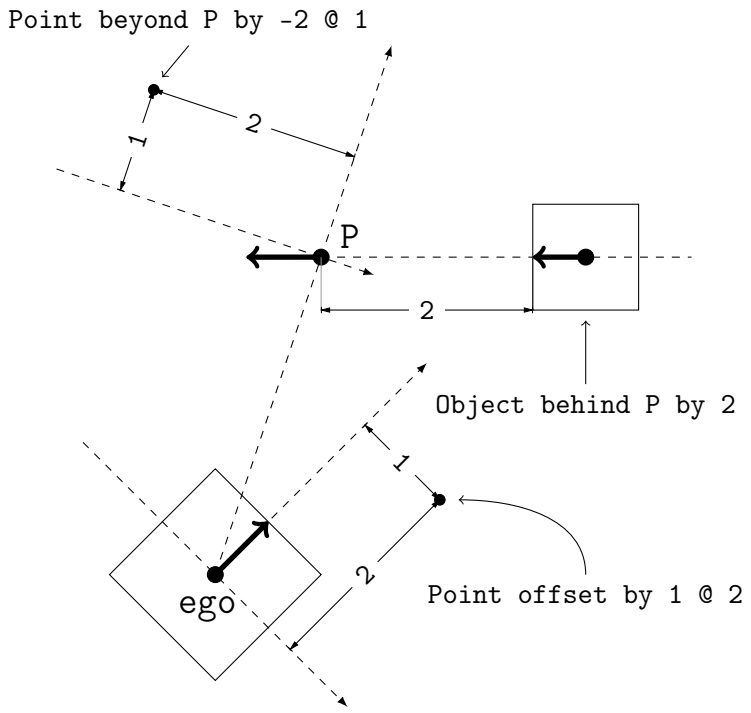


Figure 5.5: Examples of several SCENIC specifiers, using the `ego` object and an `OrientedPoint` `P`. Instances of `OrientedPoint` are shown as bold arrows.

is shown in Figure 5.5. With the optional `from` *vector*, the line of sight is computed from the given position rather than that of `ego`.

`visible` [`from` (*Point* | *OrientedPoint*)]: Positions the object uniformly at random in the visible region of the `ego` (see Section 5.3.5 for a discussion of the visibility model and the `can see` operator), or of the given `Point`/`OrientedPoint` if given.

Specifiers for `position` and optionally `heading`

(`in` | `on`) *region*: Positions the object uniformly at random in the given Region. If the Region has a preferred orientation (a vector field), also optionally specifies `heading` to be equal to that orientation at the object's position.

(`left` | `right`) `of` (*OrientedPoint* | *Object*) [`by` *scalar*]: Positions the object to the left/right of the given `OrientedPoint`, depending on the object's `width`. Also optionally specifies `heading` to be the same as that of the `OrientedPoint`. If the `OrientedPoint` is in fact an `Object`, the object being constructed is positioned to the left/right of its left/right edge. So for example, in

```
1 A = Object at 0 @ 0, with width 2
2 B = Object right of A, with width 4
```

the object `B` has center `3 @ 0`, so that it is adjacent to `A` on the right. When the optional `by` *scalar* is given, the object is placed further to the left/right by the given distance.

(`ahead of` | `behind`) (*OrientedPoint* | *Object*) [`by` *scalar*]: As above, except positioning the object ahead of or behind the given `OrientedPoint`, thereby depending on `height`.

`following` *vectorField* [`from` *vector*] `for` *scalar*: Positions the object at a point obtained by following the given vector field for the given distance starting from `ego` (or the position optionally provided with `from` *vector*). Optionally specifies `heading` to be the heading of the vector field at the resulting point. Uses a forward Euler approximation of the continuous vector field (see 5.4.4).

Specifiers for `heading`

`facing` *heading*: Orients the object along the given heading in global coordinates.

`facing` *vectorField*: Orients the object along the given vector field at the object's `position`.

`facing` (`toward` | `away from`) *vector*: Orients the object toward/away from the given position (thereby depending on the object's `position`).

`apparently facing` *heading* [`from` *vector*]: Orients the object so that it has the given heading with respect to the line of sight from `ego` (or from the position given by the optional `from` *vector*). For example, `apparently facing 90 deg` orients the object so that the camera views its left side head-on. See also the `apparent heading of` operator, illustrated in Figure 5.7.

5.3.5 Operators

SCENIC's operators are shown in Figure 5.6, organized by the type of value they compute. The standard arithmetic and Boolean operators have their usual meanings, so we will only discuss the geometric operators. As for specifiers, we will only define the operators informally here, giving formal semantics in Section 5.4.5. Examples of some of the more interesting operators are shown in Figure 5.7. Note that some operators are polymorphic: for example, *region* `visible from` *X* allows *X* to be either a *Point* or *OrientedPoint*, and has different meanings in each case (even though instances of `OrientedPoint` are also instances of `Point`).

Scalar Operators

`relative heading of` *heading* [`from` *heading*]: The relative heading of the given heading with respect to `ego` (or the heading provided with the optional `from` *heading*).

scalarOperator := `max(`*scalar*`, ...)` | `min(`*scalar*`, ...)`
| `-`*scalar* | `abs(`*scalar*`)` | *scalar* (`+` | `-` | `*` | `/` | `**`) *scalar*
| `relative heading of` *heading* [`from` *heading*]
| `apparent heading of` *OrientedPoint* [`from` *vector*]
| `distance` [`from` *vector*] `to` *vector*
| `angle` [`from` *vector*] `to` *vector*
booleanOperator := `not` *boolean* | *boolean* (`and` | `or`) *boolean*
| *scalar* (`==` | `!=` | `<` | `>` | `<=` | `>=`) *scalar*
| (*Point* | *OrientedPoint*) `can see` (*vector* | *Object*)
| (*vector* | *Object*) `in` *region*
headingOperator := *scalar* `deg` | *vectorField* `at` *vector*
| *direction* `relative to` *direction*
vectorOperator := *vector* (`relative to` | `offset by`) *vector*
| *vector* `offset along` *direction* `by` *vector*
regionOperator := `visible` *region*
| *region* `visible from` (*Point* | *OrientedPoint*)
orientedPointOperator := *vector* `relative to` *OrientedPoint*
| *OrientedPoint* `offset by` *vector*
| (`front` | `back` | `left` | `right`) `of` *Object*
| (`front` | `back`) (`left` | `right`) `of` *Object*

Figure 5.6: SCENIC operators by result type.

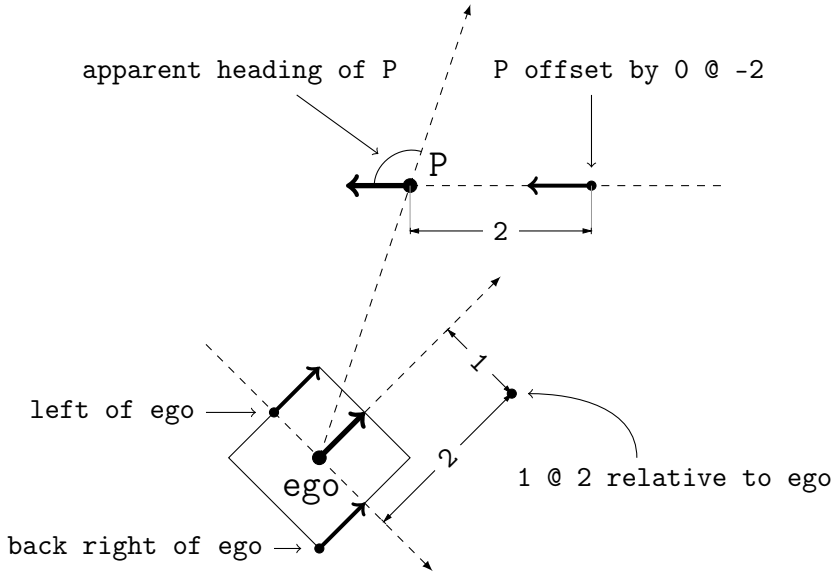


Figure 5.7: Various SCENIC operators applied to the `ego` object and an `OrientedPoint P`, as in Figure 5.5. Instances of `OrientedPoint` are shown as bold arrows.

`apparent heading of` *OrientedPoint* [`from` *vector*]: The apparent heading of the `OrientedPoint`, with respect to the line of sight from `ego` (or the position provided with the optional `from` *vector*). See Figure 5.7 for an example.

`distance` [`from` *vector*] `to` *vector*: The distance to the given position from `ego` (or the position provided with the optional `from` *vector*).

`angle` [`from` *vector*] `to` *vector*: The heading to the given position from `ego` (or the position provided with the optional `from` *vector*). For example, if `angle to taxi` is zero, then `taxi` is due North of `ego`.

Boolean Operators

(*Point* | *OrientedPoint*) `can see` (*vector* | *Object*): Whether or not a position or `Object` is visible from a `Point` or `OrientedPoint`. Visible regions are defined as follows: a `Point` can see out to a certain distance, and an `OrientedPoint` restricts this to the circular sector along its `heading` with a certain angle. The distance and angle are defined by the `viewDistance` and `viewAngle` properties in Table 5.2. A position is then visible if it lies in the visible region, and an `Object` is visible if its bounding box intersects the visible region. Note that Scenic's visibility model does not take into account occlusion, although this would be straightforward to add.

(*vector* | *Object*) `in` *region*: Whether a position or `Object` lies in the region; for the latter, the `Object`'s bounding box must be contained in the region. This allows us to use the predicate in two ways:

```
1 car.position in road    # center of car is on the road
2 car in road             # entire car is on the road
```

Heading Operators

scalar `deg`: The given heading, interpreted as being in degrees. For example `90 deg` evaluates to $\pi/2$.

vectorField `at` *vector*: The heading specified by the vector field at the given position.

direction `relative to` *direction*: The first direction, interpreted as an offset relative to the second direction. For example, `-5 deg relative to 90 deg` is simply `85 deg`. If either direction is a vector field, then this operator yields an expression depending on the `position` property of the object being specified. So if `roadDirection` is a vector field, we can write for example

```
1 direction = 30 deg relative to roadDirection  # depends on 'position'
2 Car at 0@0, facing direction  # well-defined
3 require direction > 40 deg    # error: used outside object context
```

Vector Operators

vector (`relative to` | `offset by`) *vector*: The first vector, interpreted as an offset relative to the second vector (or vice versa). For example, `5@5 relative to 100@200` is `105@205`. Note that this polymorphic operator has a specialized version for instances of `OrientedPoint`, defined below (so for example `-3@0 relative to taxi` will not use this vector version, even though the `Object taxi` can be coerced to a vector).

vector `offset along` *direction* `by` *vector*: The second vector, interpreted in a local coordinate system centered at the first vector and oriented along the given direction (which, if a vector field, is evaluated at the first vector to obtain a heading).

Region Operators

`visible` *region*: The part of the given region visible from `ego`.

region `visible from` (*Point* | *OrientedPoint*): The part of the given region visible from the given `Point`/`OrientedPoint`. See the discussion of visible regions under the `can see` operator above.

OrientedPoint Operators

vector `relative to` *OrientedPoint*: The given vector, interpreted in the local coordinate system of the `OrientedPoint`. So for example `1 @ 2 relative to ego` is 1 meter to the right and 2 meters ahead of `ego`, as shown in Figure 5.7.

OrientedPoint `offset by` *vector*: Equivalent to *vector* `relative to` *OrientedPoint* above.

(`front` | `back` | `left` | `right`) `of` *Object*: The midpoint of the corresponding edge of the bounding box of the `Object`, oriented along its `heading`. See Figure 5.7 for an example.

(`front` | `back`) (`left` | `right`) `of` *Object*: The corresponding corner of the `Object`'s bounding box, also oriented along its `heading`. See e.g. `back right of ego` in Figure 5.7.

5.3.6 Statements

Finally, we describe the syntax of Scenic's statements. As previously mentioned, a variety of statements including conditionals, loops, function definitions, and so forth are inherited directly from Python, so we do not describe them here. The other statements in Scenic are listed in Table 5.5. We have already discussed class and object definitions above, and variable assignment behaves in the standard way. Below, we summarize the remaining statements.

`import` *module*: Imports a Scenic or Python module. This statement behaves as in Python, but when importing a Scenic module M it also imports any objects created and requirements imposed in M. Scenic also supports the form `from` *module* `import`

Table 5.5: SCENIC statements (excluding loops, etc. inherited from Python).

Syntax	Meaning
`import` *module*	import SCENIC/Python module
identifier `=` *value*	variable assignment
`param` *identifier* `=` *value*`, ...`	global parameter assignment
classDefn	class definition
instance	object definition
`require` *boolean*	hard requirement
`require[`*number*`]` *boolean*	soft requirement
`mutate` *identifier*`, ...` [`by` *number*]	enable mutation

identifier`, ...`, which as in Python imports the module plus one or more identifiers from its namespace.

`param` *identifier* `=` *value*`, ...`: Defines *global parameters* of the scenario. These have no semantics in SCENIC, simply having their values included as part of the generated scene, but provide a general-purpose way to encode arbitrary global information. For example, if we write

```
param timeOfDay = (0, 24) * 3600
```

then the resulting scenes will have a parameter `timeOfDay`, which a simulator could read to set the time of day appropriately.

`require` *boolean*: Defines a *hard requirement*, requiring that the given condition hold in all instantiations of the scenario. As noted above, this is equivalent to an *observe* statement in other probabilistic programming languages (see e.g. [134, 123, 79]).

`require[`*number*`]` *boolean*: Defines a *soft requirement*, requiring that the given condition hold with at least the given probability (which must be a constant). There are several ways the semantics of soft requirements could be defined: SCENIC uses the natural definition that `require[`p`]` B is equivalent to a hard requirement `require` B that is only enforced with probability p.

`mutate` *identifier*`, ...` [`by` *number*]: Enables *mutation* of the given list of objects, adding Gaussian noise with the given standard deviation (default 1) to their `position` and `heading` properties. If no objects are specified, mutation applies to every `Object` already created. Some examples:

```
mutate taxi          # perturbs position/heading of taxi
mutate taxi by 2     # perturbs taxi by twice as much
mutate               # perturbs all existing Objects
```

The `mutate` statement works by assigning the `mutationScale` property of `Point`, shown in Table 5.2. The amount of noise added to the `position` and `heading` properties can be controlled independently by setting the `positionStdDev` and `headingStdDev` properties, which multiply `mutationScale`. So for example we can add `with positionStdDev 0` when creating an object to prevent its `position` from being mutated.

Because mutation is designed to allow easily constructing variants of existing scenarios, mutation noise is added at the end of running the SCENIC program (but before checking requirements). This means that noise added to one object does not affect the properties of other objects defined in terms of it. For example, if we write

```
ego = Object at 0 @ 0, facing 0    # at origin, facing North
B = Object offset by 0 @ 10        # 10 meters ahead of ego
mutate ego
```

then mutation will cause `ego` to no longer face exactly North; however, `B` will still be positioned exactly at `0 @ 10`, because its position is computed before mutation takes place. In this way, we can easily perturb one object without affecting others. On the other hand, if maintaining the precise *relationship* between objects is important, we can modify the scenario directly instead of using `mutate`:

```
ego = Object at 0 @ 0, facing Normal(0, 5 deg)
B = Object offset by 0 @ 10
```

5.4 Semantics of SCENIC

We have now described the syntax of SCENIC programs, and informally explained their meaning. In this section, we give a precise semantics for SCENIC expressions and statements, building up to a semantics for a complete program as a distribution over scenes.

5.4.1 Overview and Notation

The output of a SCENIC program is a *scene* consisting of the assignment to all the properties of each `Object` defined in the scenario, plus any global parameters defined with `param`. Since SCENIC is a probabilistic programming language, the semantics of a program is actually a *distribution* over possible outputs, here scenes. As for other imperative PPLs, the semantics can be defined operationally as an interpreter, but with two differences from non-probabilistic programming languages. First, the interpreter makes random choices when evaluating distributions [152]. For example, the SCENIC statement `x = (0, 1)` updates the state of the interpreter by assigning a value to `x` drawn from the uniform distribution on the interval $(0, 1)$. In this way every possible run of the interpreter has a probability associated with it. Second, every run where a `require` statement (the equivalent of an "observation" in other PPLs) is violated gets discarded, and the run probabilities appropriately normalized

(see, e.g., [35]). For example, adding the statement `require x > 0.5` above would yield a uniform distribution for `x` over the interval $(0.5, 1)$.

Along these lines, we will give a small-step operational semantics for SCENIC. As in our discussion of syntax, we will focus on the aspects of SCENIC that set it apart from ordinary imperative languages, skipping standard inference rules for sequential composition, arithmetic operations, etc. that we essentially use without change. In rules for statements, we will denote a state of a SCENIC program by $\langle s, \sigma, \pi, \mathcal{O} \rangle$, consisting of 4 parts:

- s is the statement (i.e., remaining part of the program) to be executed;
- σ is the current variable assignment (a map from identifiers to values);
- π is the current global parameter assignment (for `param` statements);
- $\mathcal{O}$ is the set of all objects defined so far.

In rules for expressions, we use the same notation, although we sometimes suppress the state on the right-hand side of rules for expressions without side effects: $\langle e, \sigma, \pi, \mathcal{O} \rangle \to v$ means that in the state $(\sigma, \pi, \mathcal{O})$, the expression e evaluates to the value v without side effects. Since none of the specifiers and operators have side effects, to simplify notation further we often write $[\![X]\!]$ for the value of the expression X in the current state, rather than giving explicit rules. Following the notation of Saheb-Djahromi [152] and Claret et al. [35], we write $\to^p$ for a rewrite rule that fires with probability p (probability density p, in the case of continuous distributions). We will discuss the meaning of such rules in more detail below.

Throughout this section, S indicates a *scalar*, V a *vector*, H a *heading*, F a *vectorField*, R a *region*, P a `Point`, and OP an `OrientedPoint`. Figure 5.8 defines notation used in the rest of the semantics. Most is self-explanatory; $\mathit{offsetLocal}(OP, v)$ offsets OP by v in its own local coordinate system (maintaining its `heading`), $\mathit{visibleRegion}(X)$ defines visible regions as explained in Section 5.3.5, and $\mathit{fwdEuler}(x, d, F)$ computes the forward Euler approximation of following the vector field F for distance d from x (with N being an implementation-defined parameter specifying how many steps should be used; our implementation uses $N = 4$ by default).

We begin by defining SCENIC's expressions — distributions, object definitions, specifiers, and operators — before moving on to statements and entire SCENIC programs.

5.4.2 Distributions

As in a typical imperative probabilistic programming language, a distribution evaluates to a *sample* from the distribution, following the rule in Figure 5.9. For example, in the expression `(1, 4)`, the *baseDistribution* is a uniform interval distribution, which has two parameters, *low* and *high* (see Tables 5.1 and 5.6). Here the parameters evaluate to 1 and 4 respectively, and the corresponding density function is that of the uniform distribution on the interval $(1, 4)$. So by the rule in the Figure, the expression can evaluate to any $v \in (1, 4)$ with probability density $1/3$, and to any $v \in \mathbb{R} \setminus (1, 4)$ with density 0.

$$
\begin{aligned}
\langle x, y\rangle &= \text{point with the given XY coordinates}\\
\mathit{rotate}\left(\langle x, y\rangle, \theta\right) &= \langle x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta\rangle\\
\texttt{OrientedPt}\left(v, h\right) &= \texttt{OrientedPoint with given position and heading}\\
\mathit{offsetLocal}\left(OP, v\right) &= \texttt{OrientedPt}\left([\![\texttt{OP.position}]\!] + \mathit{rotate}\left(v, [\![\texttt{OP.heading}]\!]\right), \texttt{OP.heading}\right)\\
\mathit{Disc}\left(c, r\right) &= \text{set of points in the disc centered at } c \text{ and with radius } r\\
\mathit{Sector}\left(c, r, h, a\right) &= \text{set of points in the sector of } \mathit{Disc}\left(c, r\right) \text{ centered along } h\text{, with angle } a\\
\mathit{boundingBox}\left(O\right) &= \text{set of points in the bounding box of object } O\\
\mathit{visibleRegion}\left(X\right) &= \begin{cases}
\mathit{Sector}\left([\![\texttt{X.position}]\!], [\![\texttt{X.viewDistance}]\!], \right. & \\
\qquad [\![\texttt{X.heading}]\!], [\![\texttt{X.viewAngle}]\!]\big) & X \in \texttt{OrientedPoint}\\
\mathit{Disc}\left([\![\texttt{X.position}]\!], [\![\texttt{X.viewDistance}]\!]\right) & X \in \texttt{Point}
\end{cases}\\
\mathit{orientation}\left(R\right) &= \text{preferred orientation of } R \text{ if any; otherwise } \perp\\
\mathit{uniformPoint}\left(R\right) &= \text{a uniformly random point in } R\\
\mathit{fwdEuler}\left(x, d, F\right) &= N\text{th iterate of the map } x \mapsto x + \mathit{rotate}\left(\langle 0, d/N\rangle, [\![F]\!](x)\right) \text{ on } x
\end{aligned}
$$

Figure 5.8: Notation used to define the semantics of Scenic.

$$
\textsc{Distributions}\quad \frac{\langle \mathit{params}, \sigma, \pi, \mathcal{O}\rangle \to \theta \qquad v \in \operatorname{dom} P_\theta}{\langle \mathit{baseDistribution}(\mathit{params}), \sigma, \pi, \mathcal{O}\rangle \to^{P_\theta(v)} v}
$$

Figure 5.9: Semantics of distributions. Here P_θ is the density function corresponding to the particular *baseDistribution* in the expression, with parameters θ (see Table 5.6).

Table 5.6: Probability density/mass functions for the built-in distributions.

Distribution	Domain	Parameters (θ)	$P_\theta(v)$
Uniform on interval	$\mathbb{R}$	low, high	$\mathbf{1}_{\mathit{low}<v<\mathit{high}}/(\mathit{high}-\mathit{low})$
Normal	$\mathbb{R}$	μ, σ	$\exp(-(v-\mu)^2/2\sigma^2)/\sqrt{2\pi}\sigma$
Discrete uniform	$\{v_1, \ldots, v_N\}$	—	$1/N$
Discrete weighted	$\{v_1, \ldots, v_N\}$	$w_{v_1}, \ldots, w_{v_N}$	$w_v/\sum_i w_{v_i}$

5.4.3 Object Definitions

The main step in defining the semantics of object definitions is resolving which specifiers are used to specify which properties, and what order they should be evaluated in. The basic procedure, when constructing an instance of class C using a set of specifiers S, is as follows:

1. If a property is specified (non-optionally) by multiple specifiers in S, an ambiguity error is raised.
2. The set of properties P for the new object is found by combining the properties specified by all specifiers in S with the properties inherited from the class C.
3. Default value specifiers from C are added to S as needed so that each property in P is paired with a unique specifier in S specifying it, with precedence order: non-optional specifier, optional specifier, then default value.
4. The dependency graph of the specifiers S is constructed. If it is cyclic, an error is raised.
5. The graph is topologically sorted and the specifiers are evaluated in this order to determine the values of all properties P of the new object.

The complete procedure is shown in Algorithm 5.1, which produces a list of pairs (s_i, p_i) where s_i is the ith specifier to evaluate and p_i are the properties it should be used to define. Note that p_i is a set, since specifiers can specify multiple properties: for example, if $s_1 =$ `following` *vectorField* `for` *scalar*, then p_1 will always contain `position`, but could also contain `heading` since the `following` specifier optionally specifies `heading`.

Now we can state the rule for evaluating object definitions, shown in Figure 5.10. In the premise, the first line uses Algorithm 5.1 to compute the specifier evaluation order. The subsequent lines evaluate the specifiers, each in a context where `self` has been updated with all properties assigned by earlier specifiers. Each specifier evaluates to a property assignment, i.e. a map from properties to values, as explained in Section 5.3.4. Once all properties have been computed, the new instance is created. For example, if we write

```
1 spot = OrientedPoint at 0@0, facing -90 deg    # facing East
2 Object left of spot, with width 6
```

then the definition on line 2 is evaluated as follows, with σ the current variable assignment:

1. Specifier resolution yields the list

   $$(\texttt{with width 6}, \{\texttt{width}\}), (\texttt{left of 0@0}, \{\texttt{position}, \texttt{heading}\})$$

   since `left of` depends on `width` (in fact there are further specifiers for the other properties of `Object`, like `height`, which we ignore here). Note that `left of` is assigned to specify two properties, since it optionally specifies `heading` and no other specifier takes precedence.

Algorithm 5.1 *resolveSpecifiers* (*class*, *specifiers*)

▷ **Gather all specified properties**
specForProperty ← ∅
optionalSpecsForProperty ← ∅
for all specifiers S in *specifiers* **do**
    for all properties P specified non-optionally by S **do**
        if $P \in$ dom *specForProperty* **then**
            syntax error: property P specified twice
        specForProperty $(P) \leftarrow S$
    for all properties P specified optionally by S **do**
        optionalSpecsForProperty (P).*append*(S)
▷ **Filter optional specifications**
for all properties $P \in$ dom *optionalSpecsForProperty* **do**
    if $P \in$ dom *specForProperty* **then**
        continue
    if $|$*optionalSpecsForProperty* $(P)| > 1$ **then**
        syntax error: property P specified twice
    specForProperty $(P) \leftarrow$ *optionalSpecsForProperty* $(P)[0]$
▷ **Add default specifiers as needed**
defaults ← *defaultValueExpressions* (*class*)
for all properties $P \in$ dom *defaults* **do**
    if $P \notin$ dom *specForProperty* **then**
        specForProperty $(P) \leftarrow$ *defaults* (P)
▷ **Build dependency graph**
$G \leftarrow$ empty graph on dom *specForProperty*
for all specifiers $S \in$ dom *specForProperty* **do**
    for all dependencies D of S **do**
        if $D \notin$ dom *specForProperty* **then**
            syntax error: missing property D required by S
        add an edge in G from *specForProperty* (D) to S
if G is cyclic **then**
    syntax error: specifiers have cyclic dependencies
▷ **Construct specifier and property evaluation order**
specsAndProps ← empty list
for all specifiers S in G in topological order **do**
    specsAndProps.*append*$((S, \{P \mid$ *specForProperty* $(P) = S\}))$
return *specsAndProps*

$$\textsc{Object Definitions}$$
$$\frac{\begin{array}{c} \mathit{resolveSpecifiers}(\mathit{class}, \mathit{specifiers}) = ((s_1, p_1), \ldots, (s_n, p_n)) \\ \langle s_1, \sigma[\texttt{self}/\bot], \pi, \mathcal{O}\rangle \to^{r_1} \langle v_1, \sigma_1, \pi, \mathcal{O}_1\rangle \\ \langle s_2, \sigma_1[\texttt{self}.p_1/v_1(p_1)], \pi, \mathcal{O}_1\rangle \to^{r_2} \langle v_2, \sigma_2, \pi, \mathcal{O}_2\rangle \\ \vdots \\ \langle s_n, \sigma_{n-1}[\texttt{self}.p_{n-1}/v_{n-1}(p_{n-1})], \pi, \mathcal{O}_{n-1}\rangle \to^{r_n} \langle v_n, \sigma_n, \pi, \mathcal{O}_n\rangle \\ \mathit{inst} = \mathit{newInstance}(\mathit{class}, \sigma_n[\texttt{self}.p_n/v_n(p_n)](\texttt{self})) \end{array}}{\langle \mathit{class}\ \ \mathit{specifiers}, \sigma, \pi, \mathcal{O}\rangle \to^{r_1 \cdots r_n} \langle \mathit{inst}, \sigma, \pi, \mathcal{O}_n \cup \{\mathit{inst}\}\rangle}$$

Figure 5.10: Semantics of object definitions. Here *newInstance*(*class*, *props*) creates a new instance of a class with the given property values.

2. Specifier s_1, namely `with width 6`, is evaluated in the context $\sigma_1 = \sigma[\texttt{self}/\bot]$, i.e., with `self` undefined. It evaluates to the map $v_1 = \{\texttt{width} \mapsto 6\}$.

3. Specifier s_2, namely `left of spot`, is evaluated in the context $\sigma_2 = \sigma_1[\texttt{self}.p_1/v_1(p_1)]$, where $\texttt{self.width} = 6$, since $p_1 = \{\texttt{width}\}$. It evaluates to the map $v_2 = \{\texttt{position} \mapsto \texttt{0@3}, \texttt{heading} \mapsto \texttt{-90 deg}\}$.

4. `self` is evaluated in the final context $\sigma_2[\texttt{self}.p_2/v_2(p_2)]$, where $\texttt{self.width} = 6$ as before but now also $\texttt{self.position} = \texttt{0@3}$ and $\texttt{self.heading} = \texttt{-90 deg}$, since $p_2 = \{\texttt{position}, \texttt{heading}\}$.

5. Finally, a new instance is created with the resulting property values, and is added to the set of `Objects`. The object definition itself evaluates to the new instance.

In this example there were no probabilities, but the rule in Figure 5.10 handles the general case: each specifier may evaluate with some probability density r_i (conditioned on the evaluations of the preceding specifiers), and the corresponding object is created with the joint density $\Pi_i r_i$, as we would expect.

5.4.4 Specifiers

Now we give precise definitions for each of the specifiers. As stated above, since the specifiers do not have side effects, for concision we write their meanings using the $[\![S]\!]$ notation rather than rewrite rules.

The semantics of the `position` specifiers are shown in Figure 5.11. Since these only specify a single property, we write the semantics as a vector value v: the semantics of each specifier is actually the map $\{\texttt{position} \mapsto v\}$. Thus for example $[\![\texttt{at 1@2}]\!] = \{\texttt{position} \mapsto [\![\texttt{1@2}]\!]\} = \{\texttt{position} \mapsto \langle 1, 2\rangle\}$.

Next, Figure 5.12 gives the semantics of the `position` specifiers that also optionally specify `heading`. The figure writes the semantics as an `OrientedPoint` value; if it has

$$
\begin{aligned}
\llbracket \texttt{at } V \rrbracket &= \llbracket V \rrbracket \\
\llbracket \texttt{offset by } V \rrbracket &= \llbracket V \texttt{ relative to ego.position} \rrbracket \\
\llbracket \texttt{offset along } H \texttt{ by } V \rrbracket &= \llbracket \texttt{ego.position offset along } H \texttt{ by } V \rrbracket \\
\llbracket \texttt{left of } V \rrbracket &= \llbracket \texttt{left of } V \texttt{ by 0} \rrbracket \quad \text{(likewise for } \texttt{right of}\text{, etc.)} \\
\llbracket \texttt{left of } V \texttt{ by } S \rrbracket &= \llbracket V \rrbracket + \mathit{rotate}\left(\langle -\llbracket \texttt{self.width} \rrbracket/2 - \llbracket S \rrbracket, 0 \rangle, \llbracket \texttt{self.heading} \rrbracket\right) \\
\llbracket \texttt{right of } V \texttt{ by } S \rrbracket &= \llbracket V \rrbracket + \mathit{rotate}\left(\langle \llbracket \texttt{self.width} \rrbracket/2 + \llbracket S \rrbracket, 0 \rangle, \llbracket \texttt{self.heading} \rrbracket\right) \\
\llbracket \texttt{ahead of } V \texttt{ by } S \rrbracket &= \llbracket V \rrbracket + \mathit{rotate}\left(\langle 0, \llbracket \texttt{self.height} \rrbracket/2 + \llbracket S \rrbracket \rangle, \llbracket \texttt{self.heading} \rrbracket\right) \\
\llbracket \texttt{behind } V \texttt{ by } S \rrbracket &= \llbracket V \rrbracket + \mathit{rotate}\left(\langle 0, -\llbracket \texttt{self.height} \rrbracket/2 - \llbracket S \rrbracket \rangle, \llbracket \texttt{self.heading} \rrbracket\right) \\
\llbracket \texttt{beyond } V_1 \texttt{ by } V_2 \rrbracket &= \llbracket \texttt{beyond } V_1 \texttt{ by } V_2 \texttt{ from ego.position} \rrbracket \\
\llbracket \texttt{beyond } V_1 \texttt{ by } V_2 \texttt{ from } V_3 \rrbracket &= \llbracket V_1 \rrbracket + \mathit{rotate}\left(\llbracket V_2 \rrbracket, \arctan\left(\llbracket V_1 \rrbracket - \llbracket V_3 \rrbracket\right)\right) \\
\llbracket \texttt{visible} \rrbracket &= \llbracket \texttt{visible from ego} \rrbracket \\
\llbracket \texttt{visible from } P \rrbracket &= \mathit{uniformPoint}\left(\mathit{visibleRegion}\left(P\right)\right)
\end{aligned}
$$

Figure 5.11: Semantics of the `position` specifiers, given as the value v such that the specifier evaluates to the map $\{\texttt{position} \mapsto v\}$.

`position` p and `heading` h, then the semantics of the specifier is the map $\{\texttt{position} \mapsto p, \texttt{heading} \mapsto h\}$.

Finally, Figure 5.13 gives the semantics of the `heading` specifiers. Again, we write the semantics as a single value v, with the specifier evaluating to the map $\{\texttt{heading} \mapsto v\}$.

5.4.5 Operators

To complete our discussion of Scenic's expressions, we define the semantics of the operators in Figures 5.14–5.19. As in Section 5.3.5, we omit the ordinary numerical and Boolean operators (`max`, `+`, `or`, `>=`, etc.), which have their standard meanings. We again use the $\llbracket O \rrbracket$ notation, since none of the operators have side effects.

5.4.6 Statements

Next, we define the semantics of Scenic's statements. Class and object definitions have been discussed above, while rules for the other statements are given in Figure 5.20. As can be seen from the first rule, variable assignment behaves in the standard way, updating the variable assignment σ with the new value. Parameter assignment is nearly identical, updating the global parameter assignment π.

As noted above, the `require` statement is equivalent to an *observation* in other languages, and following Claret et al. [35] we model it by allowing the "Hard Requirements" rule in

$$[\![(\texttt{in} \mid \texttt{on})\ R]\!] = \begin{cases} \texttt{OrientedPt}\,(x, [\![\mathit{orientation}\,(R)]\!](x)) & \mathit{orientation}\,(R) \neq \bot \\ \texttt{OrientedPt}\,(x, \bot) & \text{otherwise} \end{cases},$$
$$\text{where } x = \mathit{uniformPoint}\,([\![R]\!])$$
$$[\![\texttt{left of}\ O]\!] = [\![\texttt{left of (left of}\ O\texttt{)}]\!]$$
$$[\![\texttt{right of}\ O]\!] = [\![\texttt{right of (right of}\ O\texttt{)}]\!]$$
$$[\![\texttt{ahead of}\ O]\!] = [\![\texttt{ahead of (front of}\ O\texttt{)}]\!]$$
$$[\![\texttt{behind}\ O]\!] = [\![\texttt{behind (back of}\ O\texttt{)}]\!]$$
$$[\![\texttt{left of}\ OP]\!] = [\![\texttt{left of}\ OP\ \texttt{by 0}]\!] \quad \text{(likewise for } \texttt{right of}\text{, etc.)}$$
$$[\![\texttt{left of}\ OP\ \texttt{by}\ S]\!] = \mathit{offsetLocal}\,(OP, \langle -[\![\texttt{self.width}]\!]/2 - [\![S]\!], 0\rangle)$$
$$[\![\texttt{right of}\ OP\ \texttt{by}\ S]\!] = \mathit{offsetLocal}\,(OP, \langle [\![\texttt{self.width}]\!]/2 + [\![S]\!], 0\rangle)$$
$$[\![\texttt{ahead of}\ OP\ \texttt{by}\ S]\!] = \mathit{offsetLocal}\,(OP, \langle 0, [\![\texttt{self.height}]\!]/2 + [\![S]\!]\rangle)$$
$$[\![\texttt{behind}\ OP\ \texttt{by}\ S]\!] = \mathit{offsetLocal}\,(OP, \langle 0, -[\![\texttt{self.height}]\!]/2 - [\![S]\!]\rangle)$$

$$[\![\texttt{following}\ F\ \texttt{for}\ S]\!] = [\![\texttt{following}\ F\ \texttt{from ego.position for}\ S]\!]$$
$$[\![\texttt{following}\ F\ \texttt{from}\ V\ \texttt{for}\ S]\!] = \texttt{OrientedPt}\,(y, [\![F]\!](y)) \text{ where } y = \mathit{fwdEuler}\,([\![V]\!], [\![S]\!], [\![F]\!])$$

Figure 5.12: Semantics of the `position` specifiers optionally specifying `heading`. A result of o means the specifier evaluates to $\{\texttt{position} \mapsto o.\texttt{position}, \texttt{heading} \mapsto o.\texttt{heading}\}$.

$$[\![\texttt{facing}\ H]\!] = [\![H]\!]$$
$$[\![\texttt{facing}\ F]\!] = [\![F]\!]([\![\texttt{self.position}]\!])$$
$$[\![\texttt{facing toward}\ V]\!] = \arctan\,([\![V]\!] - [\![\texttt{self.position}]\!])$$
$$[\![\texttt{facing away from}\ V]\!] = \arctan\,([\![\texttt{self.position}]\!] - [\![V]\!])$$
$$[\![\texttt{apparently facing}\ H]\!] = [\![\texttt{apparently facing}\ H\ \texttt{from ego.position}]\!]$$
$$[\![\texttt{apparently facing}\ H\ \texttt{from}\ V]\!] = [\![H]\!] + \arctan\,([\![\texttt{self.position}]\!] - [\![V]\!])$$

Figure 5.13: Semantics of the `heading` specifiers, given as the value v such that the specifier evaluates to the map $\{\texttt{heading} \mapsto v\}$.

$$
\begin{aligned}
[\![\texttt{relative heading of } H]\!] &= [\![\texttt{relative heading of } H \texttt{ from ego.heading}]\!] \\
[\![\texttt{relative heading of } H_1 \texttt{ from } H_2]\!] &= [\![H_1]\!] - [\![H_2]\!] \\
[\![\texttt{apparent heading of } OP]\!] &= [\![\texttt{apparent heading of } OP \texttt{ from ego.position}]\!] \\
[\![\texttt{apparent heading of } OP \texttt{ from } V]\!] &= [\![\texttt{OP.heading}]\!] - \arctan\left([\![\texttt{OP.position}]\!] - [\![V]\!]\right)) \\
[\![\texttt{distance to } V]\!] &= [\![\texttt{distance from ego.position to } V]\!] \\
[\![\texttt{distance from } V_1 \texttt{ to } V_2]\!] &= |[\![V_2]\!] - [\![V_1]\!]| \\
[\![\texttt{angle to } V]\!] &= [\![\texttt{angle from ego.position to } V]\!] \\
[\![\texttt{angle from } V_1 \texttt{ to } V_2]\!] &= \arctan\left([\![V_2]\!] - [\![V_1]\!]\right)
\end{aligned}
$$

Figure 5.14: Semantics of scalar operators.

$$
\begin{aligned}
[\![P \texttt{ can see } V]\!] &= [\![V]\!] \in \mathit{visibleRegion}\left([\![P]\!]\right) \\
[\![P \texttt{ can see } O]\!] &= \mathit{boundingBox}\left([\![O]\!]\right) \cap \mathit{visibleRegion}\left([\![P]\!]\right) \neq \emptyset \\
[\![V \texttt{ in } R]\!] &= [\![V]\!] \in [\![R]\!] \\
[\![O \texttt{ in } R]\!] &= \mathit{boundingBox}\left([\![O]\!]\right) \subseteq [\![R]\!]
\end{aligned}
$$

Figure 5.15: Semantics of Boolean operators.

$$
\begin{aligned}
[\![S \texttt{ deg}]\!] &= [\![S]\!] \cdot \pi/180 \\
[\![F \texttt{ at } V]\!] &= [\![F]\!]([\![V]\!]) \\
[\![F_1 \texttt{ relative to } F_2]\!] &= [\![F_1]\!]([\![\texttt{self.position}]\!]) + [\![F_2]\!]([\![\texttt{self.position}]\!]) \\
[\![H \texttt{ relative to } F]\!] &= [\![H]\!] + [\![F]\!]([\![\texttt{self.position}]\!]) \\
[\![F \texttt{ relative to } H]\!] &= [\![H]\!] + [\![F]\!]([\![\texttt{self.position}]\!]) \\
[\![H_1 \texttt{ relative to } H_2]\!] &= [\![H_1]\!] + [\![H_2]\!]
\end{aligned}
$$

Figure 5.16: Semantics of heading operators.

$$\begin{aligned}
[\![V_1 \ (\texttt{relative to} \mid \texttt{offset by}) \ V_2]\!] &= [\![V_1]\!] + [\![V_2]\!] \\
[\![V_1 \ \texttt{offset along} \ H \ \texttt{by} \ V_2]\!] &= [\![V_1]\!] + \mathit{rotate}([\![V_2]\!], [\![H]\!]) \\
[\![V_1 \ \texttt{offset along} \ F \ \texttt{by} \ V_2]\!] &= [\![V_1]\!] + \mathit{rotate}([\![V_2]\!], [\![F]\!]([\![V_1]\!]))
\end{aligned}$$

Figure 5.17: Semantics of vector operators.

$$\begin{aligned}
[\![\texttt{visible} \ R]\!] &= [\![R \ \texttt{visible from ego}]\!] \\
[\![R \ \texttt{visible from} \ P]\!] &= [\![R]\!] \cap \mathit{visibleRegion}([\![P]\!])
\end{aligned}$$

Figure 5.18: Semantics of Region operators.

$$\begin{aligned}
[\![V \ \texttt{relative to} \ OP]\!] &= \mathit{offsetLocal}(OP, [\![V]\!]) \\
[\![OP \ \texttt{offset by} \ V]\!] &= [\![V \ \texttt{relative to} \ OP]\!] \\
[\![\texttt{front of} \ O]\!] &= \mathit{offsetLocal}(O, \langle 0, [\![\texttt{O.height}]\!]/2 \rangle) \\
[\![\texttt{back of} \ O]\!] &= \mathit{offsetLocal}(O, \langle 0, -[\![\texttt{O.height}]\!]/2 \rangle) \\
[\![\texttt{left of} \ O]\!] &= \mathit{offsetLocal}(O, \langle -[\![\texttt{O.width}]\!]/2, 0 \rangle) \\
[\![\texttt{right of} \ O]\!] &= \mathit{offsetLocal}(O, \langle [\![\texttt{O.width}]\!]/2, 0 \rangle) \\
[\![\texttt{front left of} \ O]\!] &= \mathit{offsetLocal}(O, \langle -[\![\texttt{O.width}]\!]/2, [\![\texttt{O.height}]\!]/2 \rangle) \\
[\![\texttt{back left of} \ O]\!] &= \mathit{offsetLocal}(O, \langle -[\![\texttt{O.width}]\!]/2, -[\![\texttt{O.height}]\!]/2 \rangle) \\
[\![\texttt{front right of} \ O]\!] &= \mathit{offsetLocal}(O, \langle [\![\texttt{O.width}]\!]/2, [\![\texttt{O.height}]\!]/2 \rangle) \\
[\![\texttt{back right of} \ O]\!] &= \mathit{offsetLocal}(O, \langle [\![\texttt{O.width}]\!]/2, -[\![\texttt{O.height}]\!]/2 \rangle)
\end{aligned}$$

Figure 5.19: Semantics of `OrientedPoint` operators.

VARIABLE/PARAMETER ASSIGNMENTS

$$\frac{\langle E, \sigma, \pi, \mathcal{O}\rangle \to \langle v, \sigma, \pi, \mathcal{O}'\rangle}{\langle x \texttt{ = } E, \sigma, \pi, \mathcal{O}\rangle \to \langle \texttt{pass}, \sigma[x/v], \pi, \mathcal{O}'\rangle}$$
$$\langle \texttt{param } x \texttt{ = } E, \sigma, \pi, \mathcal{O}\rangle \to \langle \texttt{pass}, \sigma, \pi[x/v], \mathcal{O}'\rangle$$

HARD REQUIREMENTS

$$\frac{\langle B, \sigma, \pi, \mathcal{O}\rangle \to \langle \texttt{True}, \sigma, \pi, \mathcal{O}'\rangle}{\langle \texttt{require } B, \sigma, \pi, \mathcal{O}\rangle \to \langle \texttt{pass}, \sigma, \pi, \mathcal{O}'\rangle}$$

SOFT REQUIREMENTS

$$\langle \texttt{require[}p\texttt{]} \ B, \sigma, \pi, \mathcal{O}\rangle \to^{p} \langle \texttt{require } B, \sigma, \pi, \mathcal{O}\rangle$$
$$\langle \texttt{require[}p\texttt{]} \ B, \sigma, \pi, \mathcal{O}\rangle \to^{1-p} \langle \texttt{pass}, \sigma, \pi, \mathcal{O}\rangle$$

MUTATIONS

$$\langle \texttt{mutate } obj_i \texttt{ by } s, \sigma, \pi, \mathcal{O}\rangle \to \langle \texttt{pass}, \sigma, \pi, \mathcal{O}[\sigma(obj_i)\texttt{.mutationScale}/s]\rangle$$

Figure 5.20: Semantics of statements (excluding class definitions and standard rules for sequential composition). `pass` is the no-op statement.

Figure 5.20 to only fire when the condition is satisfied (turning the requirement into a no-op). If the condition is not satisfied, no rules apply and the program fails to terminate normally. When defining the semantics of entire SCENIC programs below we will discard such non-terminating executions, yielding a distribution only over executions where all hard requirements are satisfied.

The soft requirement `require[`p`]` B is equivalent to a hard requirement `require` B which is only enforced with probability p, as explained in Section 5.3.6. This is reflected in the two corresponding rules in Figure 5.20, which convert the soft requirement to a hard requirement with probability p and otherwise convert it into a no-op.

Finally, mutations are handled by two rules. As we saw in Section 5.3.6, the `mutate` statement itself simply sets the `mutationScale` property of an object, indicating that noise should be added at the end of the program. This is accomplished by the "Mutations" rule in Figure 5.20. The noise is actually added by the first of two special rules, shown in Figure 5.21, that apply only once the program has been reduced to the no-op `pass` and so computation has finished. The rule first looks up the values of the properties `mutationScale`, `positionStdDev`, and `headingStdDev` for each object. Respectively, these specify the overall scale of the noise to add (by default zero, so that no noise is added) and factors allowing the standard deviation for `position` and `heading` to be adjusted individually. The rule then independently samples Gaussian noise with the desired standard deviations for each object and adds it to the object's `position` and `heading` properties.

The second rule in Figure 5.21 checks SCENIC's three built-in hard requirements:

TERMINATION, STEP 1: APPLY MUTATIONS

$$\frac{\begin{array}{c}\mathcal{O} = \{o_1, \ldots, o_n\} \qquad \forall i \in \{1, \ldots, n\} : \\ S_i = \mathcal{O}(o_i\texttt{.mutationScale}) \\ p_i = \mathcal{O}(o_i\texttt{.positionStdDev}) \qquad h_i = \mathcal{O}(o_i\texttt{.headingStdDev}) \\ \langle \texttt{Normal(0, } S_i \cdot p_i\texttt{)}, \sigma, \pi, \mathcal{O}\rangle \to^{r_{i,x}} \langle n_{i,x}, \sigma, \pi, \mathcal{O}\rangle \\ \langle \texttt{Normal(0, } S_i \cdot p_i\texttt{)}, \sigma, \pi, \mathcal{O}\rangle \to^{r_{i,y}} \langle n_{i,y}, \sigma, \pi, \mathcal{O}\rangle \\ \langle \texttt{Normal(0, } S_i \cdot h_i\texttt{)}, \sigma, \pi, \mathcal{O}\rangle \to^{r_{i,h}} \langle n_{i,h}, \sigma, \pi, \mathcal{O}\rangle \\ pos_i = \mathcal{O}(o_i\texttt{.position}) + \langle n_{i,x}, n_{i,y}\rangle \qquad head_i = \mathcal{O}(o_i\texttt{.heading}) + n_{i,h}\end{array}}{\langle \texttt{pass}, \sigma, \pi, \mathcal{O}\rangle \to^{r_{0,x} r_{0,y} r_{0,h} \cdots} \langle \textsc{Done}, \sigma, \pi, \mathcal{O}[o_i\texttt{.position}/pos_i][o_i\texttt{.heading}/head_i]\rangle}$$

TERMINATION, STEP 2: CHECK DEFAULT REQUIREMENTS

$$\frac{\begin{array}{c}\mathcal{O} = \{o_1, \ldots, o_n\} \qquad \forall i : \mathit{boundingBox}\,(o_i) \subseteq o_i\texttt{.regionContainedIn} \\ \forall i \neq j : \big(o_i\texttt{.allowCollisions} \vee o_j\texttt{.allowCollisions} \\ \vee \mathit{boundingBox}\,(o_i) \cap \mathit{boundingBox}\,(o_j) = \emptyset\big) \\ \forall i : \neg o_i\texttt{.requireVisible} \vee \langle \texttt{ego can see } o_i, \sigma, \pi, \mathcal{O}\rangle \to \texttt{True}\end{array}}{\langle \textsc{Done}, \sigma, \pi, \mathcal{O}\rangle \to (\pi, \mathcal{O})}$$

Figure 5.21: Semantics of program termination. DONE denotes a special state ready for the final termination rule to run.

1. Each `Object` must lie inside its container Region (by default, the whole workspace).
2. No two `Objects` can intersect, unless their `allowCollisions` property is true.
3. Every `Object` must be visible from `ego`, unless its `requireVisible` property is false.

If these conditions are satisfied, the program terminates, with its output being a *scene*: the set of objects $\mathcal{O}$, together with the global parameters π.

5.4.7 Programs

We can finally define the semantics of an entire SCENIC program. As we have seen above, the rewrite rules in an execution trace of a SCENIC program are annotated with probabilities, since SCENIC allows sampling from distributions. This induces a distribution over executions of the program, where each execution trace has a probability (density) given by the product of the probabilities of all its rewrite rules (see e.g. [35]). If the the program were purely imperative, with no hard or soft requirements, this would induce a distribution over outputs of the program, i.e. scenes.

However, we want a scene distribution which is *conditioned* on all hard requirements being satisfied. Towards this end, we set up the rules above (Figures 5.20 and 5.21) so that executions where a requirement is violated do not terminate, following Claret et al.

[35]. Therefore, if we discard all non-terminating traces, normalizing the probabilities of the remaining traces yields a distribution over scenes that satisfy all requirements[2]. This is the distribution defined by the SCENIC program.

5.5 Scene Improvisation

Now that we have defined the SCENIC language, and the semantics of a SCENIC program as a distribution over scenes, the question remains of how to sample from this distribution. This problem, which we call *scene improvisation*, does fall under the general umbrella of algorithmic improvisation, being a synthesis problem with all three essential elements:

Hard constraints given explicitly as `require` statements, as well as arising from the imperative part of the program, which specifies required relationships between parts of the scene (e.g. `x = y offset by 5@0`).

Soft constraints from soft requirements, like `require[0.5] distance to taxi < 10`.

Randomness constraints arising from the distributions imposed by the program; for example `x = (-3, 3)` constrains `x` to have a certain distribution.

However, scene improvisation goes beyond the theory we studied in Chapter 3 in two ways: first, the space of improvisations is (partly) continuous. Second, the randomness constraint is far more sophisticated, allowing particular distributions to be specified rather than simply requiring any distribution which is sufficiently close to uniform. Extending the theory of control improvisation to handle such problems is an interesting direction for future work.

Instead, we consider scene improvisation from the perspective of probabilistic programming languages. Since our semantics reduces soft requirements to hard requirements, scene improvisation is essentially a special case of the sampling problem for imperative PPLs with observations. A large variety of heuristic sampling techniques have been developed for this general problem. The simplest approach is *rejection sampling*: repeatedly sample from the program without regard to the requirements, until a sample satisfying the requirements is found. This procedure yields exactly the desired distribution, but could take arbitrarily long to produce a sample: indeed, if the requirements are inconsistent or have probability zero, the procedure will run forever. However, rejection sampling does have the advantage of being able to handle arbitrary requirements, and in practice it worked well for all of the scenarios in our case study in Chapter 8. It should also be possible to adapt more sophisticated sampling techniques used for PPLs, such as Markov chain Monte Carlo methods (see, e.g., [123, 130, 188, 40]).

[2]Putting this on a rigorous theoretical foundation requires measure theory, since we allow sampling from both discrete and continuous distributions (although in practice all continuous distributions are approximated by discrete distributions over floating-point numbers). For a detailed discussion see Borgström et al. [19] or Staton et al. [163].

A more interesting question is whether we can take advantage of the domain-specific nature of SCENIC to develop *specialized* sampling techniques not available in general-purpose PPLs. Below, we present several such techniques which can significantly improve the efficiency of sampling.

5.5.1 Domain-Specific Sampling Techniques

Our specialized sampling techniques take advantage of several domain-specific aspects of SCENIC:

- Direct syntax for geometric operations, which allows us, for example, to easily detect when a point is being sampled uniformly within a polygon (as in `Car on road`; if the position of the `Car` were an arbitrary arithmetic expression, it would be difficult to tell whether it was equivalent to a uniform sample over a polygon).
- Built-in requirements of a known form, e.g. visibility and object containment requirements (again, detecting that arbitrary arithmetic predicates had one of these forms would be difficult).
- Lack of probabilistic control flow, meaning that the structure of all distributions needing to be sampled are known at compile time (whereas in a general-purpose PPL, the position of a `Car` for example could be uniform over a polygon in one execution path but Gaussian in another).

Our techniques are inspired by methods for low-dimensional robotic path planning: in particular, the reduction of planning for a robot of nonzero size to planning for a point robot by constructing the *configuration space* [186]. For example, given a 2D map of an environment, dilating all obstacles by some $r > 0$ yields a new map in which all free space is at least distance r from the original obstacles. So planning a route through the free space of the new map yields a path that is collision-free for a circular robot of radius r. In a similar way, we can use geometric information in SCENIC to prune away parts of the sample space where the objects in the scene do not fit into the workspace or otherwise violate a requirement. We describe three different instantiations of this idea below.

Pruning Based on Containment

The simplest technique applies to any object X whose position is uniform in a region R and which must be contained in a region C (e.g. the road in our case study). If *minRadius* is a lower bound on the distance from the center of X to its bounding box, then we can restrict R to $R \cap \text{erode}(C, \textit{minRadius})$. This is sound, since if X is centered anywhere not in the restriction, then some point of its bounding box must lie outside of C.

Pruning Based on Orientation

The next technique applies to scenarios placing constraints on the relative heading and the maximum distance M between objects X and Y, which are oriented with respect to a vector field that is constant within polygonal regions (e.g. roads, either straight or approximated by polygonal segments). For each polygon P, we find all polygons Q_i satisfying the relative heading constraints with respect to P (up to a perturbation if X and Y need not be exactly aligned to the field), and restrict P to $P \cap \text{dilate}(\cup Q_i, M)$. This is also sound: suppose X can be positioned at x in polygon P. Then Y must lie at some y in a polygon Q satisfying the constraints, and since the distance from x to y is at most M, we have $x \in \text{dilate}(Q, M)$.

Algorithm 5.2 *pruneByOrientation* (*map, A, M, δ*)

```
map' ← ∅
for all polygons P in map do
    for all polygons Q in map do
        Q' ← dilate(Q, M)
        if P ∩ Q' ≠ ∅ ∧ relHead(P, Q) ± 2δ ∈ A then
            map' ← map' ∪ (Q' ∩ P)
return map'
```

Pruning Based on Diameter

Finally, in the setting above of objects X and Y aligned to a polygonal vector field (with maximum distance M), we can also prune the space using a lower bound on the width of the configuration. For example, in our bumper-to-bumper scenario we can infer such a bound from the `offset by` specifiers in the program. We first find all polygons that are not wide enough to fit the configuration according to the bound: call these "narrow". Then we restrict each narrow polygon P to $P \cap \text{dilate}(\cup Q_i, M)$ where Q_i runs over all polygons except P. To see that this is sound, suppose object X can lie at x in polygon P. If P is not narrow, we do not restrict it; otherwise, object Y must lie at y in some other polygon Q. Since the distance from x to y is at most M, as above we have $x \in \text{dilate}(Q, M)$.

Algorithm 5.3 *pruneByWidth* (*map, M, minWidth*)

```
narrowPolys ← narrow(map, minWidth)
map' ← map \ narrowPolys
for all polygons P in narrowPolys do
    U ← ⋃_{Q ∈ map \ {P}} dilate(Q, M)
    map' ← map' ∪ (P ∩ U)
return map'
```

5.6 Summary and Future Work

In this chapter, we developed Scenic, a probabilistic programming language for specifying distributions over *scenes*, configurations of physical objects and agents. We explained how the language's domain-specific design makes it possible to readably and concisely encode complex scenarios like those arising in the environment of an autonomous car. In particular, we saw how Scenic's *specifiers* provide a convenient, modular way to define the positions and orientations of objects with the flexibility of natural language. More broadly, Scenic's mixture of imperative syntax with declarative hard and soft requirements makes it easy to build a scenario in a forward, incremental manner while being able to impose global constraints on the final result.

We formally defined the semantics of a Scenic program as a distribution over scenes, and studied the problem of sampling from this distribution, *scene improvisation*. We presented several techniques which take advantage of the specialized structure of Scenic programs to prune away infeasible parts of the scene space and thereby improve the efficiency of sampling. As we will see in Chapter 8, our implementation [65] is able to generate complex, realistic scenes that can be used to improve the performance of actual perception systems for autonomous cars, as well as for various other applications.

Beyond exploring additional domains and applications of Scenic, there are a number of other promising directions for future work:

Extending the Language. Most straightforwardly, there are several ways the syntax of Scenic could be extended to make it more general and easier to use, including adding support for 3D scenes, user-defined specifiers, and restricted types of probabilistic control flow that would not compromise sampling efficiency.

Encoding Dynamics. A much more substantial generalization of the language would be to allow encoding of dynamic scenarios, where agents move and take actions over time. As we will see in Chapter 8, although Scenic describes static scenes, it is not limited to testing perception systems operating on single images: we can test controllers, for example, by generating initial states and controller parameters from which to launch simulations. However, being able to express dynamic behaviors, e.g. one car passing another on a freeway, would greatly increase the utility of Scenic for developing complex controllers as used in autonomous driving. This will require extending the language with constructs for time, events, and reactive agents.

Improving Sampling. As we mentioned above, rejection sampling gave acceptable performance for all of our experiments in our case study in Chapter 8. However, it is easy to write Scenic programs where rejection sampling fails, and it would be interesting to explore whether more sophisticated PPL techniques like Markov chain Monte Carlo methods [123, 130, 188, 40] can help in such cases. MCMC can only help to a certain extent, though, since even finding a single solution to the constraints in a Scenic program can be hard: for example, a scenario like

```
1 for i in range(30):
2     Car in largeCongestedIntersection
```

essentially requires solving a packing problem, and even very simple geometric packing problems are already NP-hard [61, 98]. It is possible that constraint solvers, and in particular uniform sampling algorithms like those we discussed in Chapter 2, could help here, although to our knowledge there has been no work on sampling from nonlinear real arithmetic constraints like those arising from Scenic programs.

Learning Scenarios from Data. A related issue is that manually writing a set of Scenic programs adequate for training or testing purposes could be a significant burden, given the wide variety of possible scenarios a system like an autonomous car can encounter. One way to reduce this burden would be to automatically learn Scenic programs from data. There are many problems which would need to be solved to do this in practice, e.g. clustering scenes into related groups, detecting outliers, identifying when a set of scenarios is "adequate" in some sense, doing scene understanding to generate scenes from raw data in the first place, and so forth; however, even inferring a single Scenic program from a set of scenes is already theoretically interesting. For this, we could build on a long line of work on parameter synthesis for probabilistic programs (going back to IBAL [134]), as well as more recent work on learning entire programs (e.g. Saad et al. [150], which also discusses many earlier approaches). There is also related work on *specification mining* (see for example Ammons et al. [4] and Li et al. [107]), which we have previously used for the application of CI to music improvisation [177].

Monitoring Scenarios. Finally, since Scenic programs can be used as specifications of the nominal environments a system is designed to operate in, a natural question is whether we can *monitor* a scenario to detect anomalous situations at runtime. For example, we could formalize the so-called "operational design domain" (ODD) of an autonomous vehicle [167] as a Scenic program, and if we detect that we have left the ODD, we could take some emergency action to try to recover. A first step would be to compute the likelihood of the current scene given a Scenic program, but a more sophisticated approach will probably be required to prevent frequent false alarms due to uncertainty about the environment, noisy sensor data, and more generally the inevitable mismatch between the formal Scenic model and the actual world.

Part II

Applications

Chapter 6

Robotic Planning

6.1 Introduction

In this chapter, we explore applications of algorithmic improvisation to randomized robotic planning problems. As we saw in the Introduction, it can be very useful to introduce randomness into the behavior of a robot: for example, doing a random walk can help find a target in an unknown space without needing to build a map [182]. Another interesting class of examples are *surveillance* problems, where the goal is to visit several designated locations, using randomness to make it more difficult for an adversary to predict where the robot will be at any given time[1]. For example, consider a security robot which must periodically visit every room in a museum, without running into any human staff members or visitors. The planning task is to compute a *policy* for the robot which tells it how to move over time as a function of its environment, in such a way that the goals of visiting every room and avoiding collisions are guaranteed to be met.

As we discussed in Chapter 4, we could formulate this task as a reactive synthesis problem [101], but a reactive synthesis algorithm might well generate a completely deterministic policy. In such a case, a thief could easily observe at what time the robot tends to be at the opposite end of the museum, and plan accordingly. Alternatively, rather than a general planning algorithm, we could use specialized techniques for surveillance problems: there is a substantial literature on *robotic patrolling* [137]. In particular there are patrolling strategies which use random walks to reduce predictability [81, 153].

However, patrolling strategies suffer from another problem, namely that because they are specialized to an idealized patrolling problem, they do not support imposing constraints on the robot's behavior that are often necessary in practice. The collision-avoidance constraint above is relatively easy to handle, e.g. doing a random walk globally and avoiding collisions locally; however, other useful constraints can force major changes to the robot's policy at a global level. For example, suppose a surveillance drone has a limited battery life, and must return to its home base to recharge after at most some time T. Patrolling algorithms

[1] Thanks to Stéphane Lafortune for suggesting this application.

based on random walks, for example, cannot ensure such a requirement, since there will be some probability of not returning in time. We could modify the algorithm to account for the battery, for example switching into a "return to base" mode when the battery reaches a certain threshold, but this could compromise unpredictability.

These types of planning problems need algorithms which can combine arbitrary functional correctness requirements with a randomness guarantee: this is exactly what algorithmic improvisation provides. When the robot operates in a known environment, and planning can be done offline, we can formulate the planning problem as an instance of control improvisation (Chapter 3): synthesize an improviser which generates random sequences of actions for the robot, subject to constraints. If some aspects of the environment are uncertain or even adversarial, so that planning must be done on-the-fly in response to the environment, we can instead formulate the planning problem as an instance of *reactive* control improvisation (Chapter 4): the improviser is now a *strategy* (policy) which guarantees the constraints under all possible environment behaviors.

We will illustrate both of these cases with worked examples of robotic surveillance problems, generating routes for a surveillance drone subject to safety, efficiency, and randomness constraints. Section 6.2 presents an offline planning example using control improvisation, and Section 6.3 presents an online planning example using reactive control improvisation. In both cases we show how to encode our desired requirements as DFAs, enabling the use of the efficient improvisation schemes for DFAs we developed in Chapters 3 and 4. We then perform experiments using the resulting improvisers to control simulated and real drones, demonstrating the practicality of algorithmic improvisation for robotic planning tasks. Finally, we conclude in Section 6.4 with a summary and directions for future work.

6.2 Planning in a Known Environment

Our first example will be a patrolling problem for a surveillance drone with a limited battery life. Specifically, supposing there are ℓ designated locations to be monitored, our requirements are as follows:

Mission:

1. The drone must visit all ℓ locations at least once.
2. The drone must not visit any location twice in a row (since this would be redundant).
3. The drone must end at the home base (so we can start another patrol route).

Safety: The drone must not visit more than 3 locations before recharging at the home base.

Efficiency: At least 80% of the time, each location is visited exactly once.

6.2.1 Encoding as a CI Problem

When formalizing the planning task as a CI problem, we can choose how precisely to encode the geometry of the workspace the robot operates in. At one extreme, we can discretize the workspace into a very fine grid or graph, and generate paths through this graph. This has the advantage of allowing the improviser to control the robot in a detailed way, maximizing the ability to introduce randomness into the system; however, the CI problem may be difficult to solve since the space of possible paths and the encodings of the constraints are large. At the other end of the spectrum, we can throw away all the details of the workspace, encoding only the connectivity between different destinations in the CI problem. This will greatly reduce the size of the CI problem, but the improviser will now only generate a high-level plan, which will need to be refined into a detailed plan by some other algorithm. We will take this second approach in this section, and illustrate a CI encoding with more precise geometry in Section 6.3.

Thus, we will use the alphabet $\Sigma = \{0, \ldots, \ell\}$, where 0 denotes the home base, $i > 0$ denotes the ith designated location, and a word in Σ^* corresponds to a sequence of locations visited by the drone. For example, the word 0120 means a route where starting from the home base, the drone visits locations 1 and 2 in sequence, then returns. Note that we are not saying anything about *how* the drone gets from location 1 to location 2: that will have to be decided by a lower-level planner. Our improviser will only determine the *order* in which to visit different locations.

Next, we encode the mission and safety requirements above as the hard constraint of our CI instance. Our hard specification $\mathcal{H}$ will be a DFA which is the product of three automata $\mathcal{H}_{\text{visit}}$, $\mathcal{H}_{\text{repeat}}$, and $\mathcal{H}_{\text{recharge}}$ encoding different requirements:

1. *The drone must visit every location.* To encode this as a DFA $\mathcal{H}_{\text{visit}}$, we use one state for every possible combination of having visited or not visited a location (other than the home base, which we start from). So the state space is $\{\text{T}, \text{F}\}^\ell$, where for example with $\ell = 3$ the state TFT means we have visited locations 1 and 3 but not 2. Transitions update the state in the obvious way, recording which location has been newly visited, if any: so from state TFT on input 2 we transition to TTT, whereas on input 1 we stay at TFT. We begin in state F^ℓ, since no locations have been visited yet, and the only accepting state is T^ℓ, since then all locations have been visited. The resulting automaton for $\ell = 3$ is shown in Figure 6.1.

2. *The drone must not visit any location twice in a row, and must end at the home base.* This property is even easier to encode as a DFA $\mathcal{H}_{\text{repeat}}$: we have one state for each location, representing where the drone currently is. On input x we transition to the corresponding state, unless we are already at state x, in which case we transition to a failure state. The only accepting state is that corresponding to the home base. This automaton is shown for $\ell = 3$ in Figure 6.2.

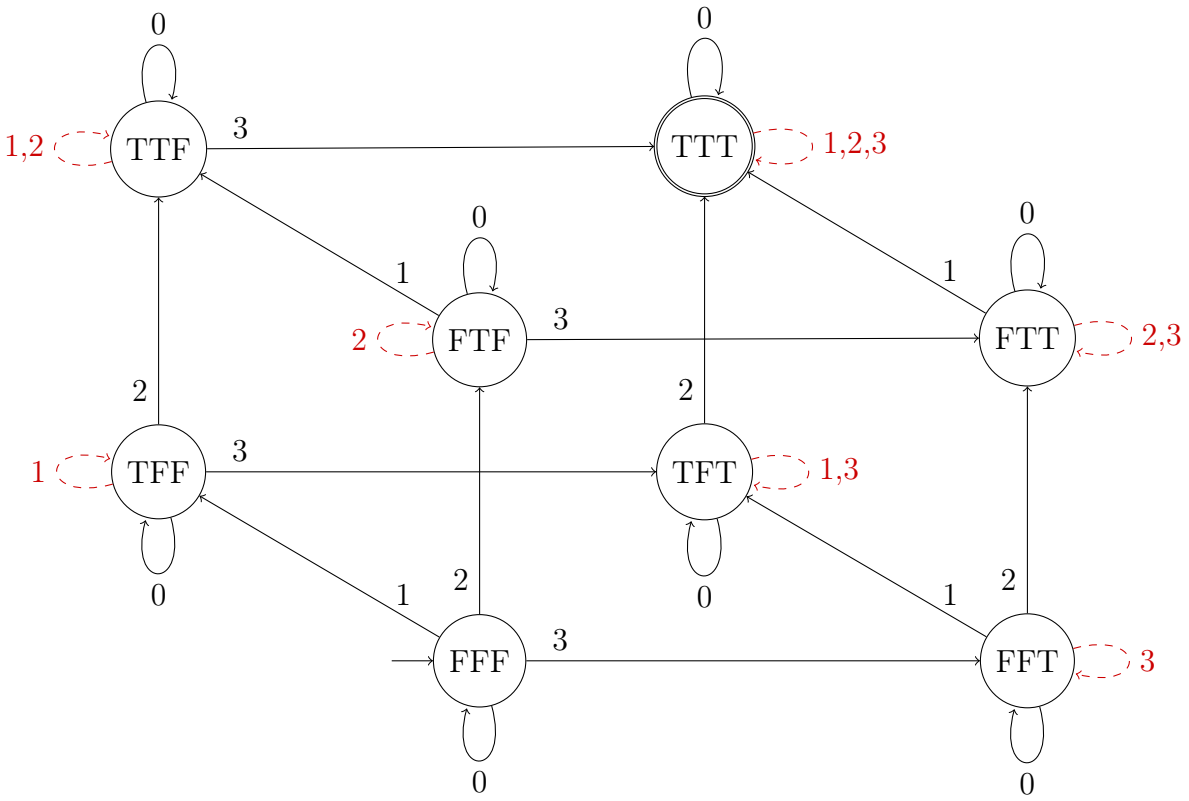


Figure 6.1: DFA requiring that all locations must be visited, for $\ell = 3$. With the dashed transitions instead leading to a failure state, requires that all locations are visited exactly once.

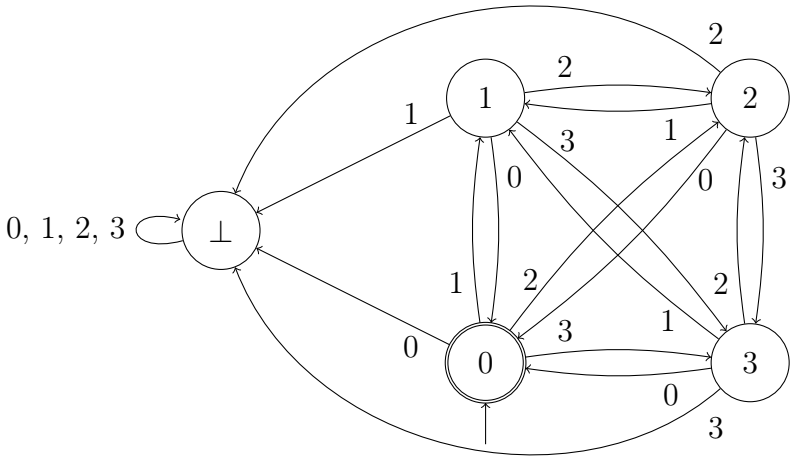


Figure 6.2: DFA requiring that no location should be visited twice in a row, and that we must end at the home base, for $\ell = 3$.

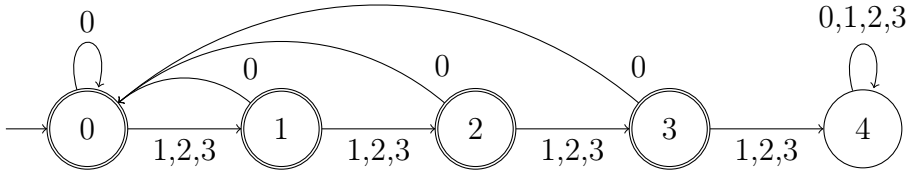


Figure 6.3: DFA requiring that we must recharge after at most 3 visits, for $\ell = 3$.

3. *The drone must not visit more than 3 locations before recharging at the home base.* To encode this as a DFA $\mathcal{H}_{\text{recharge}}$, we use a chain of states representing how long it has been since the last time we visited the home base. The chain terminates at a failure state corresponding to one time step beyond the required limit (here, 4). All other states transition to the next state in the chain on any input other than 0, which brings us back to the start of the chain since we have returned to the home base. The automaton for $\ell = 3$ is shown in Figure 6.3.

This leaves the efficiency requirement, namely that every location (except the home base) is visited exactly once at least 80% of the time. We encode this using the soft constraint of the CI instance, setting $\epsilon = 0.2$ and letting our soft specification $\mathcal{S}$ be a DFA monitoring the "visit exactly once" property. This DFA is a slight variant of $\mathcal{H}_{\text{visit}}$, simply changing all transitions which represent a repeat visit to instead transition to a failure state: see the dashed transitions in Figure 6.1.

The last component of the CI instance is the randomness requirement. Since we want an unpredictable patrol route for the drone, we want a maximally-randomized policy, i.e. an improviser with the probability bound ρ as small as possible (given the hard and soft constraints above). Rearranging Corollary 3.1 (with $\lambda = 0$), the optimal value is $\rho_{\text{opt}} = \max(1/|I|, (1-\epsilon)/|A|)$, which our algorithm can compute[2]. We can then apply the improvisation scheme of Theorem 3.4 to synthesize an improviser.

6.2.2 Experiments

With colleagues in the TerraSwarm project[3], we tested our improviser on an actual drone, shown in Figure 6.4. To do this, the sequence of locations to visit generated by the improviser was given to the DRONA motion planner [42], which refined the high-level plan into a trajectory taking into account the obstacles in the workspace. We then used the SatEX satisfiability modulo convex programming (SMC) solver [157] to reason about the dynamics of the drone and find control actions guaranteeing that the drone would track this trajectory

[2]Although there is no need to: our generic improvisation scheme (Theorem 3.2) only uses ρ to check feasibility and then compute the best possible ϵ. So if ϵ is fixed and we want the smallest feasible ρ, the algorithm is equivalent to simply sampling uniformly from $I \setminus A$ with probability ϵ and otherwise from A.

[3]Ankush Desai, Brent Schlotfeldt, Yasser Shoukry, and Dinesh Thakur. https://terraswarm.org

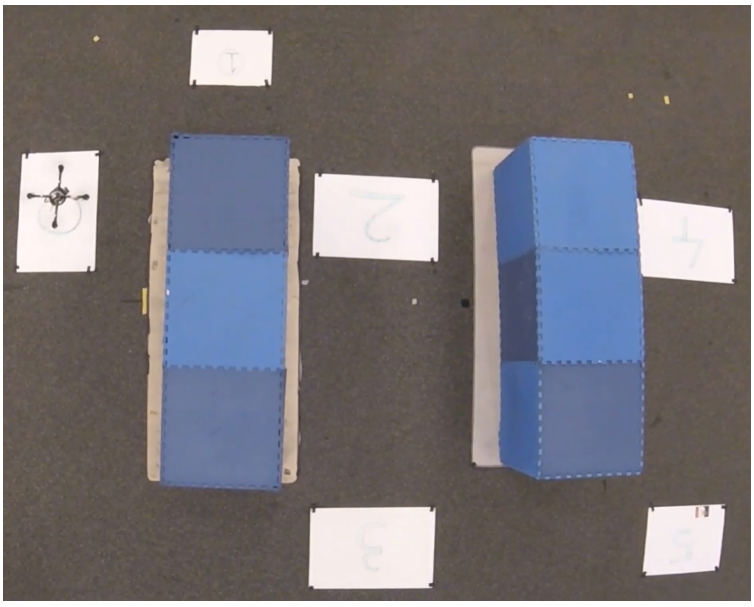


Figure 6.4: The environment and drone used to test our improviser. The white areas indicate the locations to be visited, with the drone starting at its home base on the left.

within a desired tolerance (modulo assumptions on the possible disturbances in the environment). Executing these actions, the drone successfully carried out the surveillance task in a randomized way, without colliding with obstacles[4].

To give a sense of how practical our algorithm is, Table 6.1 shows its performance on various instances of the patrolling problem as a function of the number of designated locations ℓ and the length bound n. To provide a measure of how large the automata involved are, the "Explored DFA States" column gives the number of states explored[5] of the DFA $\mathcal{B}$ representing the inadmissible improvisations, which is the largest automaton constructed by the algorithm of Theorem 3.4. We also show the synthesis time, i.e. the time needed to construct the improviser, and the average time to run the improviser and generate an improvisation. These runtimes were obtained on a MacBook Pro laptop, using a Python implementation of our algorithms; the implementation could undoubtedly be optimized, but these runtimes still give a general idea of the scalability of our algorithm.

The first part of Table 6.1 shows that the synthesis time grows exponentially with ℓ, which is as expected since the DFA $\mathcal{H}_{\text{visit}}$ has 2^ℓ states. The instance we used in our experiments above, $(\ell, n) = (5, 12)$, is solved in under a second, while we can solve instances with up to 12 locations in a few minutes. In all cases, once the improviser is constructed the time to run it and generate a plan is negligible, being on the order of tens of microseconds. The

[4] A video of this experiment is available at https://alum.mit.edu/www/dfremont/impro.html.

[5] More precisely, state-time pairs, since we need to compute the number of accepting paths from each state at every time up to the length bound n. Equivalently, the table shows the number of entries in the memoization table of the DFA path counting algorithm [86] which needed to be computed.

Locations (ℓ)	Maximum Length (n)	Explored DFA States	Synthesis Time (s)	Improvisation Time (μs)
3	8	136	0.031	14
5	12	1,490	0.38	26
7	16	11,254	3.7	42
9	20	72,018	27	65
11	24	419,800	220	95
7	18	14,154	3.9	44
7	22	19,954	5.2	47
7	26	25,754	6.1	51
7	30	31,554	7.2	56
7	34	37,354	8.3	64
7	100	133,054	28	150
7	200	278,054	63	290
7	300	423,054	110	440

Table 6.1: Performance of our DFA improvisation scheme on various patrolling problems. The improvisation times were averaged over 10,000 improvisations.

second part of the table fixes $\ell = 7$ and shows that synthesis and improvisation time grow linearly with the length bound n. Finally, the last part of the table demonstrates that our algorithm is able to handle quite large n, with the improvisation time being correspondingly larger since we need to generate long words, but even for $n = 300$ still being less than a millisecond.

6.3 Planning in an Uncertain Environment

When the environment is not known ahead of time, for example in the presence of other agents, we can use *reactive* control improvisation (Chapter 4) to find a strategy for the robot which accomplishes the desired task regardless of what environment is ultimately encountered. As usual in reactive synthesis, such a strategy likely only exists if we make some assumptions about the environment, and these assumptions have to be encoded into the RCI problem. We will illustrate this using a similar patrolling problem to the one above, adding a second drone in the same workspace which is not under our control and which we must avoid colliding with [67]. Specifically, we will use the following requirements:

Mission: The patrolling drone must visit all ℓ locations at least once.

Safety: The patrolling drone must not collide with the adversary drone.

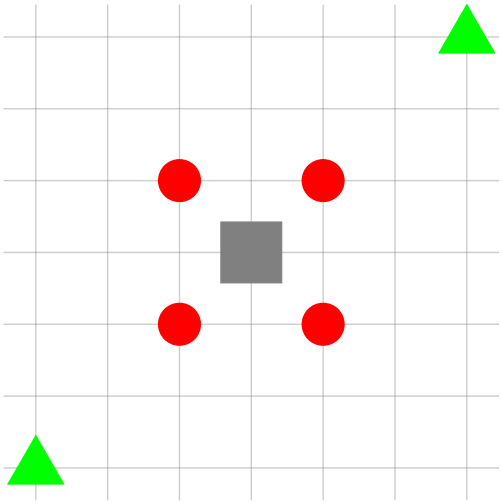


Figure 6.5: Grid world used for our RCI robotic surveillance experiments, with $k = 7$. The patrolling drone starts at bottom left, the adversary at top right (shown as triangles). The locations to visit are circled, and the adversary may not enter these or the square.

Efficiency: At least 75% of the time, each location is visited exactly once.

6.3.1 Encoding as an RCI Problem

We will use the workspace in Figure 6.5, where we control the drone in the bottom left and the potentially-adversarial drone is in the top right. There are $\ell = 4$ locations to visit in the Figure, shown as circles. In contrast to our example in Section 6.2, here we will model the geometry of the workspace, discretizing the map into a $k \times k$ grid: the lines in the Figure show the grid for $k = 7$. Our alphabet Σ will consist of the 4 possible movement directions on this grid, so that the improviser will generate a sequence of actions for the drone.

Note that with the workspace in Figure 6.5, the planning problem above is not realizable (as our algorithm will detect): the adversary can simply sit on one of the circled locations and prevent us from visiting it. To make the problem solvable, we will assume that the adversary cannot move onto the circled locations, or onto the square in the center of the workspace (otherwise, the adversary can prevent us reaching the far circle)[6].

Our hard specification $\mathcal{H}$ will be built from two DFAs encoding the safety and mission requirements:

1. *The patrolling drone must not collide with the adversary.* To monitor this property, we need to keep track of the current positions of both drones, as a function of the actions

[6]For our experiments with larger grid sizes k, we generalize the workspace as follows: if the bottom left corner of the grid is $(0, 0)$, then the 4 locations to visit are (L, L), (L, H), (H, L), and (H, H), where $L = \lfloor k/3 \rfloor$ and $H = \lfloor 2k/3 \rfloor$. The adversary is assumed to not be able to move to these grid points, as well as the interior of the square they define.

they have taken so far. Letting G be the set of all points in the grid, our state space is $G \times G \times \{1, 2\}$, where the three components respectively represent the positions of both players and whose turn it is. Since there are k^2 points in the grid, we have $2k^4$ states in total.

Transitions update the positions of the players in the natural way: given a movement action $m \in \Sigma$, based on whose turn it is (stored in the third component) we add the appropriate offset to either the first or second component of the state. For example, in state $((1, 2), (4, 3), 1)$, indicating the patrolling drone is at position $(1, 2)$, the adversary is at $(4, 3)$, and it is the patrolling drone's turn, suppose the input symbol is *North*. This causes the patrolling drone to move to position $(1, 3)$, so we transition to the state $((1, 3), (4, 3), 2)$: note the last component is 2, since it is now the adversary's turn.

To model the assumption on the adversary and the non-collision requirement, we modify these transitions slightly. If we would ever transition to a state where the adversary is in a position it is not allowed to enter, we instead transition to an accepting state with a self-loop. Otherwise, if we would transition to a state where both drones are at the same position, we instead transition to a failure state. The resulting automaton $\mathcal{H}_{\text{collision}}$ accepts exactly those traces where either the adversary violates its assumption, or the two drones do not collide.

2. *The patrolling drone must visit each of the designated locations.* Recall that the DFA $\mathcal{H}_{\text{visit}}$ of Figure 6.1 monitors exactly this property, when the input denotes which of the designated locations we are currently visiting (and is 0 when we aren't visiting any of them). However, in our setup here, the input is a movement action, not a global position. Fortunately, $\mathcal{H}_{\text{collision}}$ above already maintains the drones' positions as a function of the trace. If we view $\mathcal{H}_{\text{collision}}$ as a finite state transducer whose output is $i \in \{1, \ldots, \ell\}$ when the patrolling drone visits the corresponding designated location, and 0 otherwise, then composing this transducer with $\mathcal{H}_{\text{visit}}$ yields our desired DFA $\mathcal{H}$: it accepts exactly those traces where either the adversary violates its assumption, or the patrolling drone visits all the designated locations without colliding with the adversary. Recalling that $\mathcal{H}_{\text{visit}}$ has 2^ℓ states if there are ℓ locations to visit, $\mathcal{H}$ has $2k^4 \cdot 2^\ell$ states (plus the two special success and failure states).

To encode the efficiency requirement, that at least 75% of the time each location is visited exactly once, we put $\epsilon = 0.25$ and use a soft specification $\mathcal{S}$ given by a DFA very similar to $\mathcal{H}$. We simply alter $\mathcal{H}_{\text{visit}}$ in the same way as in Section 6.2, changing transitions corresponding to repeat visits so that they lead to a failure state (as shown in Figure 6.1).

Finally, once again we want to generate a maximally-randomized strategy for the patrolling drone, so we compute the smallest feasible ρ for our constraints above. Rearranging Corollary 4.1, we find $\rho_{\text{opt}} = \max(1/W(I), (1 - \epsilon)/W(A))$. We can then apply the improvisation scheme of Theorem 4.4 to synthesize an improviser for any desired trace length n.

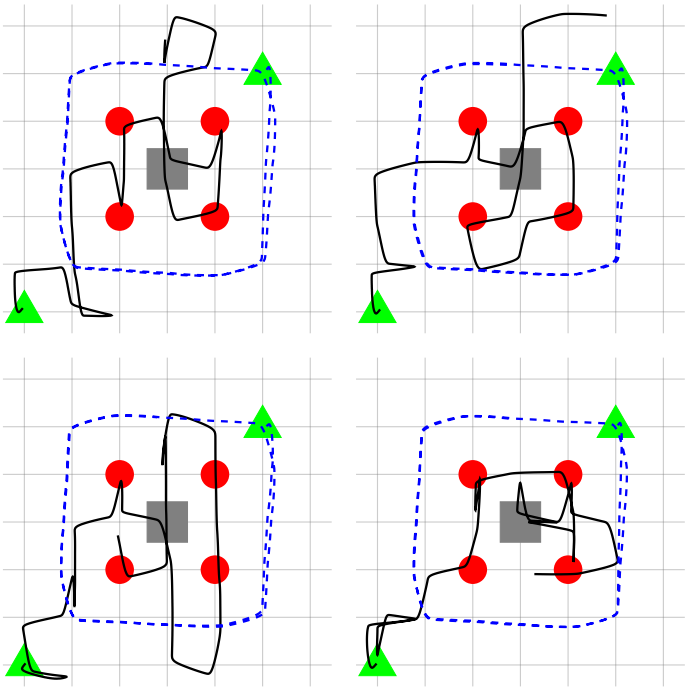


Figure 6.6: Simulations of our improviser (solid) avoiding a looping adversary (dashed).

6.3.2 Experiments

We tested our improviser for $(k, n) = (7, 60)$ in the Gazebo simulator [99], using the PX4 autopilot [121] to refine the generated routes in the grid world into control actions for the drones[7]. We did experiments with two types of adversaries: one that moves in a fixed loop, and one that actively tries to pursue the patrolling drone, moving towards it whenever possible (i.e. when this would not violate the assumption on the adversary). Simulations with these adversaries are shown from a top-down perspective in Figures 6.6 and 6.7 respectively (notice that the routes deviate from our idealized grid, because we simulate the actual continuous dynamics of the drones in 3D space). In every simulation, the patrolling drone successfully visits the 4 designated locations without colliding with the adversary[8]. Furthermore, as we see in Figure 6.6, the improviser is highly randomized, giving different routes even when the adversary behaves in the same way each time.

Finally, we do an experiment to study the performance of our RCI algorithm. We already saw in Section 6.2 that the sizes of the specification DFAs grow exponentially with the number of locations needing to be visited, so we will keep this fixed at 4 and instead study performance as a function of the grid world size k and the game length n. Our results are shown in Table 6.2. Here the "DFA States" column shows the number of states of $\mathcal{H}$

[7]Thanks to Ankush Desai and Tommaso Dreossi for assistance running the simulations.

[8]For videos of the simulations (which make it clearer that the two drones are never in the same place at the same time), see https://alum.mit.edu/www/dfremont/impro.html.

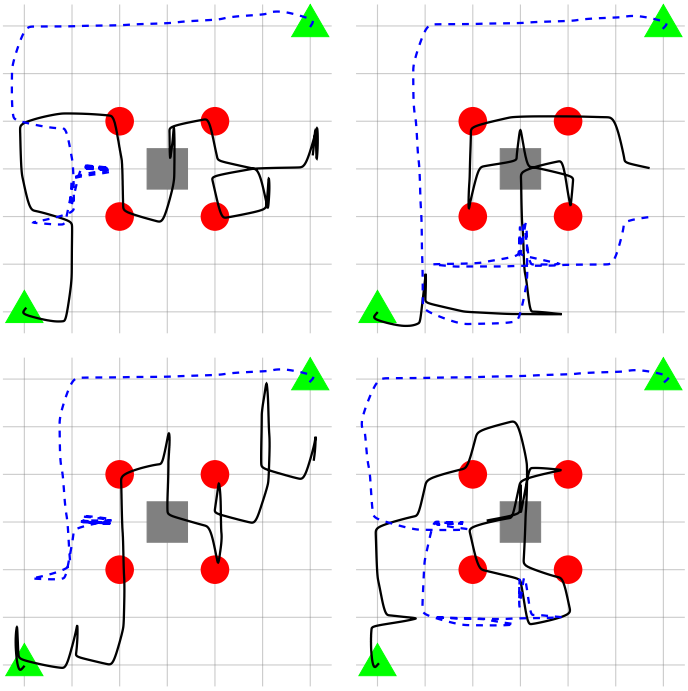


Figure 6.7: Simulations of our improviser (solid) avoiding a pursuing adversary (dashed).

and $\mathcal{S}$ (which are equal). As in our CI experiments above, the last two columns give the time to synthesize the improviser and the average time to run it and generate an improvisation (against the looping adversary described above). Again, these runtimes are from an unoptimized Python implementation of our algorithm and only give a general idea of its scalability.

The first part of Table 6.2 shows that the synthesis time grows superlinearly (but not exponentially) in k, as expected since the DFAs $\mathcal{H}$ and $\mathcal{S}$ have $\Theta(k^4 \cdot 2^\ell)$ states. The instance used in our simulations above, $(k, n) = (7, 60)$, takes around a minute to solve, and we can solve up to 13×13 grids in $\sim$10 minutes. As with our CI algorithm, the execution time of the improviser itself is negligible, being less than a millisecond. The second part of the table fixes $k = 5$ and shows that the synthesis and improvisation times grow linearly with the length of the game n. Finally, the last part of the table shows some instances with very long games: even with $n = 900$, so that the improviser makes 450 moves, the improviser still runs in less than 15 milliseconds.

6.4 Summary and Future Work

In this chapter, we showed how to use algorithmic improvisation to synthesize randomized planners for robotic tasks requiring diverse or unpredictable behavior. We illustrated our techniques with examples of offline and on-the-fly planning for a surveillance drone which

Grid Size (k)	Trace Length (n)	DFA States	Synthesis Time (s)	Improvisation Time (ms)
5	60	20,002	9.4	0.86
7	60	76,834	44	0.96
9	60	209,954	120	0.83
11	60	468,514	300	0.98
13	60	913,954	630	0.88
5	80	20,002	12	1.1
5	100	20,002	15	1.4
5	120	20,002	17	1.7
5	140	20,002	21	2.1
5	160	20,002	24	2.3
5	300	20,002	45	4.5
5	600	20,002	88	9.2
5	900	20,002	140	14

Table 6.2: Performance of our DFA reactive improvisation scheme on various patrolling problems. The improvisation times were averaged over 10,000 improvisations.

must patrol an area in an unpredictable way while ensuring safety and other constraints. We showed how to encode various natural requirements for such a system as DFAs, allowing us to generate a randomized planner using our efficient improvisation schemes for such automata. Finally, we conducted experiments testing the resulting improvisers on simulated and actual drones. Our results demonstrate that our algorithms are practical, at least for relatively small problems, and can be made to work on real robots.

For future work, there are multiple interesting directions for both applied and theoretical research:

Other Tasks. In this chapter we focused on surveillance problems, but there are a number of other robotic tasks where randomness can be helpful. Examples include exploration and search and rescue tasks, especially for swarms of small robots with limited computation and communication ability [131]: here, randomness can be a partial substitute for coordination and planning[9].

Multi-Robot Planning. In swarm applications like those suggested above, planning would be done independently on each robot. However, for applications where we want to synthesize a *joint* planner for multiple robots, we quickly run into the state explosion problem: recall in Section 6.3 how the collision-avoidance DFA had a quadratic factor since it needed to track the positions of both drones. Naïvely constructing explicit

[9]Thanks to Marcell Vazquez-Chanlatte for suggesting this application.

automata for problems involving many robots would be totally impractical. One way to potentially circumvent this is to change the representation: recall that our improvisation scheme for Boolean formulas based on SAT solvers (Section 3.4.3) does not suffer from state explosion.

Infinite-Horizon Planning. Because control improvisation is defined over finite words, we can only plan for a finite horizon; this is in contrast to robotic planning by reactive synthesis, where one can use specifications like "visit room 4 infinitely often" in temporal logic [101]. While finite horizon planning is adequate for many applications, problems like patrolling are most naturally expressed over infinite words. We illustrated one way to get around this in Section 6.2, by requiring the route end up at the home base so we could immediately start another route. However, in the reactive setting this is only a heuristic: the adversary will now start from a different position, and the new RCI problem could be unrealizable. A more principled solution would require extending RCI to infinite words, which we mentioned as future work in Chapter 4.

Better Unpredictability. Finally, while the motivation for using control improvisation for robotic surveillance was unpredictability, the type of unpredictability guaranteed by the definition of CI is not necessarily what is desired. Specifically, the randomness constraint in CI forces the distribution of traces to be close to uniform and therefore unpredictable; however, particular *statistics* of the traces could be highly predictable. For example, suppose we are generating length-100 binary strings, and $\rho = 2^{-50}$. Then every individual string has extremely low probability and cannot be predicted, but it is entirely legal for an improviser to output only strings where the first half of the bits are all zero! Returning to the reactive drone example in Section 6.3, the drone's route might be close to uniform, but the *order* in which the 4 locations are visited is not guaranteed to be. To address this issue, we need extensions of CI and RCI with more sophisticated randomness constraints, as suggested in Chapters 3 and 4.

Chapter 7

Human Modeling

In this chapter, we explore the use of algorithmic improvisation to synthesize *models of human behavior*. We have already seen how algorithmic improvisation can be useful in creating models: in Chapter 5, we proposed language-based improvisation as a way to write models of useful tests, and in Chapter 8 below we will see that such models are very helpful in practice. When building *human* models, however, there is an additional complication, namely that we generally do not have formal specifications of human behavior, only *examples*. To solve this problem we introduce a variant of control improvisation which *learns* a distribution from training data. We demonstrate our approach in a case study synthesizing a randomized lighting controller that behaves in a human-like way while respecting constraints on power consumption [2].

7.1 Introduction

Accurate human models are extremely useful, both for mimicking human behavior and for designing and analyzing systems that interact with humans. Applications of the first type include conversation systems (e.g. chatbots) which should engage in human-like dialog, or a home security system which should turn lights in a house on and off in a human-like way to obscure the fact that the owner is on vacation [2]. The second category includes applications like environment modeling for autonomous cars, where realistic models of human drivers, bicyclists, and pedestrians are needed to do meaningful testing and training in simulation.

For both types of applications, algorithmic improvisation provides a natural way to construct models: we need *randomized* models since human behavior is inherently stochastic, but we also need constraints, either to enforce desired properties on our synthesized system or to generate human-like behaviors of a particular type. In the lighting control application, we can for example use soft constraints to ensure that the power consumption of the house is usually below a given limit. As another example, Ge and Murray [69] have used the techniques we propose in this chapter to synthesize a human-like lane-change controller, using constraints to require that lane changes occur at a desired rate.

However, the versions of algorithmic improvisation we explored in earlier chapters have a serious drawback when used for human modeling: they assume that the desired requirements on the synthesized model are explicitly given as hard, soft, and randomness constraints (as well as an explicit generative process for language-based improvisation in Chapter 5). This is not a reasonable assumption for human modeling, where we do not have a formal definition of "human-like" behavior. Instead, we should take the typical approach in modeling: *learn* the model from data.

Therefore, we propose a heuristic improvisation procedure for a certain class of algorithmic improvisation problems useful for human modeling, and in particular for the randomized lighting control problem described above. Our procedure first learns a probabilistic model from training data, which captures the distribution of the data without necessarily respecting the desired soft constraints. We then iteratively *calibrate* the model to increase the probability that the soft constraints are satisfied, using probabilistic model checking [12] to determine when the model is adequate and we can terminate. We propose two types of calibration heuristics suited to soft constraints which impose upper bounds on *costs*, e.g. accumulated power consumptions, of the system.

We apply this procedure in a case study on the randomized lighting control problem, learning a model from power consumption data from a real house. Using this model, we construct improvisers subject to soft constraints limiting hourly power consumption. Our experiments demonstrate that our improvisers generate synthetic lighting behaviors that are qualitatively and quantitatively similar to human behaviors in our training set, while also respecting our desired soft constraints. These results show that algorithmic improvisation provides a useful framework for building constrained models of human behavior.

We begin in Section 7.2 with background on the types of systems and probabilistic models we consider, leading up to a definition of the modeling problem and a partial formulation in terms of the multi-constraint control improvisation problem from Chapter 3. Next, Section 7.3 describes our procedure for learning an improviser from data, including our model calibration heuristics. Section 7.4 then presents our experiments from the lighting control case study. Finally, we conclude in Section 7.5 with a summary and directions for future work. Throughout the chapter, we focus on the general application of human modeling and its connection to algorithmic improvisation; for a more detailed discussion of applications to Internet-of-Things systems and related work in appliance use modeling, see Akkaya et al. [2] and Akkaya [1].

7.2 Background and Problem Definition

7.2.1 Background

Discrete-Event Systems with Hidden States

We will focus on learning models of systems (possibly involving humans) whose behavior can be described by a sequence of *events*, where an event is a pair (τ, v) consisting of a *timestamp*

from a totally-ordered set T and a *value* from a finite set V. A sequence of events ordered by their timestamps is called a *signal* [106].

We assume that the system's dynamics can be represented by a transition system with a finite number of states, with events corresponding to transitions between states. However, the underlying transition system is not known *a priori*, and we can only observe some function of the state, called the *observation*. The observation function can be time-dependent and probabilistic, so that many different observations are possible in a single state. We assume that the possible observations are finite, forming an alphabet Σ, and that observations are made at discrete time steps. For continuous systems like the lighting control example we will study below, we can satisfy these assumptions by appropriate quantization. Then, a sequence of observations corresponding to a behavior of the system forms a word in Σ^*, which we call a *trace*.

For example, consider a system consisting of the lights in three rooms of a house. We can model this using a finite transition system whose hidden state represents which room's lights are currently turned on (so that there are $2^3 = 8$ states). For each room, we have "ON" and "OFF" events for turning its lights on and off. The observation function models what we can actually measure about the system, namely the power consumptions for each of the three rooms. An example trace, along with the corresponding events, is shown in Figure 7.1. Initially, we start in the state where all lights are off. At 19:50, the kitchen light is turned on, causing the corresponding event to be emitted and moving the system to a new state representing that the kitchen light is the only one on. Note that while we remain in this state for some time (until the next event), the observation is not constant: the power consumption of the kitchen fluctuates, which we can model using a probabilistic observation function. Note also that given only the trace of observations, we do not know what the underlying sequence of events and states was: these have to be *inferred* from a set of traces.

Explicit-Duration Hidden Markov Models

The simplest type of probabilistic model suited to learning the dynamics of the type of system described above is the *Hidden Markov Model (HMM)* [140]. An HMM has a hidden state which evolves according to Markovian dynamics (so that the states and transitions are just a Markov chain), and a state-dependent probabilistic observation function. Although HMMs are widely used, their Markovian dynamics implies that the amount of time spent in a given state before moving to a different one follows a geometric distribution, which is not a realistic assumption in many applications. In such cases, the quality of learned models can be significantly improved by using a *semi-Markov model* allowing transition probabilities to depend on the time elapsed in the current state.

Here, we will use the *Explicit-Duration Hidden Markov Model (EDHMM)* [140, 193, 32], which extends the HMM by adding the *duration* of each state as a hidden variable. The EDHMM, evolved for T time steps, is shown as a graphical model in Figure 7.2. As in an HMM, we have a sequence of states $x_1, \ldots, x_T$ from the state space $\mathcal{X}$, with corresponding

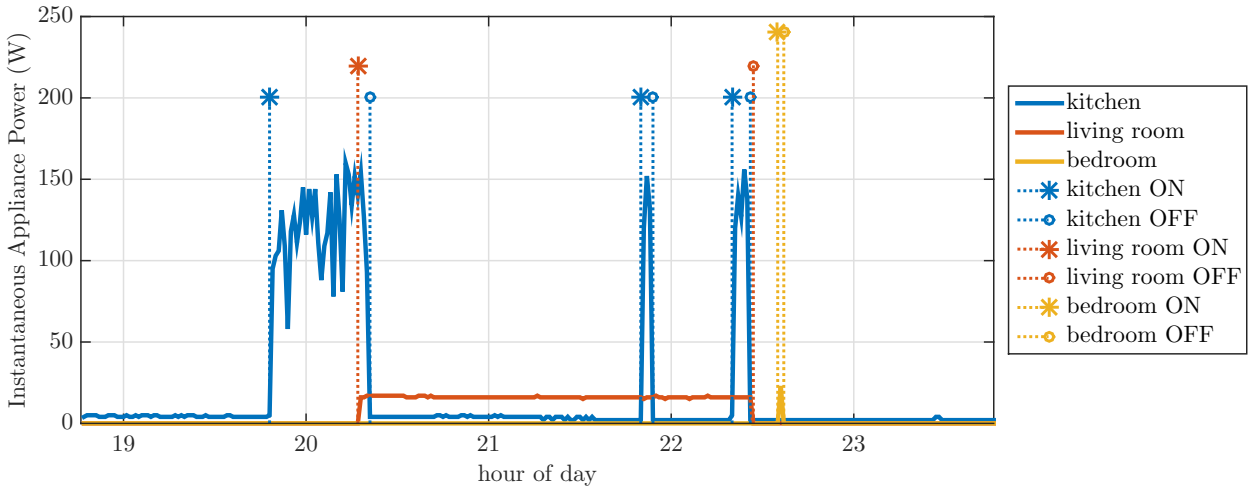


Figure 7.1: Example power consumption trace, with the underlying hidden events.

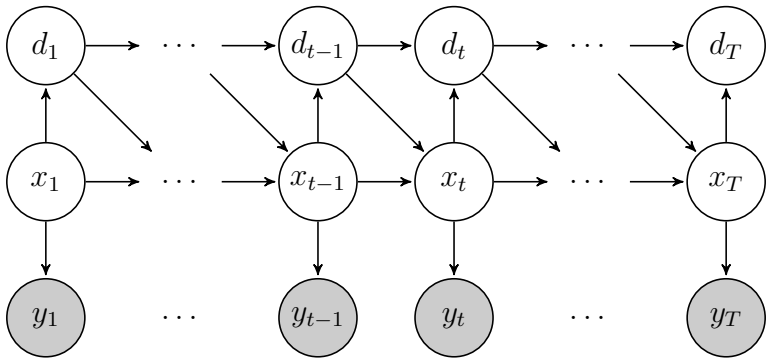


Figure 7.2: Graphical model representation of an EDHMM, consisting of the states x_i, observations y_i, and durations d_i.

observations $y_1, \ldots, y_T$ from the alphabet Σ (recall that we assume $\mathcal{X}$ and Σ are both finite). We also have a sequence of durations $d_1, \ldots, d_T$, which represent the time remaining in the current underlying state as an integer from 1 to some upper bound D. Informally, the dynamics are as follows: when $d_{t-1} = 1$, then it is time for a transition, so we sample a new state according to the HMM dynamics (some distribution $p(x_t|x_{t-1})$) and a duration for it (according to some distribution $p(d_t|x_t)$); otherwise, time continues to elapse, so we put $x_t = x_{t-1}$ and $d_t = d_{t-1} - 1$ (see Section 7.3.3 and Akkaya et al. [2] for details).

Note that an EDHMM is simply an HMM on a larger state space, namely pairs of states and durations. Thus, its underlying dynamics (on the enlarged state space) are described by a Markov chain, which will be useful later. We will actually use a slight extension of the EDHMM where we allow a limited form of time-dependence, namely having different transition and duration distributions for each of the 24 hours of the day. This can again be

reduced to a Markov chain by expanding the state space with a variable representing the current time step (we will describe this encoding in detail in Section 7.3.3).

Finally, to learn an EDHMM from data, we use the typical Bayesian inference approach of finding the model parameters which maximize the likelihood of the given set of traces. As for HMMs, this can be done using a variant of the Expectation-Maximization (EM) algorithm (see Yu [193]). For our extended EDHMM with hourly dynamics, we use the same algorithm, simply inferring the parameters for each hour independently from the corresponding parts of the traces.

Probabilistic Model Checking

As mentioned above, our improvisation procedure will iteratively adjust our learned probabilistic model until the soft constraints are satisfied. To determine whether our current model satisfies these constraints, we use *probabilistic model checking*, which provides algorithms for checking whether a variety of different probabilistic models satisfy a formal specification expressed in a probabilistic temporal logic. The particular logic we use here is *Probabilistic Computation Tree Logic (PCTL)*. We will give only an informal overview of the logic; see Baier and Katoen [12] for details.

Formulas ϕ of PCTL are defined by the following grammar:

$$\phi := \text{True} \mid \omega \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid P_{\bowtie p}[\psi] \qquad (\textit{state formulas})$$
$$\psi := X\phi \mid \phi_1 U^{\leq k} \phi_2 \mid \phi_1 U \phi_2 \qquad (\textit{path formulas})$$

where ω is an *atomic proposition*, i.e. a predicate over states, $\bowtie$ is one of the operators $\{\leq, <, \geq, >\}$, $p \in [0, 1]$, and $k \in \mathbb{N}$. State formulas ϕ are interpreted at states of the probabilistic model: an atomic proposition ω is true if and only if it holds in the current state, the negation and conjunction Boolean operators have their obvious meanings, and $P_{\bowtie p}[\psi]$ is true if and only if the probability q that the path formula ψ holds starting from the current state satisfies $q \bowtie p$. Path formulas are interpreted over runs of the probabilistic model and are built up using the *temporal operators*:

Next. $X\phi$ is true if and only if ϕ is true in the next state of the run.

Bounded Until. $\phi_1 U^{\leq k} \phi_2$ is true if and only if ϕ_2 is true in one of the next k steps of the run, and ϕ_1 is true at every step until then.

Unbounded Until. $\phi_1 U \phi_2$ is true if and only if ϕ_2 is true at some future step, and ϕ_1 is true at every step until then.

For convenience we can define other Boolean operators using $\neg$ and $\wedge$ in the usual way, as well as two additional temporal operators:

Finally. $F\phi := \text{True}\, U\phi$ is true if and only if ϕ eventually becomes true.

Globally. $G\phi := \neg F \neg \phi$ is true if and only if ϕ is always true.

For example, if *bad* is a predicate designating a set of unsafe states for the system, then the formula $P_{\leq 0.1}[F\, bad]$ holds on the model if and only the probability of ever reaching an unsafe state is at most 0.1.

Given a Markov chain and a PCTL formula, a probabilistic model checker such as PRISM [103] can determine whether the model satisfies the formula. Moreover, PRISM can also compute the probability with which a path formula holds in a model (rather than simply deciding whether the probability exceeds a given threshold). Since as we mentioned above an EDHMM can be reduced to a Markov chain, we can in particular use PRISM to check whether an EDHMM satisfies a PCTL formula.

7.2.2 Problem Definition

We begin with a set of traces of a discrete-event system whose set of events is known, but whose dynamics are not. Our goal is to learn an EDHMM randomly generating new traces with similar characteristics to the training data, subject to two types of constraints:

- *Hard constraints* forbidding transitions between states that never occur in the input traces. More precisely, if no part of the training data can be explained as a state transition t, then we want to assume that t is impossible and not generate any trace that is only possible using it. Recalling that our approach will be to first learn an EDHMM M_0 from the training data ignoring the constraints, and then iteratively construct a sequence of modified EDHMMs M_i, the hard constraint simply ensures that the adjusted models M_i do not introduce any new behaviors not possible in M_0.
- *Soft constraints* that need only be satisfied with some given probabilities. In particular, we focus on soft constraints upper bounding nonnegative *costs* which can be computed from the observations, either at a particular time or accumulated over a time period. For example, we could require that 90% of the time, the total power consumed over the day does not exceed some bound.

We can partially formulate this problem in the framework of multi-constraint control improvisation introduced in Chapter 3. To do this, we need to define the alphabet Σ, hard specification $\mathcal{H}$, soft specifications $\mathcal{S}_i$ and corresponding error probabilities ϵ_i, and the probability bounds λ and ρ:

Σ: Since we want to generate traces, which are sequences of observations, we let Σ be the set of all possible observations (i.e. those occurring anywhere in the input traces, after quantization).

$\mathcal{H}$: To ensure the hard constraint above, we want the set of improvisations $L(\mathcal{H})$ to consist of all traces that are assigned nonzero probability by the initial EDHMM M_0. It is straightforward to build an NFA $\mathcal{H}$ accepting exactly these traces (although we do not actually need to do this for the heuristic improvisation procedure we use below).

$\mathcal{S}_i, \epsilon_i$: We let $\mathcal{S}_i$ be a PCTL formula encoding the i-th soft constraint. In our lighting example, $\mathcal{S}_i$ could encode that the total power consumption within hour $i \in \{1, \ldots, 24\}$ of the day never exceeds a given bound. Correspondingly, ϵ_i is the greatest probability we are willing to tolerate of the improviser generating a trace violating the bound.

λ, ρ: We put $\lambda = 0$ since it is not necessary to generate every possible trace (indeed, the hard constraint likely forbids some traces). Picking a small value for $\rho > 0$, we can ensure that at least $1/\rho$ different traces can be generated.

The major gap in this formulation is that it does not constrain the improviser's behavior to be "human-like" (more than superficially through the hard constraint). One could imagine using additional soft constraints to try to do this; however, it is not clear that any small set of soft constraints would be sufficient for this purpose. Furthermore, there is no single concept of "human-like", since any two people will likely behave differently: we want our controller to mimic the behaviors seen in our particular training set. This issue leads us to propose a specialized improvisation algorithm which learns a model directly from the training set.

7.3 An Iterative Improvisation Procedure

7.3.1 Overview

As described above, our desired improviser must satisfy two potentially-conflicting goals: have a similar behavior distribution to the training set, and satisfy hard and soft constraints. Our approach deals with these concerns separately, using Bayesian inference to learn an initial model of the training data, then iteratively adjusting the model until the constraints are satisfied. This procedure, outlined in Figure 7.3, has three main steps, which we will detail in subsequent sections:

(1) *Data-Driven Modeling:* From the given traces, we learn an EDHMM which represents the (possibly time-dependent) dynamics of the underlying system. This yields a *candidate improviser*, which satisfies our hard constraints by definition.

(2) *Probabilistic Model Checking:* Next, we check whether the candidate improviser satisfies the soft constraints, expressed as PCTL formulas, using a probabilistic model checker. If so, we have found a valid improviser and terminate. Note that in Figure 7.3 we allow the PCTL properties to depend on the training data: as we will see below, it can be useful to set parameters of the soft constraints based on the training data.

(3) *Model Calibration:* Otherwise, if the soft constraints are violated, we modify the parameters of the candidate improviser to increase the probability that it satisfies the soft constraints. How to do this depends on the form of the soft constraints: below, we will give two heuristics suitable for constraints bounding costs, as in our lighting control example. These heuristics do not introduce any new behaviors into the model,

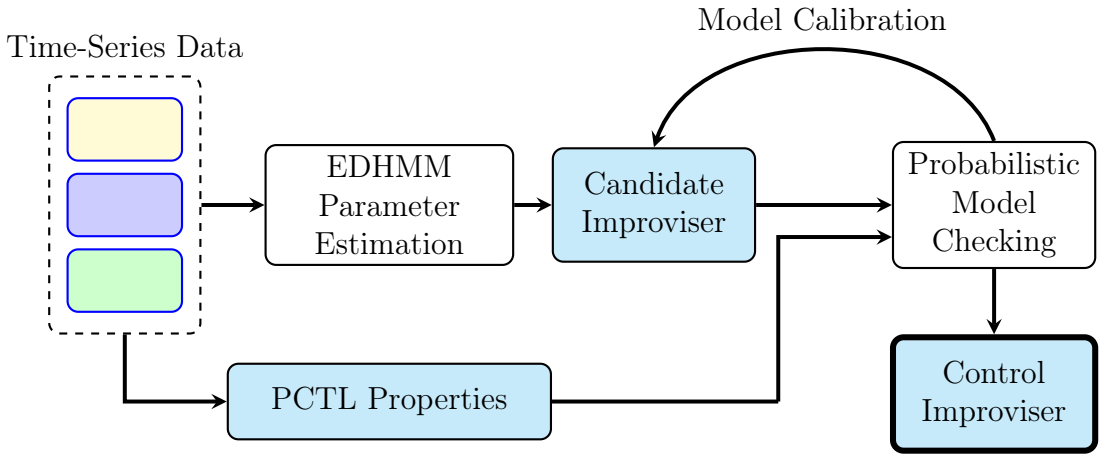


Figure 7.3: Iterative algorithm for learning an improviser from data.

thereby preserving the satisfaction of the hard constraints. We then return to step (2) with the new candidate improviser.

As this procedure is heuristic, it is not guaranteed to produce an improviser for the CI problem we defined earlier. If the procedure terminates, the hard and soft constraints will be satisfied by construction; however, the randomness constraint may not be satisfied (indeed, our procedure makes no attempt to enforce it). For example, even if the initial EDHMM has a significant amount of randomness in its behavior, over many rounds the model calibration heuristics could restrict the system so much that it ultimately becomes deterministic. Short of this, as long as the EDHMM is ergodic (after conversion to a Markov chain), the probability of generating any particular trace goes to zero as its length goes to infinity. We can efficiently detect when the EDHMM is not ergodic using standard graph algorithms, and otherwise satisfy the randomness requirement by generating sufficiently long strings. However, this is unlikely to be necessary in practice for reasonable soft constraints, and in fact was not used in our experiments below.

More interestingly, our procedure can also fail by never terminating. In general, this is unavoidable, since if the soft constraints are themselves inconsistent then no modification of the original EDHMM can satisfy them. When it *is* possible to satisfy the soft constraints with some model, termination depends on the adequacy of the model calibration heuristics to converge to such a model. As we will show below, our heuristics for soft constraints upper bounding costs have this property: they decrease the expected cost of the system, so that after sufficiently-many iterations the model will satisfy the soft constraints.

As a final note, our technique can be extended to enforce other types of hard constraints. For example, we can easily disallow undesired transitions between hidden states by setting their probabilities to zero in the initial candidate improviser (normalizing the other transition probabilities appropriately). Such constraints can be useful, for example, when controlling

an IoT system with unreliable components or networks: if a component becomes unusable, we can disable all transitions to states in which the component is active.

Now we describe each of the steps of our procedure in more detail.

7.3.2 Data-Driven Modeling

Our procedure begins by learning a probabilistic model from the training set. Note that the overall procedure is not specific to any particular kind of model, and could be used with any model for which there are practical parameter estimation and probabilistic model checking algorithms. As explained in Section 7.2.1, here we use an extended EDHMM with different dynamics for each hour of the day $h \in \{1, \ldots, 24\}$. Its parameters consist of matrices $\{A_h\}$ and $\{C_h\}$ representing state transition and duration probabilities, a matrix B representing state observation probabilities, and a prior π on the state space. Here $\{C_h\}$ and B encode the duration and observation distributions as categorical distributions, although it is also possible to use parametric distributions.

We estimate these parameters from the training set using an EM algorithm, as described in Section 7.2.1. Note that the EM algorithm is an iterative method, whose convergence to a good set of parameters depends in practice on having a large enough training set. This can be problematic in our setting since we learn the dynamics for each hour of the day independently, and even in a large training set there may be few events during certain hours. Naïvely, this would prevent us from estimating the state transition and duration probabilities during such hours. However, there are many application-specific heuristics for dealing with insufficient training data, outlined in Rabiner [140]. The particular heuristic used in our experiments will be described in Section 7.4.1.

7.3.3 Probabilistic Model Checking

Next, we use probabilistic model checking to test whether the soft constraints are satisfied. In particular we use an algorithm for PCTL model checking of Markov chains, as implemented in PRISM [103]. This requires us to encode our soft constraints as PCTL formulas, and our EDHMM as a Markov chain.

Encoding Soft Constraints in PCTL

As explained above, we consider soft constraints which put upper bounds on the cost observed at a particular time or accumulated over a time period. We illustrate how to encode upper bounds on the hourly cost — other time periods are handled analogously.

Recall that our traces have the form $(\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_T)$ where each $\mathbf{y}_i$ is an observation, itself a tuple $(y_{i,1}, \ldots, y_{i,K})$ of nonnegative costs (which, like durations, we assume to be integers after quantization). For example, $\mathbf{y}_3$ could represent the power consumptions of all rooms in a house at time step 3, with $\mathbf{y}_{3,2}$ being that of the kitchen. Let $Y_i = \sum_{k=1}^{K} y_{i,k}$ be the total cost at time step i. If there are N_s time steps in an hour, at time step t the current hour of

the day is $h(t) = [(\lceil t/N_s \rceil - 1) \bmod 24] + 1$, and it began at time $N_s(\lceil t/N_s \rceil - 1) + 1$ (recalling that we number time steps and hours starting from 1). Then the total cost accumulated in the current hour is

$$\Delta = \sum_{i=N_s(\lceil t/N_s \rceil - 1)+1}^{t} Y_i.$$

As mentioned above, when encoding the EDHMM as a Markov chain, we need to add the time step t to the state in order to model the different dynamics for each hour of the day. Having done this, $h(t)$ is a function defined over the state of the chain, and so we can use atomic predicates testing it in PCTL (since PCTL atomic predicates are defined over individual states of the system). However, since Δ accumulates over multiple time steps, we also need to maintain Δ as part of the state. Below, we will show how to do this by adding a simple monitor to the encoding of the EDHMM as a Markov chain. So we can assume we have atomic predicates testing the values of $h(t)$ and Δ in the current state.

Now we can formalize our soft constraints. To require that the accumulated cost during hour h is at most Δ^h_{max}, we use the PCTL formula

$$\mathcal{S}_h = P_{\geq 1-\epsilon_h} \left(G \left[(h(t) = h) \rightarrow (\Delta \leq \Delta^h_{max}) \right] \right).$$

As we might expect, $\mathcal{S}_h$ simply asserts that with probability at least $1-\epsilon_h$, at every time step during hour h the accumulated hourly cost Δ is at most Δ^h_{max}. If we do not have a particular error probability ϵ_h in mind, as mentioned above we can ask the probabilistic model checker to *compute* the probability that the inner path formula of $\mathcal{S}_h$ holds.

Encoding the EDHMM as a Markov Chain

As outlined in Section 7.2.1, we can view an EDHMM as a Markov chain on an expanded state space with a new variable $d \in \{1, \ldots, D\}$ which represents the remaining duration of the current hidden state $x \in \mathcal{X}$. Following the semantics of the EDHMM, when $d > 1$ we stay in x for another time step, only decrementing d; when $d = 1$, we instead transition to a new state, picking a new duration d from the corresponding duration distribution.

Since we use an extended EDHMM with time-dependent dynamics, we need to further expand the state space to keep track of the current time. We introduce a state variable $t \in \{0, \ldots, T\}$ representing the current time step, with $t = 0$ being a special initialization step indicating that we need to sample the initial state $x \in \mathcal{X}$ from its prior π. For simplicity we write t as representing absolute time, being incremented each time step. However, it is not in fact necessary to have the domain of t grow unboundedly with the length of the run T: in our example with different dynamics in every hour of the day, we need only track the time modulo a day, i.e. $24N_s$ time steps.

Finally, to detect when the soft constraints are violated, we need to monitor the accumulated hourly cost Δ defined above. We add a state variable $\Delta \in \{0, \ldots, \Delta_{max} + 1\}$, where Δ_{max} is the largest of the hourly upper bounds Δ^h_{max} imposed by the soft constraints. This range of values is obviously sufficient to detect when the accumulated cost exceeds its bound

for any particular hour. The dynamics of Δ are straightforward: at each time step we add the observed cost to it, except when starting a new hour, in which case we first reset it to zero.

Putting this all together, we obtain a Markov chain whose states are tuples (x, d, t, Δ) with the variables described above. We fix the initial state to any state with $t = 0$, since we will use time step 0 to initialize the other state variables. Given the current state, the next state (x', d', t', Δ') and observation $\mathbf{y}'$ are determined as follows:

EDHMM:

$$\begin{array}{rcll} (t=0) & \rightarrow & x' \sim \pi_x \wedge & \text{(initialization)} \\ & & t' = t+1 \wedge & \\ & & d' \sim C_{h(t')}(x') \wedge & \\ & & \mathbf{y}' \sim B(x') & \\ (t>0) \wedge (d>1) & \rightarrow & x' = x \wedge & \text{(countdown)} \\ & & t' = t+1 \wedge & \\ & & d' = d-1 & \\ & & \mathbf{y}' \sim B(x') & \\ (t>0) \wedge (d=1) & \rightarrow & x' \sim A_{h(t)}(x) \wedge & \text{(state transition)} \\ & & t' = t+1 \wedge & \\ & & d' \sim C_{h(t')}(x') & \\ & & \mathbf{y}' \sim B(x') & \end{array}$$

Cost Monitor:

$$\begin{array}{rcll} (t=0) & \rightarrow & \Delta' = \sum_{i=1}^{K} \mathbf{y}'_i & \text{(initialization)} \\ (t>0) \wedge (h(t') = h(t)) & \rightarrow & \Delta' = \Delta + \sum_{i=1}^{K} \mathbf{y}'_i & \text{(accumulation)} \\ (t>0) \wedge (h(t') \neq h(t)) & \rightarrow & \Delta' = \sum_{i=1}^{K} \mathbf{y}'_i & \text{(start of new hour)} \end{array}$$

recalling that $h(t) = [(\lceil t/N_s \rceil - 1) \bmod 24] + 1$.

7.3.4 Model Calibration

When model checking determines that our candidate improviser does not satisfy a soft constraint, we *calibrate* the model to increase the probability with which the constraint is satisfied. The calibration procedure should make a small change to the model parameters, preserving as much as possible the original distribution of behaviors which was learned from the training data. In particular, the procedure should not *introduce* any new behaviors, as required by our hard constraints.

We present two types of calibration heuristics for soft constraints of the form described above, namely upper bounds on instantaneous or accumulated costs. These heuristics, *duration* and *transition calibration*, both seek to reduce the cost of one or more behaviors of the improviser.

Duration Calibration

One basic way to decrease costs is to alter the duration distribution of a state with high expected cost so that less time is spent in the state. Recalling that we assumed when learning the EDHMM that state durations can be at most some finite duration D, one way to decrease the expected duration is simply to truncate the distribution at some threshold below D. Of course, reducing the time spent in one or several states does not necessarily decrease the overall cost: transitioning quickly out of one state may cause us to spend more time in an expensive state. However, as we will see below, it can be effective to eliminate outliers (unusually long durations) in the duration distributions of the states with the highest expected costs.

Duration calibration has the advantage of being a relatively minor modification, as it leaves the state transition probabilities of the model unchanged. On the other hand, it cannot reduce the duration of a state below 1 time step. So although it may be able to eliminate some high-cost behaviors from the model, it is not guaranteed to eventually yield an improviser satisfying the soft constraints.

Transition Calibration

A different approach is to modify the state transition probabilities, making the model less likely to enter states with high expected costs. Specifically, for any transition $x \to y$, we can limit the probability of the transition during hour h to be at most some bound $p_h^{x \to y}$. The removed probability mass must be shifted somewhere else, and a simple approach is to add it to the transition $x \to x_{min}$, where x_{min} is the state which has the lowest expected cost (in our lighting example, the state where all lights are off). Given the original transition probability vector $A_h(x)$ of the EDHMM, we replace it with a new vector $\tilde{A}_h(x)$ defined by

$$\tilde{A}_h(x)(j) = \begin{cases} \min(p_h^{x \to y}, A_h(x)(y)) & \text{if } j = y \\ A_h(x)(x_{min}) + \max(A_h(x)(y) - p_h^{x \to y}, 0) & \text{if } j = x_{min} \\ A_h(x)(j) & \text{otherwise.} \end{cases}$$

Note that the second case ensures that the transition probabilities from the state x are properly normalized.

Provided that the bounds $p_h^{x \to y}$ are chosen so that the probability of entering some state other than x_{min} decreases by at least some fixed amount, this heuristic will decrease the expected cost of a run of the improviser. In the limit, applying the heuristic iteratively for every choice of state y other than x_{min} and for every hour $h \in \{1, \ldots, 24\}$, we will eventually obtain an improviser which remains in the state x_{min} for all time (assuming it starts there). Thus for any soft constraints which are true for behaviors that only stay at x_{min}, our procedure will eventually terminate and yield a valid improviser. Of course, this over-simplified improviser is unlikely to model the original data well, but it is only attained as the limit of the heuristic: in practice, judicious choices of the transitions to modify and

the limits $p_h^{x \to y}$ can improve the probability of satisfying the soft constraints significantly in a few iterations without drastically changing the model.

7.4 Experiments

To demonstrate our approach, we performed a case study on the randomized lighting control problem described informally above. We trained our model on real power consumption data from a house, applying our algorithm to obtain an improviser which generates human-like lighting behaviors while respecting constraints on hourly power consumption. Below, we describe the setup and results of our experiments.

7.4.1 Experimental Setup

Our lighting control scenario is based on data from the UK Domestic Appliance-Level Electricity (UK-DALE) dataset [96], which contains power consumption time series for residential appliances from 5 homes over a period of 3 years. We consider the three most-used lighting appliances from different rooms of the house, namely, the main kitchen light, a dimmable living room light, and the bedroom light respectively. Accordingly, our observations $\mathbf{y}_i$ are triples of instantaneous power readings from these three appliances, measured in Watts and sampled every minute (so that there are $N_s = 60$ time steps per hour). We now describe the specific choices we made when implementing the procedure of Section 7.3.

Data-Driven Modeling

To define the EDHMM, we assume that for each appliance there are hidden events corresponding to the appliance being turned on and off. The hidden state space $\mathcal{X}$ therefore has $2^3 = 8$ states, one for each combination of active appliances. Based on inspecting the dataset, we chose the maximum state duration D to be 720 time steps, i.e. 12 hours (sufficient to allow long periods when all appliances are off). The power consumption traces in the dataset are already quantized, so that the observations come from the alphabet $\Sigma = \Sigma_1 \times \Sigma_2 \times \Sigma_3$ where $\Sigma_1 = \{0, 1, \ldots, 350\}$, $\Sigma_2 = \{0, 1, \ldots, 20\}$, and $\Sigma_3 = \{0, 1, \ldots, 30\}$ (the bounds for each appliance again being obtained by inspecting the dataset). For training, we selected a 100-day subset of the dataset for one residence. The complete set of dataset and model parameters used in our experiments is shown in Table 7.1. Several representative traces for a single appliance are shown at the top of Figure 7.9 in the next section.

We trained the EDHMM as described in Section 7.3.2, with one minor change: since the power consumptions of each appliance are independent, we factored the probability matrix for observations into a product of matrices for each appliance. As we mentioned in Section 7.3.2, due to some events never occurring during early morning hours of low activity (typically hours $h \in \{1, \ldots, 5\}$), the EM algorithm did not learn the corresponding state transition probabilities: that is, the learned EDHMM had some states where all outgoing

Parameter	Value
Data Source	UK-DALE Dataset [96]
House ID	house_1
Appliance IDs	kitchen_lights livingroom_s_lamp bedroom_ds_lamp
Training Duration Training Start Date Sampling Period Training Sequence Length (T)	100 days 30 Jul 2013 19:07:56 GMT 60 s 144000
Maximum State Duration (D) Appliance 1 Costs (Σ_1) Appliance 2 Costs (Σ_2) Appliance 3 Costs (Σ_3)	720 $\{0, 1, \ldots, 350\}$ $\{0, 1, \ldots, 20\}$ $\{0, 1, \ldots, 30\}$
Hidden States ($\mathcal{X}$)	OFF: All appliances off K/L/B: Kitchen / living room / bedroom on KL: Kitchen and living room on KB: Kitchen and bedroom on LB: Living room and bedroom on KLB: All appliances on

Table 7.1: Parameters of the training dataset and EDHMM used in our experiments.

transitions had probability zero. To fix this, we used a completion strategy which specified all such states must transition to the OFF state (see Table 7.1) with probability 1. This yielded a fully-specified EDHMM.

Probabilistic Model Checking

We imposed soft constraints upper bounding the total power consumed during each hour of the day. Figure 7.4 depicts the hourly consumptions of each appliance, as well as the aggregated consumption, averaged across each day in the training data. We set our upper bounds Δ^h_{max} to be one standard deviation above these average consumptions (the upper edge of the shaded region in the aggregate graph in the Figure). Out of all samples in the training data, 89.2% lay within these bounds.

To compute the probability that the candidate improviser satisfied these constraints, we used the PRISM model checker [103], implementing the Markov chain encoding of the EDHMM we described in Section 7.3.3 in the PRISM modeling language. The soft constraints were put directly into PRISM using the PCTL formulation explained in that section.

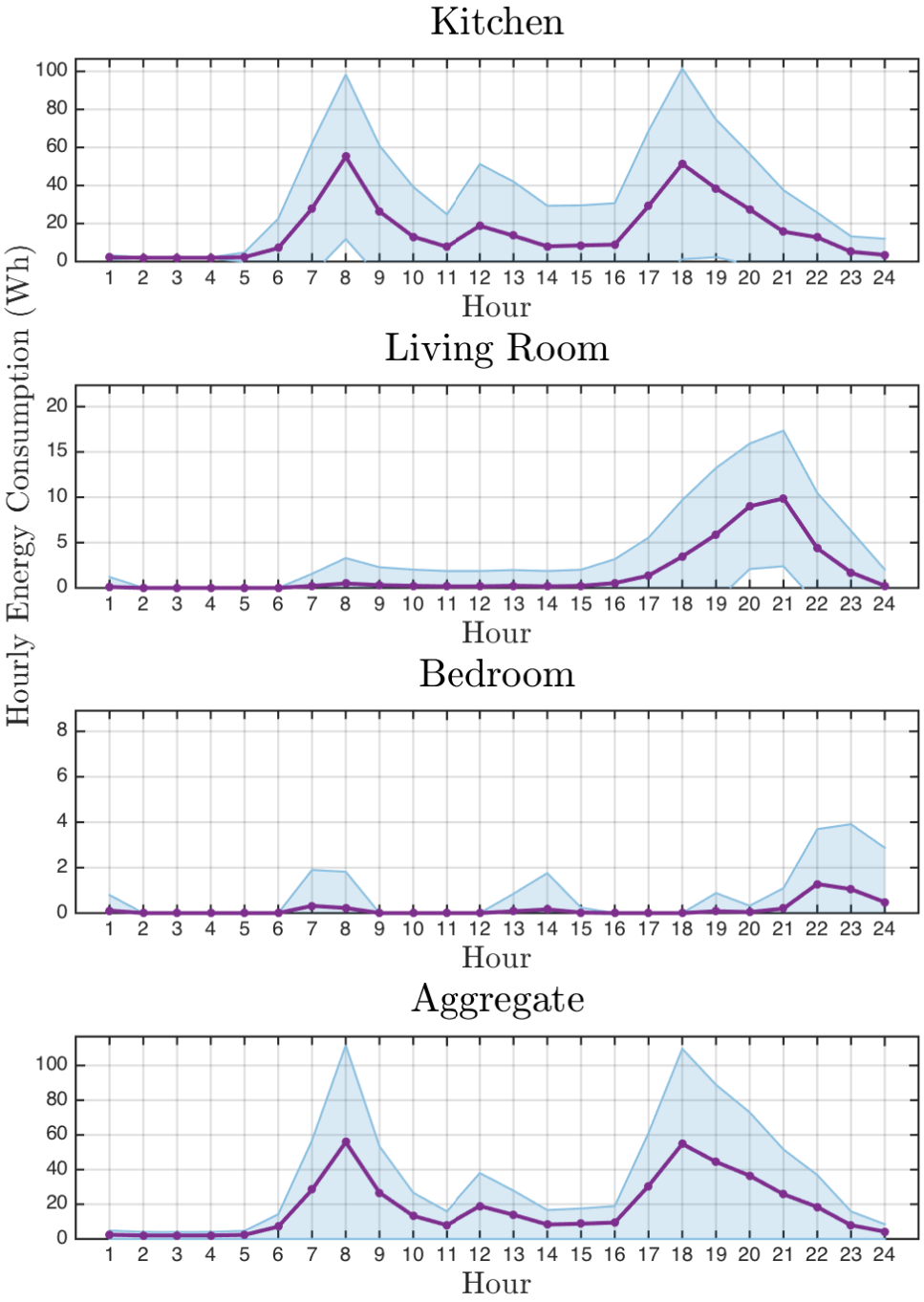


Figure 7.4: Average hourly usage patterns of the appliances in our training dataset. The solid curves represent average consumption, and the shaded areas show one standard deviation around the mean.

Model Calibration

To evaluate the effect of model calibration, we tested three types of improvisers:

Uncalibrated Improviser. As a baseline, we created an "improviser" using the initial EDHMM with no calibration. We would expect this model to be closest in behavior to the training data, although it is not a true improviser since it may not satisfy our soft constraints.

Duration-Calibrated Improviser. This improviser uses the duration calibration heuristic described in Section 7.3.4. From the aggregate power profile shown in Figure 7.4, we identified peak power consumption as occurring during hours 7, 8, 9, 17, 18, 19, 20, and 21. For these hours, all state duration distributions (except for the OFF state) were clamped at 60 minutes, setting the probabilities of longer durations to zero and re-normalizing. The effect of calibration is illustrated in Figure 7.5, which shows the duration distributions for two states before and after calibration.

Transition-Calibrated Improviser. This improviser extends the previous one by also applying the transition calibration heuristic from Section 7.3.4. Transition probabilities were calibrated for the peak hours used for the previous calibration, as well as hours 4 and 5, during which very few events were recorded in the training data. As shown in Figure 7.4, the kitchen and living room appliances consume significantly more power than the bedroom appliance. Therefore, we applied calibration to all transitions leading to states K, L, KL, and KLB.

Figure 7.6 shows the transition probability matrices for several hours of the day before and after calibration. Each circle indicates a nonzero transition probability from state x_t to x_{t+1}, with its area being proportional to the probability. The blue circles show the original learned probabilities, and the green circles show the probabilities decreased by calibration. For clarity, we do not show the corresponding increases in the probabilities of transitioning to the OFF state.

7.4.2 Experimental Results

Now we evaluate the performance of our synthesized improvisers, both with respect to their satisfaction of the soft constraints and their similarity to the original training data. To start, Figure 7.7 shows the probabilities of each improviser satisfying the hourly soft constraints, as computed by PRISM. For comparison, the Figure also shows the empirical probabilities with which the training data satisfies the soft constraints. We can see that the uncalibrated improviser (blue) behaves similarly to the training data (purple) for much of the time, but unsurprisingly has poor performance with respect to the soft constraints, with relatively low satisfaction probabilities in hours 4–7. Applying duration calibration (red) generally improves the model, increasing the satisfaction probabilities for all hours except 9, 21, and

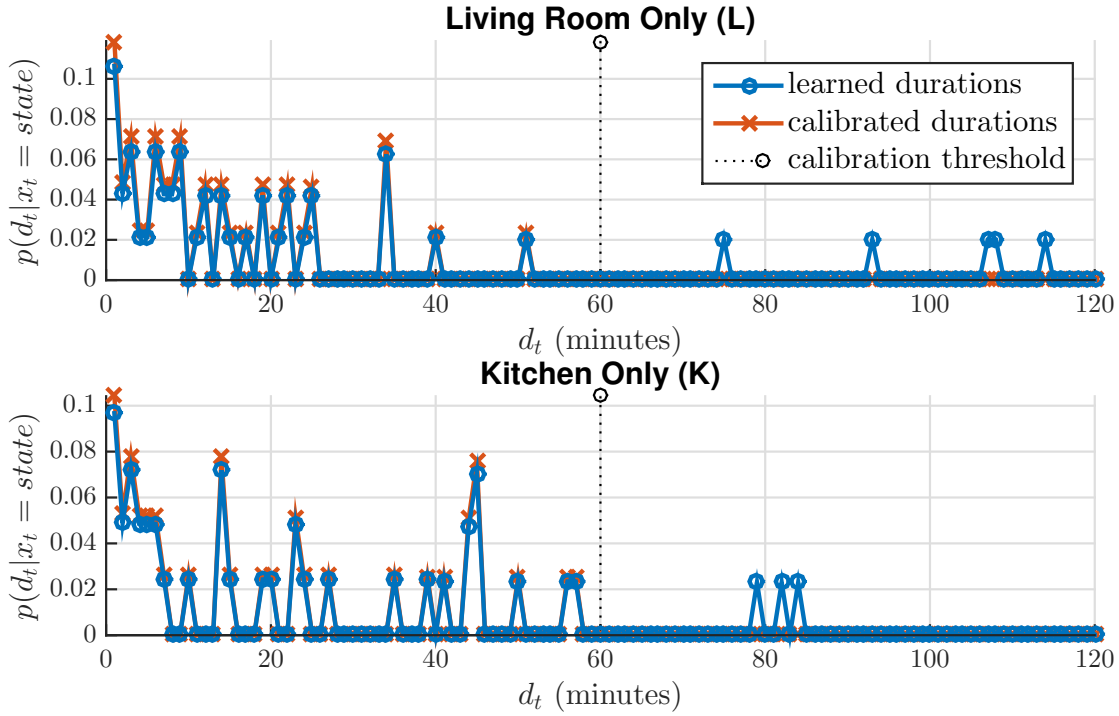


Figure 7.5: Learned and calibrated duration distributions for states L and K at $h = 19$.

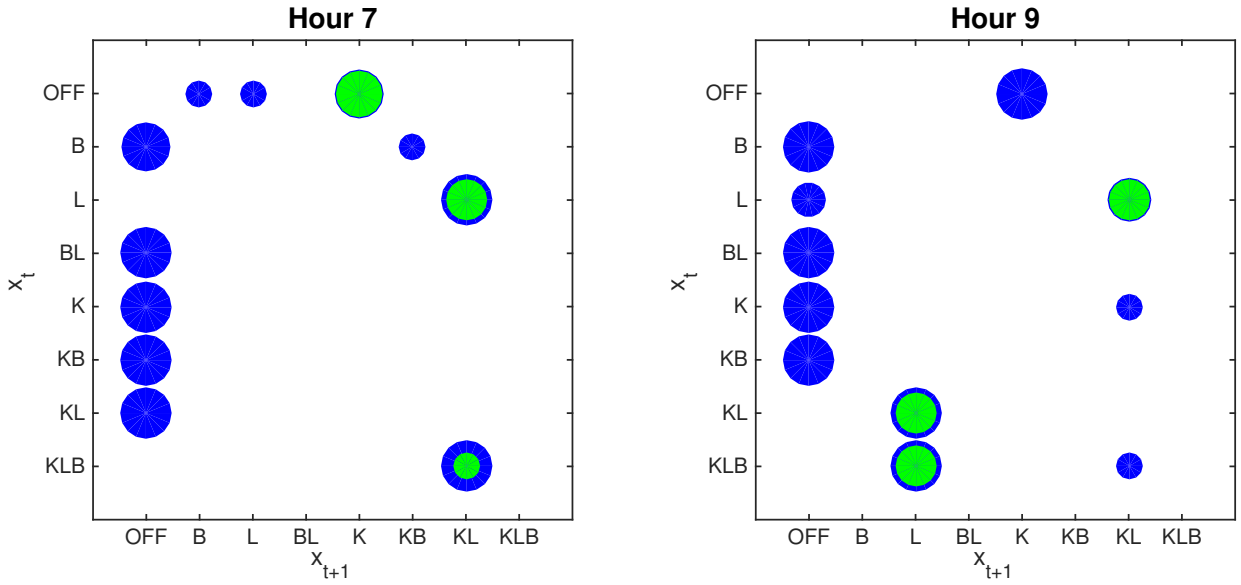


Figure 7.6: Example learned (blue) and calibrated (green) state transition distributions. The corresponding increases in probability for transitions to the OFF state are not shown. See Table 7.1 for state labels.

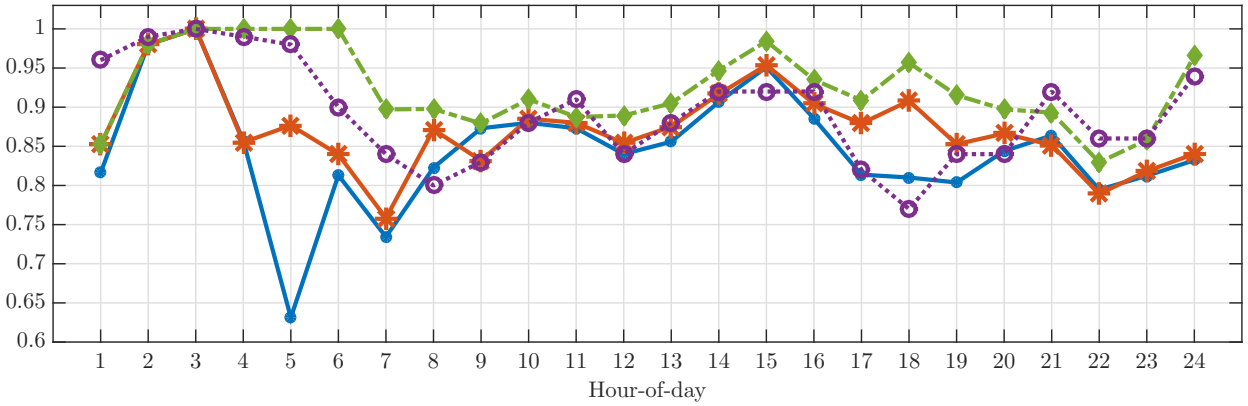


Figure 7.7: Satisfaction probabilities of the hourly soft constraints by improvisers with no calibration (blue), duration calibration (red), and transition calibration (green). For comparison, the empirical satisfaction probabilities of the training data are shown in purple.

22. However, for much of the day the improvement is quite small, and the probabilities are still lower than those of the training data. Transition calibration (purple) gives a significant further improvement, allowing us to finally achieve higher satisfaction probabilities than the training data for most of the soft constraints.

Next, we compare the hourly power consumption profiles of the improvisers with that of the training data in Figure 7.8. The improviser profiles are averaged over 100 20-day improvisations for each type of improviser. Notice that for all three improvisers, the mean power trend (blue curve) matches that of the training data (green curve) quite well, with the last and most highly calibrated improviser having somewhat lower consumption as a result of satisfying the soft constraints most strictly. However, notice that the uncalibrated improviser (at top) has significantly higher variability than the training data, with the hours 9–14 for example having much larger standard deviation (shaded region). Duration calibration ameliorates this problem significantly, and transition calibration eliminates it almost completely.

Finally, to qualitatively compare the similarity of improvised behaviors to the training data, we show several day-long traces in Figure 7.9. Observe that the uncalibrated improvisations are visually similar to the training data, illustrating the quality of the EDHMM as a model. Furthermore, this similarity continues to hold for the calibrated improvisers, showing how our model calibration techniques are effective at enforcing soft constraints without drastically changing system behavior.

7.5 Summary and Future Work

In this chapter, we showed how algorithmic improvisation can be used to synthesize models of human behavior subject to constraints. To make it possible to learn such models from

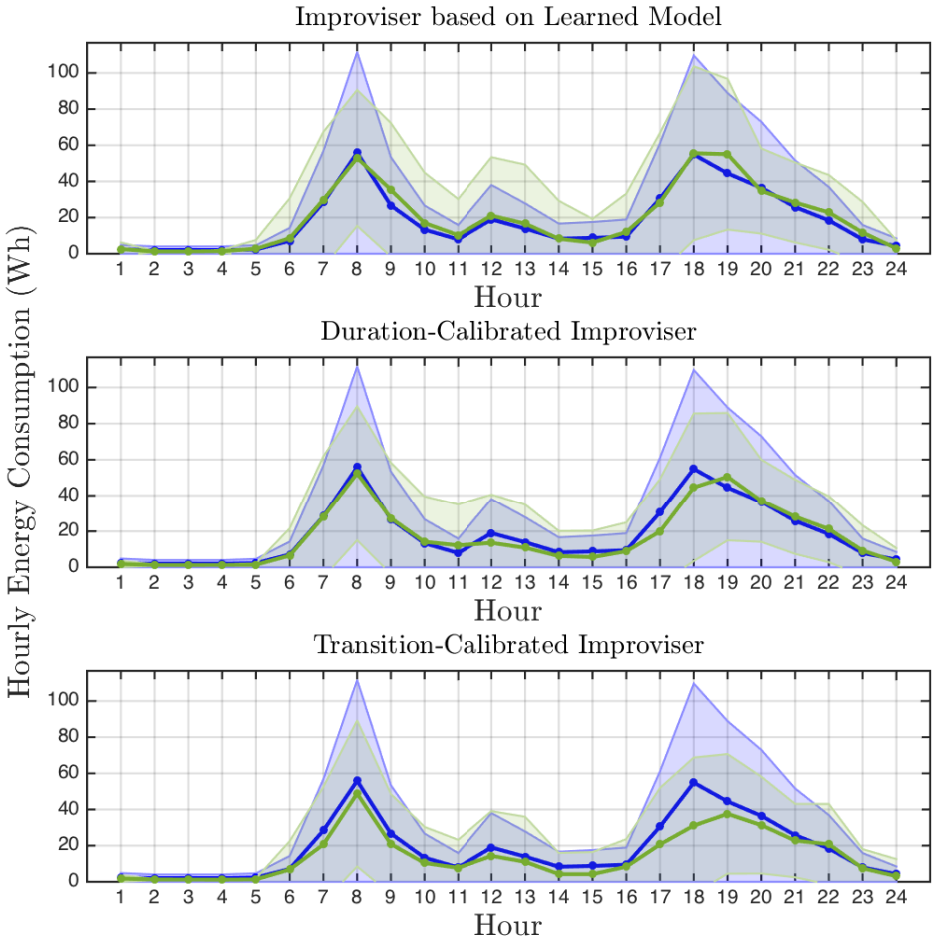


Figure 7.8: Hourly aggregate energy profiles of our improvisers (green), compared to that of the training data (blue). Solid curves indicate average consumption, and the shaded regions show one standard deviation above the mean.

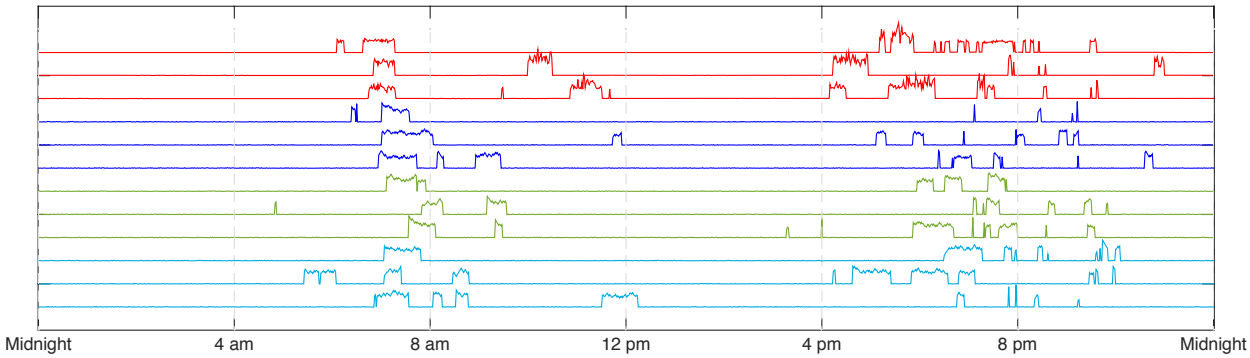


Figure 7.9: Example kitchen appliance traces from the training data (red) and improvisers with no calibration (blue), duration calibration (green), and transition calibration (cyan).

data, we proposed an improvisation procedure which infers an initial model from the training data, then iteratively calibrates the model until the soft constraints are satisfied. Although the procedure is heuristic, we demonstrated its utility in practice by learning a human-like lighting controller subject to soft constraints on power consumption. Our experiments showed that we are able to generate improvisers which satisfy our desired constraints, while remaining both quantitatively and qualitatively faithful to the original training data.

There are several interesting directions for future work:

Other Application Domains. As we mentioned in the Introduction, there are many domains besides home automation where human models are useful and our techniques could be applied. The domain of autonomous driving is particularly interesting: in Chapter 5, we introduced the language SCENIC, suitable for modeling the *static* environments of autonomous cars, and mentioned extending the language to *dynamic* scenarios as future work. Such an extension would require models of various types of agents — humans in particular — and algorithmic improvisation could be used to build these. A first step in this direction has already been taken by Ge and Murray [69], who use our techniques to build a model of when human drivers make lane changes.

Better Model Calibration. The heuristics we proposed for model calibration were quite simple, and moreover required domain knowledge to decide how to best apply them. Two natural questions are whether we can develop heuristics which work for a wider range of soft constraints, and whether we can develop algorithms which *automatically* decide how to calibrate the model. One potential approach would be to use information from the probabilistic model checker about why a soft constraint was violated to decide how to change the model, as in counterexample-guided inductive synthesis (CEGIS) [160, 159].

Stronger Guarantees. More ambitiously, we can ask whether it is possible to dispense with heuristics entirely and develop algorithms which are *guaranteed* to find an improviser if one exists, otherwise proving that there is none (as our algorithms in Chapters 3 and 4 do). More precisely, we could ask whether, given an initial probabilistic model (possibly learned from data), there exists another model which is within a given statistical distance but also satisfies desired hard and soft constraints. This is closely related to the problem of *model repair* [14], where we wish to make minimal changes to a model so that it satisfies a given probabilistic specification.

Chapter 8

Synthetic Data Generation

8.1 Introduction

In this chapter, we explore applications of algorithmic improvisation to the design and analysis of cyber-physical systems. Specifically, we show how language-based improvisation, which we introduced in Chapter 5, can be used to generate synthetic data for systems like autonomous cars and aircraft. We can specify what type of data we want by writing programs in our Scenic language [64], use our algorithm for scene improvisation to generate scenes, and use the resulting data to evaluate the performance of a system under particular conditions or, for systems based on machine learning, fill in a gap in an existing training set. More generally, we can use Scenic to write formal environment models, which are a critical part of any rigorous design process for dependable systems.

In the rest of this section, we give an overview of why designing reliable machine learning-based systems is difficult, how language-based improvisation can help, and earlier approaches to the problem. We also summarize the various applications and domains we have experimented with so far. Next, in Section 8.2, we present our general methodology for training, testing, and debugging cyber-physical systems using language-based improvisation. In Section 8.3, we apply this framework to a practical deep neural network for autonomous driving, improving its performance beyond what is achieved by state-of-the-art synthetic data generation methods. Finally, we conclude in Section 8.4 with a summary and prospects for future work.

8.1.1 Challenges in the Design of Reliable ML-Based Systems

Cyber-physical systems like robots and self-driving cars are increasingly being deployed in complex, uncontrolled environments, while being expected to operate safely without human supervision. To solve the very difficult perception problems arising from real-world environments, such systems often make use of machine learning algorithms. However, this compounds the hard problem of making cyber-physical systems reliable, adding black-box, uninterpretable components to systems which are already highly complex. Thus, there is a

Figure 8.1: Three more scenes of bumper-to-bumper traffic generated from the SCENIC program used in Figure 5.1.

great need for a rigorous design process which would ensure the dependability of ML-based safety-critical systems [149, 155, 5]. In particular, we need techniques to more systematically:

- *train* the system so that it correctly handles events that occur only rarely,
- *test* the system under a variety of conditions, especially unusual ones, and
- *debug* the system to understand the root cause of a failure and eliminate it.

The traditional ML approach to these problems is to gather more data from the environment, retraining the system (and possibly increasing the complexity of the model) until its performance is adequate. The major difficulty here is that collecting real-world data can be slow and expensive, since it must be preprocessed and correctly labeled before use: for example, collecting thousands of traffic images is easy, but to train a car detector, humans must first draw bounding boxes around every car in every image. Furthermore, it may be difficult or impossible to collect data from corner cases that are rare but nonetheless necessary to train and test against: for example, a car accident. As a result, recent work has investigated training and testing systems with *synthetically generated data*, which has two major advantages over real data: it can be produced in bulk without manual labeling, since the ground truth is known during generation, and the designer has full control over the distribution of the data [84, 170, 91].

The main obstruction to the use of synthetic data is that it can be extremely difficult to generate *meaningful* data, since this usually requires detailed modeling of complex environments [155]. Suppose, for example, that we wanted to train a neural network to identify cars in road images. If we simply sampled uniformly at random from all possible configurations of, say, 12 cars, we would get data that was at best unrealistic, with cars facing sideways or backward, and at worst physically impossible, with cars intersecting each other. Instead, we need scenes like those in Figure 8.1, where the cars are laid out in a consistent and realistic way. Furthermore, we may want scenes that are not only realistic but represent particular *scenarios* of interest for training or testing, for example parked cars, cars passing across the field of view, or bumper-to-bumper traffic as in Figure 8.1. In general, we need a way to *guide* data generation toward scenes that makes sense for our application.

8.1.2 Language-Based Improvisation for Data Generation

We argue that language-based improvisation provides a natural solution to this problem. Using a probabilistic programming language, the designer of a system can construct distributions representing different input regimes of interest, and sample from these distributions to obtain concrete inputs for training and testing. Specifically, LBI can help with all three design problems mentioned above:

- *training* a system to better handle rare events by generating many instances of such events to include in the training set,
- *testing* a system under conditions of interest by writing programs describing those conditions, and
- *debugging* a system by improvising variations on a known failure case to identify its cause, then generating a broader set of failing inputs suitable for retraining.

More generally, as we noted in Chapter 5, a program in a PPL can be used as an environment model, providing a formal specification of the distribution of environments under which a system must safely operate with high probability.

In this chapter, we focus on systems whose environment is a *scene*, a configuration of objects in space (including dynamic agents, such as vehicles). As we saw in Chapter 5, careful design of a domain-specific PPL can make it possible to express complex distributions over scenes in a concise and readable way. This is clearly demonstrated by the scenes in Figure 8.1, which were generated from a 20-line Scenic program, and we will see further examples later in this chapter.

Our proposed approach for using language-based improvisation to train, test, and debug systems critically depends on the PPL we use being capable of encoding a wide range of general and specific environment scenarios. This is the case for Scenic: the variety of constructs it provides makes it possible to write scenarios anywhere on a spectrum from concrete scenes (i.e. individual test cases) to extremely broad classes of abstract scenes, as shown in Figure 8.2. A scenario can be reached by moving along the spectrum from either end: top-down, by progressively constraining a very general scenario, or bottom-up, by generalizing from a concrete example (such as a known failure case), for example by adding random noise. Probably most usefully, one can write a scenario in the middle which is far more general than simply adding noise to a single scene, but has much more structure than a completely random scene: for example, the traffic scenario

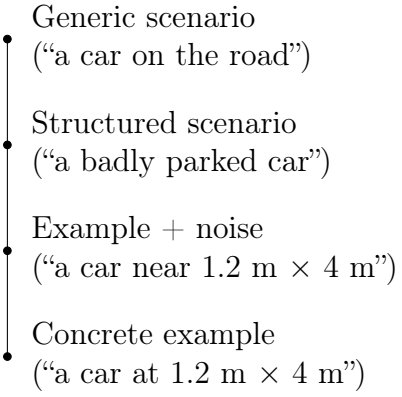


Figure 8.2: Spectrum of Scenic scenarios, general to specific.

depicted in Figure 8.1. Later in the chapter we will illustrate all three ways of developing a scenario, which as we will see are useful for different training, testing, and debugging tasks.

8.1.3 Related Work

Data Generation and Testing for ML

There has been a large amount of work on generating synthetic data for specific applications, including text recognition [88], text localization [84], robotic object grasping [170], and autonomous driving [91, 55]. Closely related is work on *domain adaptation*, which attempts to correct differences between synthetic and real-world input distributions. Domain adaptation has enabled synthetic data to successfully train models for several other applications including 3D object detection [109, 162], pedestrian detection [179], and semantic image segmentation [146]. All of this work provides important context for our setting, demonstrating that models trained exclusively on synthetic data (possibly domain-adapted) can achieve acceptable performance on real-world data. The major difference in our work is that our methodology is not specific to any particular application, but, through Scenic, provides a general way to do language-based systematic data generation for *any* system whose environment is a scene.

Another line of work has explored using adversarial (i.e. misclassified) examples to retrain and improve ML models [192, 187, 76]. Some of these methods use optimization to find minimal perturbations of given inputs which lead to a misclassification [166, 124, 129], or to systematically trigger different behaviors of the network [133]. Another approach uses Generative Adversarial Networks [75], a kind of neural network able to generate synthetic data, to augment training sets [108, 115]. These techniques usually require pre-existing training sets, whereas our approach requires only a simulator. Furthermore, they search through the input space of the network, e.g. raw images, unlike our work, which uses a high-level *semantic* space [49] like that of scenes. In this respect, we follow another line of work which searches for misclassifications in a semantic space using various types of random and systematic sampling techniques [46, 48]. However, none of these approaches provides any detailed control over what type of synthetic data is generated. Scenic, on the other hand, gives complete control over the distribution of the generated data, including being able to impose declarative constraints, and does so in a flexible and explainable manner.

Model-Based Generation of Tests and Graphics

There is a long history of techniques which use a model to guide the generation of content, both in testing [20] and computer graphics [51]. A wide variety of different types of models have been considered, ranging in complexity and expressivity. A simple and popular approach is to use *examples* as a model, as in mutational fuzz testing [165] and example-based scene synthesis [57] (many of the data augmentation techniques discussed above also essentially use this kind of model). While examples are easy to use, they do not provide

fine-grained control over the generated data. Another class of models uses *rules* or a *grammar* to specify how the data can be generated, as in generative fuzz testing [165], procedural generation from shape grammars [125], and grammar-based scene synthesis [90]. Grammars allow much greater control than a set of examples, but they do not easily allow enforcing global properties. The same is true of most models based on *programming languages*, for example domain-specific nondeterministic languages for test generation [52] and procedural modeling languages in computer graphics [180]. Conversely, models given by *constraints*, as in constrained-random verification [127], allow global properties but can be difficult to write. Using a *probabilistic programming language* model, Scenic improves on these techniques by simultaneously providing fine-grained control, enforcement of global properties, specification of probability distributions, and simple imperative syntax.

Probabilistic Programming Languages

The semantics (and to some extent, the syntax) of Scenic are similar to that of other probabilistic programming languages such as Prob [79], Church [77], and BLOG [123]. In probabilistic programming the focus is usually on *inference* rather than *generation* (the main application in our case), and in particular to our knowledge probabilistic programming languages have not previously been used for test generation. However, the most popular inference techniques are based on sampling and so could be directly applied to generate scenes from Scenic programs, as we discussed in Chapter 5.

Several PPLs have been used to define generative models of objects and scenes: both general-purpose languages such as WebPPL [78] (see, e.g., Ritchie [143]) and languages specifically motivated by such applications, namely Quicksand [144] and Picture [102]. The latter are in some sense the most closely-related to Scenic, although neither provides specialized syntax or semantics for dealing with geometry (Picture also was used only for inverse rendering, not data generation). The main advantage of Scenic over these languages is that its domain-specific design permits concise representation of complex scenarios and enables specialized sampling techniques.

8.1.4 Case Studies

We have successfully applied Scenic to help design and analyze a number of different systems. Our main case study is on SqueezeDet [189], a convolutional neural network which is being used industrially for object detection in autonomous cars[1]. For this task, it has been shown [91] that good performance on real images can be achieved with networks trained purely on synthetic images from the video game Grand Theft Auto V (GTA V [68]). We wrote an interface between Scenic and GTA V, allowing us to import generated scenes into the game and render images. Our experiments demonstrate using Scenic to:

[1]For example by DeepScale (http://deepscale.ai/).

- evaluate the accuracy of the system under particular conditions: we find that the network performs worse in dark, rainy weather than bright, clear weather;
- improve performance on corner cases: we use SCENIC to both identify a deficiency in the state-of-the-art car detection data set of Johnson-Roberson et al. [91], and generate a new training set of equal size but yielding significantly better performance;
- debug a failure case: we use SCENIC to find an image the network misclassifies, discover the root cause, and fix the bug, in the process improving the network's performance on its original test set (again, without increasing training set size).

These experiments show that SCENIC can be a very useful tool for understanding and improving ML-based perception systems.

We stress that while our GTA V case study is performed in the domain of visual perception for autonomous driving, and uses one particular simulator, our methodology is not specific to either. As we explained in Chapter 5, SCENIC can produce data of any desired type (e.g. LIDAR point clouds) by interfacing it to an appropriate simulator. In fact, we have already experimented with several different simulators and domains:

- We demonstrated the use of SCENIC to test *controllers* as well as perception systems, using the interface to the Webots simulator [122] mentioned in Chapter 5. We found bugs in a simple neural network-based collision-avoidance system, and also simulated variations on an actual crash involving an autonomous car (using map and lane data from OpenStreetMap [59] and the Intelligent Intersections Toolkit [82]). For details on these experiments, conducted using the VERIFAI toolkit for formal design and analysis of AI-based systems (which uses SCENIC as an environment modeling language), see Dreossi et al. [47].
- In collaboration with Boeing, we used SCENIC to test an experimental neural network-based automated taxiing system for aircraft in the X-Plane flight simulator [141]. We found that the system is highly sensitive to differences in lighting and weather conditions (see Figure 8.3 for an example), and are currently using SCENIC to retrain the system to be more robust[2].
- Finally, we have also developed a prototype interface to the CARLA simulator [45] to enable more sophisticated autonomous driving experiments.

8.2 Using Scene Improvisation to Design and Analyze Cyber-Physical Systems

We propose a methodology for training, testing, and debugging cyber-physical systems using language-based improvisation. The core idea is to use a probabilistic programming language

[2]Thanks to Johnathan Chiu for helping to implement the X-Plane interface and run experiments.

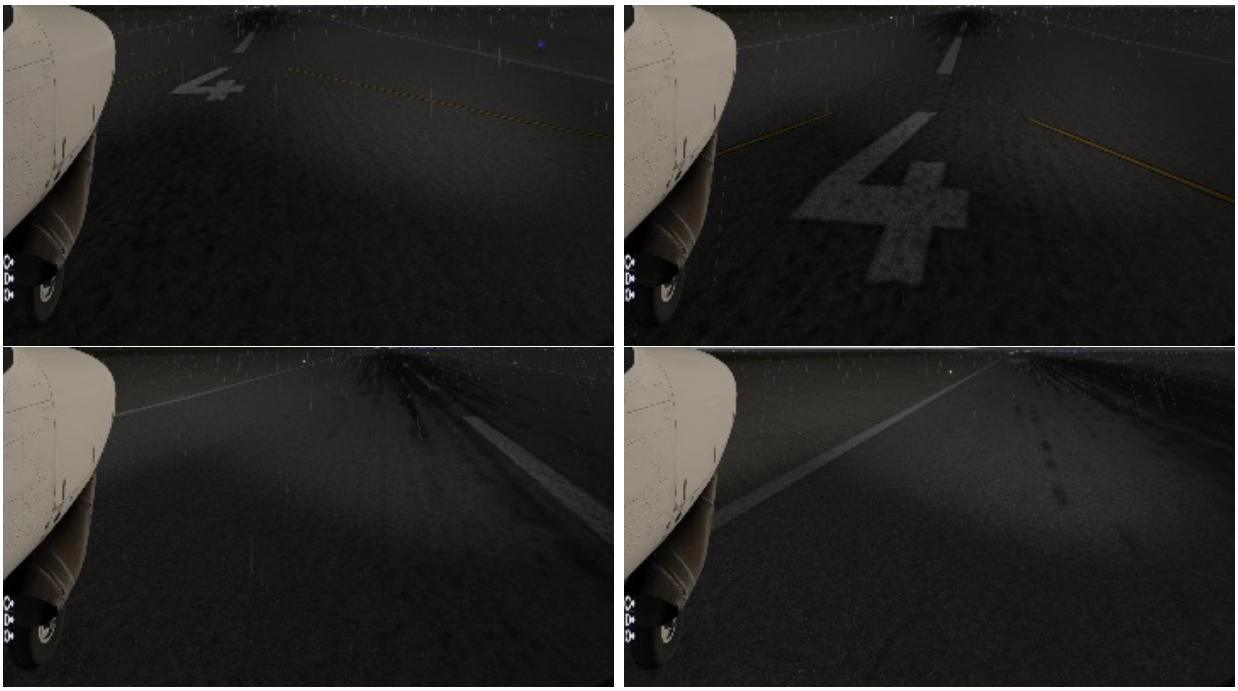


Figure 8.3: Frames (4 seconds apart) from an X-Plane simulation where an automated taxiing system gets confused by dark, rainy conditions and swerves off the runway.

to formalize general operation scenarios, then sample from these distributions to generate concrete environment configurations. Putting these configurations into a simulator, we obtain sensor data which can be used to test and train a perception system, or we run dynamic simulations to test/train a controller or closed-loop system. In particular, for systems like autonomous cars whose environment is a scene, we can write scenarios in Scenic, use scene improvisation to generate concrete scenes, and render these into images using a simulator. The general procedure is outlined in Figure 8.4.

Note that as shown in the Figure, the training and testing data sets need not be purely synthetic: we can generate data to supplement existing real-world data, for example to provide more instances of a situation which is not well-represented in the original training set. Furthermore, even for models trained purely on real data, synthetic data can still be useful for testing and debugging, as we will see below. Now we discuss how our methodology addresses the three design problems from the Introduction in more detail.

Testing under Different Conditions

The most straightforward problem is that of assessing system performance under different conditions. We can simply write scenarios capturing each condition, generate a test set from each one, and evaluate the performance of the system on these. Note that conditions which

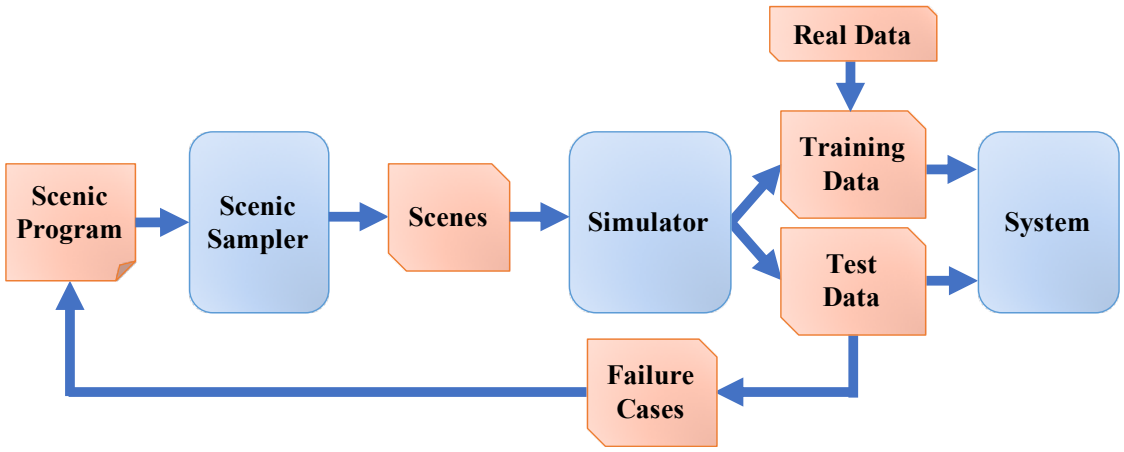


Figure 8.4: Tool flow using scene improvisation to train, test, and debug a cyber-physical system.

occur rarely in the real world present no additional problems: as long as the PPL we use can encode the condition, we can generate as many instances as desired.

Training on Rare Events

Extending the previous application, we can use this procedure to help ensure the system performs adequately even in unusual circumstances or particularly difficult cases. Writing a scenario capturing these rare events, we can generate instances of them to augment or replace part of the original training set. Emphasizing these instances in the training set can improve the system's performance in the hard case without impacting performance in the typical case. In Section 8.3.3 we will demonstrate this for car detection, where an obvious hard case is when one car partially overlaps another in the image. We wrote a SCENIC program to generate a set of these overlapping images. Training the car-detection network on a state-of-the-art synthetic dataset obtained by randomly driving around inside the simulated world of GTA V and capturing images periodically [91], we find its performance is significantly worse on the overlapping images. However, if we keep the training set size fixed but increase the proportion of overlapping images, performance on such images dramatically improves *without harming performance on the original generic dataset*. This shows the benefit of designing a training set in an intelligent, deliberate manner, which is made possible by the detailed control over distributions provided by language-based improvisation.

Debugging Failures

Finally, we can use the same procedure to help understand and fix bugs in the system. If we find an environment configuration where the system fails, we can write a scenario which reproduces exactly that particular configuration. Having the configuration encoded

as a program then makes it possible to explore the neighborhood around it in a variety of different directions, leaving some aspects of the scene fixed while varying others. This can give insight into which features of the scene are relevant to the failure: for example, we might find that the failure happens regardless of the color of a car, but depends sensitively on the car model. By performing several experiments of this type iteratively, we can eventually converge on the root cause of the failure. The root cause can then itself be encoded into a scenario which generalizes the original failure, allowing retraining without overfitting to the particular counterexample we originally found. We will demonstrate this approach in Section 8.3.4, starting from a single misclassification, identifying a general deficiency in the training set, replacing part of the training data to fix the gap, and ultimately achieving higher performance on the original test set.

8.3 Case Study: Neural Networks for Car Detection

We demonstrate the methodology described above in a case study on a practical neural network for object detection, using synthetic data from GTA V. We begin by describing the general setup for all of our experiments in Section 8.3.1, then discuss testing, training, and debugging in Sections 8.3.2, 8.3.3, and 8.3.4 respectively.

8.3.1 Experimental Setup

Simulator Interface

We generated scenes in the virtual world of the video game Grand Theft Auto V [68], which has previously been used in computer vision experiments because of its photorealistic rendering and highly-detailed world [142, 55, 91]. Although it is possible to write Scenic scenarios which randomly position roads, signs, buildings, and other kinds of objects, for our experiments we only generated configurations of cars, laid out within the fixed GTA V world. To interface Scenic to GTA V, we needed to carry out the two steps mentioned in Chapter 5: writing a Scenic library defining the GTA V world, and an interface layer importing the generated scenes into GTA V.

For the first step, we wrote a library `gta` defining:

- `Regions` `road` and `curb` representing the roads and curbs in (part of) the GTA V world. Since GTA V does not provide an explicit representation of its map, we obtained an approximate map by processing a bird's-eye schematic view of the game world[3]. To identify points on a road, we converted the image to black and white, effectively turning roads white and everything else black. We then used edge detection to find curbs. Since the resulting road information was imperfect, some generated scenes placed cars in undesired places such as sidewalks or medians, and we had to manually filter the

[3] http://gta-5-map.com/

generated images to remove these. With a real simulator, e.g. Webots, this is not necessary.

- A vector field `roadDirection` representing the nominal traffic direction at each point on a road. This was computed from the map above by finding for each curb point X the nearest curb point Y on the other side of the road, and assuming traffic flows perpendicular to the segment XY (this was more robust than using the directions of the edges in the image).

- A type of object `Car` defined as follows (slightly simplified):

```
class Car:
    position: Point on road
    heading: (roadDirection at self.position) + self.roadDeviation
    roadDeviation: 0
    width: self.model.width
    height: self.model.height
    viewAngle: 80 deg      # view angle of GTA V camera
    visibleDistance: 30    # by default, put other cars within 30 m
    model: CarModel.defaultModel()
    color: CarColor.defaultColor()
```

  The `roadDeviation` property is a convenience, allowing us to specify a car's heading relative to the local road direction: we can write `Car with roadDeviation 10 deg` instead of `Car facing 10 deg relative to roadDirection`, for example. The `model` property, representing the type of car, has by default a uniform distribution over 13 diverse models provided by GTA V. The `color` property has a default distribution based on real-world car color statistics [50].

- Global parameters `time` and `weather` representing the time of day and weather. The default distribution for `time` is uniform over the entire day, while for `weather` it is a non-uniform distribution over the 14 discrete weather types supported by GTA V, e.g. "clear" and "snow", with less weight on more extreme types.

Since GTA V is closed-source and does not expose any kind of scene description language. Therefore, to import scenes generated by Scenic into GTA V, we wrote a plugin based on DeepGTAV [148]. The plugin calls internal functions of GTA V to create cars with the desired positions, colors, etc., as well as to set the camera position, time of day, and weather.

Perception System and Performance Metrics

We conducted our experiments on SqueezeDet [189], a convolutional neural network real-time object detector for autonomous driving. The task of the network in our experiments was to place 2D bounding boxes around every car in an image. The output of the network

is a list of bounding boxes, with associated confidence scores. When training, we used a batch size of 20, and trained all models for 10,000 iterations unless otherwise noted. Images captured from GTA V at 1920×1200 resolution were resized to 1248×384, the resolution used by SqueezeDet and the standard KITTI benchmark [70]. All models were trained and evaluated on NVIDIA TITAN Xp GPUs.

We used the metrics *precision* and *recall* to measure the accuracy of the network on a given set of images. Intuitively, precision measures how many predicted bounding boxes are correct, while recall measures how many objects are actually detected. To define these formally, we used a two-step process standard in object detection [53, Section 4.4]. First, a predicted bounding box B is said to *match* a ground truth box B' if their *Intersection over Union (IoU)* score, $\text{area}(B \cap B')/\,\text{area}(B \cup B')$, is at least $1/2$ (for example, disjoint boxes have an IoU of zero, and identical boxes have an IoU of 1). Second, we go through all predicted boxes for a given image in order of decreasing confidence: the box is a *true positive* if it matches a ground truth box that has not already been matched (since each object should only be detected once), and otherwise it is a *false positive*. If any ground truth box remains unmatched, it is a *false negative*. If TP and FP are the numbers of true/false positives over the entire image set, and FN likewise is the number of false negatives, then the precision is $TP/(TP + FP)$ and the recall is $TP/(TP + FN)$.

Note that some work on object detection uses a different metric, AP (which stands for *Average Precision*, but is *not* simply the average of the precision over the test images). For most of our experiments we use precision/recall rather than AP because the latter is not sensitive to the main type of failure we discovered, where the network correctly detects most objects but also outputs spurious boxes (false positives) with lower confidence than the correct predictions. However, for the experiment in Section 8.3.3 we report results in both metrics, since our baseline training set comes from Johnson-Roberson et al. [91], who use AP in their experiments. We used the tool of Cartucho [24] to compute AP.

8.3.2 Testing under Different Conditions

When testing a model, one may be interested in a particular operation regime. For instance, an autonomous car manufacturer may be more interested in certain road conditions (e.g. desert vs. forest roads) depending on where its cars will be mainly used. A designer may also wish to do extra testing in simulation of cases that are difficult or dangerous to reproduce in reality, like construction sites and car accidents. SCENIC provides a systematic way to describe such scenarios of interest and construct corresponding test sets.

To demonstrate this, we first wrote very general scenarios describing scenes of 1–4 cars (not counting the camera), specifying only that the cars face within 10° of the road direction. The 4-car scenario is shown in Figure 8.5 (note the use of the `resample` function to conveniently give each car an independent heading sampled from the same distribution). We generated 1,000 images from each scenario, yielding a training set X_{generic} of 4,000 images, and used these to train a model M_{generic} as described in Section 8.3.1. We also generated an additional 50 images from each scenario to obtain a generic test set T_{generic} of 200 images.

```
wiggle = (-10 deg, 10 deg)    # perturbation around road direction
ego = Car with roadDeviation wiggle
for i in range(4):    # create 4 cars visible in camera
    Car visible, with roadDeviation resample(wiggle)
```

Figure 8.5: A generic scenario of four cars roughly aligned with the road.

Table 8.1: Performance of M_{generic} under different road conditions.

Test Set	Precision	Recall
Generic (T_{generic})	83.1%	92.6%
Bright, sunny (T_{good})	85.7%	94.3%
Dark, rainy (T_{bad})	72.8%	92.8%

Next, we specialized the general scenarios in opposite directions, creating scenarios for good and bad road conditions. Specifically, we added either the lines

```
param weather = 'EXTRASUNNY'    # bright, with no clouds
param time = 12 * 60            # time of day: noon
```

for good conditions, and

```
param weather = 'RAIN'    # rainy weather
param time = 0 * 60       # time of day: midnight
```

for bad. Using these scenarios, we generated specialized test sets T_{good} and T_{bad}. Example images are shown in Figure 8.6.

Evaluating M_{generic} on T_{generic}, T_{good}, and T_{bad}, we obtained the results shown in Table 8.1. As might be expected, the model performs better on bright days than on rainy nights. This suggests there might not be enough examples of rainy nights in the training set, and indeed under our default weather distribution rain is less likely than shine. This illustrates how specialized test sets can highlight the weaknesses and strengths of a particular model. In the next section, we go one step further and use SCENIC to redesign the training set and improve model performance.

8.3.3 Training on Rare Events

In the synthetic data setting, we are limited not by data availability but by the cost of training. The natural question is then how to generate a synthetic data set that as effective as possible given a fixed size. In this section we show that *over-representing* a type of input that may occur rarely but is difficult for the model can improve performance on the hard

Figure 8.6: Scenes of four cars in good (above) and bad (below) driving conditions.

case without compromising performance in the typical case. SCENIC makes this possible by allowing the user to write a scenario capturing the hard case specifically.

For our car detection task, an obvious hard case is when one car substantially occludes another. We wrote a simple scenario, shown in Figure 8.7, which generates such scenes by placing one car behind the other as viewed from the camera, offset to left or right so that it is at least partially visible. Generating images from this scenario, we obtained a training set X_{overlap} of 250 images and a test set T_{overlap} of 200 images. Example images are shown in Figure 8.8.

For a baseline training set we used the "Driving in the Matrix" synthetic data set of Johnson-Roberson et al. [91], which has been shown to yield good car detection performance even on real-world images. Like our images, the "Matrix" images were rendered in GTA V; however, rather than using a PPL to guide generation, they were produced by allowing the game's AI to drive around randomly while periodically taking screenshots. We randomly selected 5,000 of these images to form a training set X_{matrix}, and 200 for a test set T_{matrix}. We trained SqueezeDet for 5,000 iterations on X_{matrix}, evaluating it on T_{matrix} and T_{overlap}. To reduce the effect of jitter during training we used a standard technique [7], saving the last 10 models in steps of 10 iterations and picking the one achieving the best total precision and recall. This yielded the results in the first row of Table 8.2. Although X_{matrix} contains many

```
wiggle = (-10 deg, 10 deg)
ego = Car with roadDeviation wiggle
c = Car visible,
        with roadDeviation resample(wiggle)
leftRight = Uniform(1.0, -1.0) * (1.25, 2.75)
Car beyond c by leftRight @ (4, 10),
    with roadDeviation resample(wiggle)
```

Figure 8.7: A scenario where one car partially occludes another.

Figure 8.8: Scenes generated from the occlusion scenario in Figure 8.7.

Table 8.2: Performance of models trained on 5,000 images from X_{matrix} or a mixture with X_{overlap}, averaged over 8 training runs with random selections of images from X_{matrix}.

Mixture %	T_{matrix} Precision	T_{matrix} Recall	T_{matrix} AP	T_{overlap} Precision	T_{overlap} Recall	T_{overlap} AP
100 / 0	**72.9** $\pm$ 3.7	37.1 $\pm$ 2.1	36.1 $\pm$ 1.1	**62.8** $\pm$ 6.1	65.7 $\pm$ 4.0	61.7 $\pm$ 2.2
95 / 5	**73.1** $\pm$ 2.3	37.0 $\pm$ 1.6	36.0 $\pm$ 1.0	**68.9** $\pm$ 3.2	67.3 $\pm$ 2.4	65.8 $\pm$ 1.2

images of overlapping cars, the precision on T_{overlap} is significantly lower than for T_{matrix}, indicating that the network is predicting spurious bounding boxes for such cars[4].

Next we attempted to improve the effectiveness of the training set by mixing in the difficult images produced with SCENIC. Specifically, we replaced a random 5% of X_{matrix} (250 images) with images from X_{overlap}, keeping the overall training set size constant. We then retrained the network on the new training set and evaluated it as above. To reduce the dependence on which images were replaced, we averaged over 8 training runs with different random selections of the 250 images to replace. The results are shown in the second row of Table 8.2. Even altering only 5% of the training set, performance on T_{overlap} significantly improves. Critically, the improvement on T_{overlap} is not paid for by a corresponding decrease on T_{matrix}: performance on the original data set remains the same. Both results remain true if we measure performance using the AP metric, as in Johnson-Roberson et al. [91]. Thus, by allowing us to specify and generate instances of a difficult case, SCENIC enables the generation of more effective training sets than can be obtained through simpler approaches not based on PPLs.

8.3.4 Debugging Failures

In our final experiment, we show how SCENIC can be used to generalize a single input on which a model fails, exploring its neighborhood in a variety of different directions and giving insight into which features of the scene are responsible for the failure. The original failure can then be generalized to a broader scenario describing a class of inputs on which the model misbehaves, which can in turn be used for retraining.

First, we selected one scene from our first experiment (Section 8.3.2), consisting of a single car viewed from behind at a slight angle, which M_{generic} wrongly classified as three cars (thus having 33.3% precision and 100% recall). The misclassified image is shown in Figure 8.9. Extracting the positions and other properties of the cars from the original generated scene, we wrote a SCENIC scenario, shown in Figure 8.10, which exactly reproduces the scene.

[4]Recall and AP are much *higher* on T_{overlap}, meaning the false-negative rate is better; this is presumably because all the T_{overlap} images have exactly 2 cars and are in that sense easier than the T_{matrix} images, which can have many cars.

Figure 8.9: An image misclassified as having three cars, with the predicted bounding boxes.

```
param time = 12 * 60
param weather = 'EXTRASUNNY'

ego = Car at -628.78787878787944 @ -540.60676779463461,
          facing -359.16913666080427 deg

c = Car at -625.4444493298472 @ -530.76549003839568,
        facing 8.287256822061408 deg,
        with model CarModel.models['DOMINATOR'],
        with color CarColor.withBytes([187, 162, 157])
```

Figure 8.10: Scenario reproducing the misclassified image in Figure 8.9.

Table 8.3: Performance of M_{generic} on different variants of the scenario in Figure 8.10.

Scenario	Precision	Recall
(1) varying model and color	**80.3**	100
(2) varying background	50.5	99.3
(3) varying local position, orientation	62.8	100
(4) varying position but staying close	53.1	99.3
(5) any position, same apparent angle	58.9	98.6
(6) any position and angle	67.5	100
(7) varying background, model, color	61.3	100
(8) staying close, same apparent angle	52.4	100
(9) staying close, varying model	58.6	100

We then generalized this scenario in several ways, leaving most of the features of the scene fixed but allowing others to vary. Specifically, scenario (1) varied the model and color of the car, (2) left the position and orientation of the car relative to the camera fixed but varied the absolute position, effectively changing the background of the scene, and (3) used the mutation feature of SCENIC to add a small amount of noise to the car's position, heading, and color. For each scenario we generated 150 images and evaluated M_{generic} on them. As seen in the first three rows of Table 8.3, changing the model and color improved performance the most, suggesting they were most relevant to the misclassification, while local position and orientation were less important and global position (i.e. the roads and scenery in the background of the image) was least important.

To investigate these possibilities further, we wrote a second round of variant scenarios, obtaining the results shown in rows 4–7 of Table 8.3. These confirmed the importance of model and color (compare (2) to (7)), as well as angle (compare (5) to (6)), but also suggested that being close to the camera could be the relevant aspect of the car's local position. We confirmed this with a final round of scenarios (compare (5) and (8) in the Table), which also showed that the effect of car model is small among scenes where the car is close to the camera (compare (4) and (9)).

Having established that car model, closeness to the camera, and view angle all contribute to poor performance of the network, we wrote broader scenarios capturing these features. To avoid overfitting, and since our experiments indicated car model was not very relevant when the car is close to the camera, we decided not to fix the car model. Instead, we specialized the generic one-car scenario from our first experiment to produce only cars close to the camera. We also created a second scenario specializing this further by requiring that the car be viewed at a shallow angle. This scenario is shown in Figure 8.11.

Finally, we used these scenarios to retrain M_{generic}, hoping to improve performance on its original test set T_{generic} (to better distinguish small differences in performance, we increased the test set size to 400 images). To keep the size of the training set fixed as in the previous

```
wiggle = (-10 deg, 10 deg)
ego = Car with roadDeviation wiggle
c = Car offset by (-5, 5) @ (7, 12),
        with roadDeviation resample(wiggle)
require abs((apparent heading of c) - 27 deg) <= 10 deg
```

Figure 8.11: Scenario of a single car close to the camera, viewed at a shallow angle. The "close car" scenario is almost identical, simply omitting the `require` statement.

Table 8.4: Performance of M_{generic} after retraining, replacing 10% of X_{generic} with new data.

Replacement Data	Precision	Recall
Original (no replacement)	82.9	92.7
Classical augmentation	78.7	92.1
Close car	87.4	91.6
Close car at shallow angle	84.0	92.1

experiment, we replaced 400 one-car images in X_{generic} (10% of the whole training set) with images generated from our retraining scenarios. As a baseline, we used images produced with classical image augmentation techniques implemented in `imgaug` [92]. Specifically, we modified the original misclassified image by randomly cropping 10%–20% on each side, flipping horizontally with probability 50%, and applying Gaussian blur with $\sigma \in [0.0, 3.0]$.

The results of retraining M_{generic} on the resulting data sets are shown in Table 8.4. Interestingly, classical augmentation actually *hurt* performance, presumably due to overfitting to relatively slight variants of a single image. On the other hand, replacing part of the data set with specialized images of cars close to the camera significantly reduced the number of false positives like the original misclassification (while the improvement for the "shallow angle" scenario was less, perhaps due to overfitting to the restricted angle range). This demonstrates how Scenic can be used to improve performance by generalizing individual failures into scenarios that capture the essence of the problem but are broad enough to prevent overfitting during retraining.

8.4 Summary and Future Work

In this chapter, we presented a methodology for using language-based improvisation to design and analyze cyber-physical systems. Using a probabilistic programming language to express different environment distributions, we can generate specialized test sets, improve the effectiveness of training sets by intelligently designing their composition, and generalize from individual failure cases to broader scenarios suitable for retraining. We demonstrated

all of these applications, applying SCENIC to generate traffic images for a practical neural network object detector for autonomous cars. In particular, we were able to boost the performance of the network (given a fixed training set size) significantly beyond what could be achieved by prior synthetic data generation techniques [91].

For future work, there are several promising directions. On the language side, many of the generalizations of SCENIC mentioned as future work in Chapter 5 would be useful for the data generation application, for example extending the language with dynamics and learning SCENIC programs from data. There are also some interesting directions to explore on the application side:

Other Systems and Domains. Our methodology is quite general and could be applied to many types of systems. As we mentioned in Section 8.1.4, besides our GTA V case study we have already applied SCENIC to a simple collision-avoidance system for cars and an automated taxiing system for aircraft. Going forward, we plan to apply our techniques to more realistic closed-loop systems, like the driving agents participating in the CARLA autonomous driving challenge [145].

Searching for Corner Cases. A major goal of SCENIC is to give system developers the ability to encode their domain knowledge about interesting and potentially-problematic scenarios in a formal model, which can then be used for test generation or other kinds of analysis. However, SCENIC does not address the issue of *which program* to write: we wrote all of the scenarios above by hand, and a developer could easily write SCENIC programs which completely miss some important failure case. An urgent question is therefore how we can use *automated analysis* of the system to find previously-unsuspected bugs, while still taking advantage of the domain knowledge expressed by a SCENIC program. One possible approach, which we are currently exploring in the VERIFAI toolkit [47], is to allow parameters of SCENIC programs to be explored by *active sampling* techniques like Bayesian optimization [156], rather than purely random sampling. Optimization techniques used in temporal logic falsification [128] or adversarial analysis of neural networks [166] could be used in a similar way.

Chapter 9

Conclusion

Algorithmic improvisation is a new and fruitful subject, with a rich theory and a broad range of applications in robotics, cybersecurity, software engineering, music, machine learning, and many other areas. In this thesis we established the core theory of algorithmic improvisation and explored several of its applications, namely synthesizing randomized planners for robots, learning models of human behavior subject to constraints, and generating synthetic data to help design, test, and debug cyber-physical systems. Despite this, so far we have only scratched the surface of algorithmic improvisation: in both theory and practice, there are numerous directions for future work.

On the theory side, among the open problems we mentioned in Chapters 3–5, several theoretical questions arose multiple times in the applications we studied:

Generalized Soft and Randomness Constraints. In Chapters 3 and 4 we described several ways the concept of control improvisation could be generalized to enable new applications. Two extensions in particular are of particular interest: generalizing the soft constraint to an arbitrary quantitative constraint, for example minimizing the expected value of a cost function, and generalizing the randomness constraint to bounding a measure like entropy or more powerful distributional constraints. Another natural type of randomness constraint would be to require that the distribution of the improviser is as close as possible to a desired distribution, with respect to a metric like total variation distance: so for example we could start with a generative process like the factor oracle in the music improvisation application (see Section 3.4.1) or the EDHMM in the lighting control application (Chapter 7), then try to minimally change its distribution so as to satisfy the desired soft constraint. Such extensions would likely require different techniques than those we have used so far.

Continuous Time and Alphabets. A more drastic extension would be to study control improvisation with continuous time instead of discrete sequences, as well as continuous alphabets, possibly even with a metric defined on improvisations. This would be the natural setting for an extension of scene improvisation with dynamics, as we discussed

in Chapter 5, as well as test generation problems for cyber-physical systems which take signals as input.

Symbolic Algorithms. Finally, as we observed in Chapters 3 and 4, our efficient improvisation schemes for CI and RCI with DFA specifications both suffer from problem of state explosion when multiple requirements are conjoined. While we described a symbolic algorithm for CI based on SAT solvers, we have not yet evaluated whether it works in practice on problems like multi-robot surveillance in Chapter 6: it is possible that model counting algorithms are not yet powerful enough to solve the resulting queries. Furthermore, we do not have an equivalent symbolic algorithm for RCI: although it is plausible that a suitable generalization of QBF solvers could help to compute widths, as needed for our RCI scheme, to our knowledge such counting/minimization generalizations of QBF have never been studied. This is a clear area for further theoretical and algorithmic work.

On the practical side, there are many further potential applications of algorithmic improvisation. In Chapters 6–8 we mentioned several: planning for search and rescue missions by robot swarms, modeling the dynamic behaviors of human drivers, and testing entire closed-loop autonomous driving agents. There are also good prospects for applications in completely different fields, including education and cybersecurity:

Automated Exercise Generation. A natural application of algorithmic improvisation is automatically generating individualized homework and exam problems for large courses such as MOOCs to prevent the transfer of answers from one student to another [158, 151]. Given a reference problem written by the instructor, we can generate variations on it with algorithmic improvisation, using the soft constraint to enforce similarity, the hard constraint to preserve any important features which should remain invariant (e.g. consistency between different parts of the problem), and the randomness constraint to ensure a diversity of problems.

Software Diversity. As we mentioned in Chapter 1, there is a body of work in cybersecurity on heuristics for introducing diversity into the implementation of a program to make developing exploits more difficult [105]. Algorithmic improvisation could be used to do this in a more principled way, where we use the hard constraint to enforce functional equivalence to the original implementation and a quantitative soft constraint to ensure, for example, that runtime or some other performance metric is not adversely affected to an unacceptable degree. Turning this application around, we could also use algorithmic improvisation to test or train an antivirus program by generating many variants on existing malware samples subject to hard constraints enforcing that the variants still perform a damaging or undesirable operation.

Crypto-Free Privacy Schemes. Finally, in settings like wireless sensor networks or RFID tags, there is significant interest in security protocols for authentication, privacy, etc.

which avoid using traditional cryptographic primitives, since the devices do not have sufficient resources to execute such operations [95, 104]. Here, algorithmic improvisation could be used to perform simple types of obfuscation, generating variations on secret data which preserve some required structure (possibly along the lines of Wu et al. [190]).

As these directions make abundantly clear, we are truly at the beginning of exploring the theory and applications of algorithmic improvisation. Our hope is that, beginning from this first step into the correct-by-construction synthesis of randomized systems, we will continue to find a wealth of results and algorithms that will enable us to make safety-critical systems ever more safe, secure, and dependable.

Bibliography

[1] Ilge Akkaya. "Data-Driven Cyber-Physical Systems via Real-Time Stream Analytics and Machine Learning". PhD thesis. University of California, Berkeley, Oct. 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-159.html (cit. on p. 141).

[2] Ilge Akkaya, Daniel J. Fremont, Rafael Valle, Alexandre Donzé, Edward A. Lee, and Sanjit A. Seshia. "Control Improvisation with Probabilistic Temporal Specifications". In: *1st IEEE International Conference on Internet-of-Things Design and Implementation (IoTDI)*. Apr. 2016, pp. 187–198. DOI: 10.1109/IoTDI.2015.33 (cit. on pp. 2, 6, 10, 46, 47, 140, 141, 143).

[3] Cyril Allauzen, Maxime Crochemore, and Mathieu Raffinot. "Efficient Experimental String Matching by Weak Factor Recognition". In: *12th Annual Symposium on Combinatorial Pattern Matching (CPM)*. July 2001, pp. 51–72. DOI: 10.1007/3-540-48194-X_5 (cit. on p. 33).

[4] Glenn Ammons, Rastislav Bodík, and James R. Larus. "Mining specifications". In: *29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Jan. 2002, pp. 4–16. DOI: 10.1145/503272.503275 (cit. on p. 125).

[5] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul F. Christiano, John Schulman, and Dan Mané. *Concrete Problems in AI Safety*. 2016. arXiv: 1606.06565 (cit. on p. 161).

[6] Marcelo Arenas, Luis Alberto Croquevielle, Rajesh Jayaram, and Cristian Riveros. "Efficient Logspace Classes for Enumeration, Counting, and Uniform Generation". In: *38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS)*. July 2019, pp. 59–73. DOI: 10.1145/3294052.3319704 (cit. on p. 19).

[7] Sylvain Arlot and Alain Celisse. "A survey of cross-validation procedures for model selection". In: *Statist. Surv.* 4 (2010), pp. 40–79. DOI: 10.1214/09-SS054 (cit. on p. 172).

[8] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. New York: Cambridge University Press, 2009. ISBN: 978-0-521-42426-4 (cit. on pp. 13, 18).

[9] Gérard Assayag and Shlomo Dubnov. "Using Factor Oracles for Machine Improvisation". In: *Soft Computing* 8.9 (2004), pp. 604–610. DOI: 10.1007/s00500-004-0385-4 (cit. on pp. 4, 5, 33).

[10] Gérard Assayag, Shlomo Dubnov, Marc Chemillier, Georges Bloch, and Benjamin Lévy. *OMax*. 2018. URL: http://repmus.ircam.fr/omax/home (cit. on p. 33).

[11] Christel Baier, Tomáš Brázdil, Marcus Größer, and Antonín Kučera. "Stochastic game logic". In: *Acta Informatica* 49.4 (2012), pp. 203–224. DOI: 10.1007/s00236-012-0156-0 (cit. on p. 55).

[12] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008. ISBN: 978-0-262-02649-9 (cit. on pp. 141, 144).

[13] Yehoshua Bar-Hillel, Micha Perles, and Eliyahu Shamir. "On Formal Properties of Simple Phrase Structure Grammars". In: *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung* 14 (1961), pp. 143–172 (cit. on pp. 37–39, 41, 42). Reprinted in Yehoshua Bar-Hillel. *Language and Information*. Addison-Wesley, 1964.

[14] Ezio Bartocci, Radu Grosu, Panagiotis Katsaros, C. R. Ramakrishnan, and Scott A. Smolka. "Model Repair for Probabilistic Systems". In: *17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Mar. 2011, pp. 326–340. DOI: 10.1007/978-3-642-19835-9_30 (cit. on p. 159).

[15] Mihir Bellare, Oded Goldreich, and Erez Petrank. "Uniform Generation of NP-Witnesses Using an NP-Oracle". In: *Information and Computation* 163.2 (2000), pp. 510–526. DOI: 10.1006/inco.2000.2885 (cit. on pp. 4, 5, 19, 32, 45).

[16] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. "Symbolic Model Checking without BDDs". In: *5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*. Mar. 1999, pp. 193–207. DOI: 10.1007/3-540-49059-0_14 (cit. on pp. 43, 44).

[17] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, eds. *Handbook of Satisfiability*. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009. ISBN: 978-1-58603-929-5 (cit. on pp. 17, 18).

[18] Roderick Bloem, Stefan Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. "Specify, Compile, Run: Hardware from PSL". In: *6th International Workshop on Compiler Optimization Meets Compiler Verification (COCV)*. Mar. 2007, pp. 3–16. DOI: 10.1016/j.entcs.2007.09.004 (cit. on p. 55).

[19] Johannes Borgström, Andrew D. Gordon, Michael Greenberg, James Margetson, and Jurgen Van Gael. "Measure Transformer Semantics for Bayesian Machine Learning". In: *Logical Methods in Computer Science* 9.3 (2013). DOI: 10.2168/LMCS-9(3:11)2013 (cit. on p. 121).

[20] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005. ISBN: 3540262784 (cit. on p. 163).

[21] Hans Kleine Büning and Uwe Bubeck. "Theory of Quantified Boolean Formulas". In: *Handbook of Satisfiability*. Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009. Chap. 23, pp. 735–760. ISBN: 978-1-58603-929-5 (cit. on p. 18).

[22] Ondrej Burkacky, Johannes Deichmann, Georg Doll, and Christian Knochenhauer. *Rethinking car software and electronics architecture*. McKinsey & Company, Feb. 2018. URL: https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/rethinking-car-software-and-electronics-architecture (cit. on p. 1).

[23] Bob Carpenter, Andrew Gelman, Matthew Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. "Stan: A Probabilistic Programming Language". In: *Journal of Statistical Software* 76.1 (2017), pp. 1–32. DOI: 10.18637/jss.v076.i01 (cit. on p. 93).

[24] João Cartucho. *mean Average Precision*. https://github.com/Cartucho/mAP. 2019 (cit. on p. 170).

[25] Christos G. Cassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006. ISBN: 0387333320 (cit. on p. 6).

[26] Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. "Distribution-Aware Sampling and Weighted Model Counting for SAT". In: *28th AAAI Conference on Artificial Intelligence*. July 2014, pp. 1722–1730. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8364 (cit. on p. 19).

[27] Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. "On Parallel Scalable Uniform SAT Witness Generation". In: *21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Apr. 2015, pp. 304–319. DOI: 10.1007/978-3-662-46681-0_25 (cit. on p. 20).

[28] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. "Algorithmic Improvements in Approximate Counting for Probabilistic Inference: From Linear to Logarithmic SAT Calls". In: *25th International Joint Conference on Artificial Intelligence (IJCAI)*. July 2016, pp. 3569–3576. URL: http://www.ijcai.org/Abstract/16/503 (cit. on p. 19).

[29] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. "Balancing Scalability and Uniformity in SAT Witness Generator". In: *51st Design Automation Conference (DAC)*. June 2014, 60:1–60:6. DOI: 10.1145/2593069.2593097 (cit. on pp. 4, 5, 19, 45).

[30] Abraham Chan. "Automated Program Diversity Using Program Synthesis". In: *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) Workshops*. June 2017, pp. 156–159. DOI: 10.1109/DSN-W.2017.30 (cit. on p. 5).

[31] Taolue Chen, Vojtěch Forejt, Marta Kwiatkowska, Aistis Simaitis, and Clemens Wiltsche. "On Stochastic Games with Multiple Objectives". In: *38th International Symposium on Mathematical Foundations of Computer Science (MFCS)*. Aug. 2013, pp. 266–277. DOI: 10.1007/978-3-642-40313-2_25 (cit. on p. 55).

[32] Silvia Chiappa. "Explicit-Duration Markov Switching Models". In: *Foundations and Trends in Machine Learning* 7.6 (2014), pp. 803–886. DOI: 10.1561/2200000054 (cit. on p. 142).

[33] Alonzo Church. "Application of recursive arithmetic to the problem of circuit synthesis". In: *Summaries of talks presented at the Summer Institute for Symbolic Logic, Cornell University, 1957*. Princeton, New Jersey, USA: Communications Research Division, Institute for Defense Analyses, 1960, 3–50, 3a–45a (cit. on p. 55).

[34] Koen Claessen and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Sept. 2000, pp. 268–279. DOI: 10.1145/351240.351266 (cit. on p. 83).

[35] Guillaume Claret, Sriram K. Rajamani, Aditya V. Nori, Andrew D. Gordon, and Johannes Borgström. "Bayesian Inference Using Data Flow Analysis". In: *9th Joint Meeting on Foundations of Software Engineering (FSE)*. 2013, pp. 92–102. DOI: 10.1145/2491411.2491423 (cit. on pp. 110, 115, 120).

[36] Edmund M. Clarke and Jeannette M. Wing. "Formal Methods: State of the Art and Future Directions". In: *ACM Computing Surveys* 28.4 (1996), pp. 626–643. DOI: 10.1145/242223.242257 (cit. on p. 1).

[37] Loek G. Cleophas, Gerard Zwaan, and Bruce W. Watson. "Constructing Factor Oracles". In: *Prague Stringology Conference 2003*. Sept. 2003, pp. 37–50. URL: http://www.stringology.org/event/2003/p4.html (cit. on p. 33).

[38] Anne Condon and Richard J. Lipton. "On the Complexity of Space Bounded Interactive Proofs". In: *30th Annual Symposium on Foundations of Computer Science (FOCS)*. Oct. 1989, pp. 462–467. DOI: 10.1109/SFCS.1989.63519 (cit. on p. 36).

[39] Robert L. Constable, Harry B. Hunt III, and Sartaj Sahni. *On the computational complexity of scheme equivalence*. Tech. rep. TR 74-201. Cornell University, Mar. 1974. URL: http://hdl.handle.net/1813/6041 (cit. on p. 42).

[40] Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. "Gen: a general-purpose probabilistic programming system with programmable inference". In: *40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. June 2019, pp. 221–236. DOI: 10.1145/3314221.3314642 (cit. on pp. 121, 124).

[41] Giuseppe De Giacomo and Moshe Y. Vardi. "Linear Temporal Logic and Linear Dynamic Logic on Finite Traces". In: *23rd International Joint Conference on Artificial Intelligence*. IJCAI '13. Beijing, China: AAAI Press, 2013, pp. 854–860. ISBN: 978-1-57735-633-2. URL: http://dl.acm.org/citation.cfm?id=2540128.2540252 (cit. on p. 79).

[42] Ankush Desai, Indranil Saha, Jianqiao Yang, Shaz Qadeer, and Sanjit A. Seshia. "DRONA: A Framework for Safe Distributed Mobile Robotics". In: *8th International Conference on Cyber-Physical Systems (ICCPS)*. Apr. 2017, pp. 239–248. DOI: 10.1145/3055004.3055022 (cit. on p. 131).

[43] Alexandre Donzé, Sophie Libkind, Sanjit A. Seshia, and David Wessel. *Control Improvisation with Application to Music*. Tech. rep. UCB/EECS-2013-183. EECS Department, University of California, Berkeley, Nov. 2013. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-183.html (cit. on pp. 6, 7, 23, 36).

[44] Alexandre Donzé, Rafael Valle, Ilge Akkaya, Sophie Libkind, Sanjit A. Seshia, and David Wessel. "Machine Improvisation with Formal Specifications". In: *40th International Computer Music Conference (ICMC)*. Sept. 2014, pp. 1277–1284. URL: http://hdl.handle.net/2027/spo.bbp2372.2014.196 (cit. on pp. 2, 6, 7, 22, 23, 33).

[45] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio M. López, and Vladlen Koltun. "CARLA: An Open Urban Driving Simulator". In: *1st Annual Conference on Robot Learning (CoRL)*. Nov. 2017, pp. 1–16. URL: http://proceedings.mlr.press/v78/dosovitskiy17a.html (cit. on pp. 87, 165).

[46] Tommaso Dreossi, Alexandre Donzé, and Sanjit A. Seshia. "Compositional Falsification of Cyber-Physical Systems with Machine Learning Components". In: *9th NASA Formal Methods Symposium (NFM)*. May 2017, pp. 357–372. DOI: 10.1007/978-3-319-57288-8_26 (cit. on p. 163).

[47] Tommaso Dreossi, Daniel J. Fremont, Shromona Ghosh, Edward Kim, Hadi Ravanbakhsh, Marcell Vazquez-Chanlatte, and Sanjit A. Seshia. "VerifAI: A Toolkit for the Formal Design and Analysis of Artificial Intelligence-Based Systems". In: *31st International Conference on Computer Aided Verification (CAV)*. July 2019. DOI: 10.1007/978-3-030-25540-4_25. URL: https://github.com/BerkeleyLearnVerify/VerifAI (cit. on pp. 11, 85, 165, 178).

[48] Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Kurt Keutzer, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia. "Counterexample-Guided Data Augmentation". In: *27th International Joint Conference on Artificial Intelligence (IJCAI)*. July 2018, pp. 2071–2078. DOI: 10.24963/ijcai.2018/286 (cit. on p. 163).

[49] Tommaso Dreossi, Somesh Jha, and Sanjit A. Seshia. "Semantic Adversarial Deep Learning". In: *30th International Conference on Computer Aided Verification (CAV)*. July 2018, pp. 3–26. DOI: 10.1007/978-3-319-96145-3_1 (cit. on p. 163).

[50] DuPont. *Global Automotive Color Popularity Report*. 2012. URL: https://web.archive.org/web/20130818022236/http://www2.dupont.com/Media_Center/en_US/color_popularity/Images_2012/DuPont2012ColorPopularity.pdf (cit. on p. 169).

[51] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing & Modeling: A Procedural Approach*. 3rd edition. San Francisco: Morgan Kaufmann, 2003. ISBN: 1-55860-848-6 (cit. on p. 163).

[52] Tayfun Elmas, Jacob Burnim, George Necula, and Koushik Sen. "CONCURRIT: A Domain Specific Language for Reproducing Concurrency Bugs". In: *34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. June 2013, pp. 153–164. DOI: 10.1145/2491956.2462162 (cit. on pp. 83, 164).

[53] Mark Everingham and John Winn. *The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Development Kit*. 2012. URL: http://host.robots.ox.ac.uk/pascal/VOC/voc2012/htmldoc/index.html (cit. on p. 170).

[54] Jerzy Filar and Koos Vrieze. *Competitive Markov Decision Processes*. New York, NY: Springer, 1997. ISBN: 978-1-4612-8481-9. DOI: 10.1007/978-1-4612-4054-9 (cit. on p. 67).

[55] Artur Filipowicz, Jeremiah Liu, and Alain Kornhauser. "Learning to recognize distance to stop signs using the virtual world of Grand Theft Auto 5". In: *Transportation Research Board 96th Annual Meeting*. 17-05456. Jan. 2017. URL: https://orfe.princeton.edu/~alaink/TRB'17/Using_GTAV_to_Learn_Distances_TRB_Final.pdf (cit. on pp. 163, 168).

[56] Bernd Finkbeiner. "Synthesis of Reactive Systems". In: *Dependable Software Systems Engineering*. Ed. by Javier Esparza, Orna Grumberg, and Salomon Sickert. Vol. 45. NATO Science for Peace and Security Series, D: Information and Communication Security. Amsterdam, Netherlands: IOS Press, 2016, pp. 72–98. ISBN: 978-1-61499-626-2 (cit. on pp. 1, 5, 55, 57).

[57] Matthew Fisher, Daniel Ritchie, Manolis Savva, Thomas Funkhouser, and Pat Hanrahan. "Example-based synthesis of 3D object arrangements". In: *ACM Transactions on Graphics* 31.6 (2012), 135:1–135:11. DOI: 10.1145/2366145.2366154 (cit. on p. 163).

[58] Robert W. Floyd. "On ambiguity in phrase structure languages". In: *Communications of the ACM* 5.10 (1962), p. 526. DOI: 10.1145/368959.368993 (cit. on pp. 41, 42).

[59] OpenStreetMap Foundation. *OpenStreetMap*. 2019. URL: https://www.openstreetmap.org/ (cit. on pp. 87, 165).

[60] Python Software Foundation. *Python 3.7 documentation*. 2019. URL: https://docs.python.org/3.7/index.html (cit. on p. 95).

[61] Robert J. Fowler, Michael S. Paterson, and Steven L. Tanimoto. "Optimal packing and covering in the plane are NP-complete". In: *Information Processing Letters* 12.3 (1981), pp. 133–137. DOI: 10.1016/0020-0190(81)90111-3 (cit. on p. 125).

[62] Daniel J. Fremont, Alexandre Donzé, and Sanjit A. Seshia. *Control Improvisation*. 2017. arXiv: 1704.06319 (cit. on pp. 5–7, 22, 23, 46).

[63] Daniel J. Fremont, Alexandre Donzé, Sanjit A. Seshia, and David Wessel. "Control Improvisation". In: *35th IARCS Annual Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS)*. Dec. 2015, pp. 463–474. DOI: 10.4230/LIPIcs.FSTTCS.2015.463 (cit. on pp. 5–7, 10, 22, 23, 29, 36).

[64] Daniel J. Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia. "Scenic: A Language for Scenario Specification and Scene Generation". In: *40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. June 2019, pp. 63–78. DOI: 10.1145/3314221.3314633 (cit. on pp. 2, 6, 9, 11, 84, 160).

[65] Daniel J. Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia. *Scenic: A Language for Scenario Specification and Scene Generation*. 2019. arXiv: 1809.09310. URL: https://github.com/BerkeleyLearnVerify/Scenic (cit. on pp. 9, 84, 95, 97, 124).

[66] Daniel J. Fremont, Markus N. Rabe, and Sanjit A. Seshia. "Maximum Model Counting". In: *31st AAAI Conference on Artificial Intelligence*. Feb. 2017, pp. 3885–3892. URL: http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14968 (cit. on p. 20).

[67] Daniel J. Fremont and Sanjit A. Seshia. "Reactive Control Improvisation". In: *30th International Conference on Computer Aided Verification (CAV)*. July 2018, pp. 307–326. DOI: 10.1007/978-3-319-96145-3_17 (cit. on pp. 5, 6, 8, 10, 55, 133).

[68] Rockstar Games. *Grand Theft Auto V*. Windows PC version. 2015. URL: https://www.rockstargames.com/games/info/V (cit. on pp. 85, 164, 168).

[69] Jin I. Ge and Richard M. Murray. "Voluntary lane-change policy synthesis with control improvisation". In: *57th IEEE Conference on Decision and Control (CDC)*. Dec. 2018, pp. 3640–3647. DOI: 10.1109/CDC.2018.8619616 (cit. on pp. 140, 159).

[70] Andreas Geiger, Philip Lenz, and Raquel Urtasun. "Are we ready for autonomous driving? The KITTI vision benchmark suite". In: *2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2012, pp. 3354–3361. DOI: 10.1109/CVPR.2012.6248074 (cit. on p. 170).

[71] Seymour Ginsburg and Joseph S. Ullian. "Ambiguity in context free languages". In: *Journal of the ACM (JACM)* 13.1 (1966), pp. 62–89. DOI: 10.1145/321312.321318 (cit. on p. 37).

[72] James Gleick. "Little Bug, Big Bang". In: *The New York Times Magazine* (Dec. 1, 1996). URL: https://nyti.ms/2nAlfve (cit. on p. 1).

[73] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Reading, Massachusetts: Addison-Wesley, 1983. ISBN: 0-201-11371-6 (cit. on pp. 88, 96).

[74] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. "Model Counting". In: *Handbook of Satisfiability*. Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009. Chap. 20, pp. 633–654. ISBN: 978-1-58603-929-5 (cit. on p. 19).

[75] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative Adversarial Nets". In: *Advances in Neural Information Processing Systems (NIPS) 27*. Dec. 2014, pp. 2672–2680. URL: https://papers.nips.cc/paper/5423-generative-adversarial-nets (cit. on p. 163).

[76] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. "Explaining and Harnessing Adversarial Examples". In: *3rd International Conference on Learning Representations (ICLR)*. May 2015. arXiv: 1412.6572 (cit. on p. 163).

[77] Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. "Church: a language for generative models". In: *24th Conference in Uncertainty in Artificial Intelligence (UAI)*. July 2008, pp. 220–229. URL: https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=1346&proceeding_id=24 (cit. on pp. 84, 164).

[78] Noah D. Goodman and Andreas Stuhlmüller. *The Design and Implementation of Probabilistic Programming Languages*. http://dippl.org. Accessed: 2018-7-11. 2014 (cit. on p. 164).

[79] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. "Probabilistic Programming". In: *Future of Software Engineering (FOSE)*. June 2014, pp. 167–181. DOI: 10.1145/2593882.2593900 (cit. on pp. 5, 84, 92, 108, 164).

[80] Vivek Gore, Mark R. Jerrum, Sampath Kannan, Z. Sweedyk, and Steve Mahaney. "A Quasi-Polynomial-Time Algorithm for Sampling Words from a Context-Free Language". In: *Information and Computation* 134.1 (1997), pp. 59–74. DOI: 10.1006/inco.1997.2621 (cit. on p. 19).

[81] Jeremy Grace and John Baillieul. "Stochastic Strategies for Autonomous Robotic Surveillance". In: *44th IEEE Conference on Decision and Control (CDC)*. Dec. 2005, pp. 2200–2205. DOI: 10.1109/CDC.2005.1582488 (cit. on pp. 2, 127).

[82] Offer Grembek, Alex Kurzhanskiy, Aditya Medury, Pravin Varaiya, and Mengqiao Yu. "Making intersections safer with I2V communication". In: *Transportation Research Part C: Emerging Technologies* 102 (2019), pp. 396–410. DOI: 10.1016/j.trc.2019.02.017. URL: https://github.com/ucbtrans/intelligent_intersection (cit. on p. 165).

[83] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. "Program Synthesis". In: *Foundations and Trends in Programming Languages* 4.1-2 (2017), pp. 1–119. DOI: 10.1561/2500000010 (cit. on pp. 1, 5).

[84] Ankush Gupta, Andrea Vedaldi, and Andrew Zisserman. "Synthetic Data for Text Localisation in Natural Images". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016, pp. 2315–2324. DOI: 10.1109/CVPR.2016.254 (cit. on pp. 161, 163).

[85] Hans Hansson and Bengt Jonsson. "A Logic for Reasoning about Time and Reliability". In: *Formal Aspects of Computing* 6.5 (1994), pp. 512–535. DOI: 10.1007/BF01211866 (cit. on p. 55).

[86] Timothy Hickey and Jacques Cohen. "Uniform Random Generation of Strings in a Context-Free Language". In: *SIAM Journal on Computing* 12.4 (1983), pp. 645–655. DOI: 10.1137/0212044 (cit. on pp. 4, 5, 18, 19, 35–37, 40, 67, 77, 132).

[87] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 2nd edition. Addison-Wesley, 2001. ISBN: 978-0-201-44124-6 (cit. on pp. 15–17, 37, 38, 40, 42).

[88] Max Jaderberg, Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. *Synthetic Data and Artificial Neural Networks for Natural Scene Text Recognition*. 2014. arXiv: 1406.2227 (cit. on p. 163).

[89] Mark R. Jerrum, Leslie G. Valiant, and Vijay V. Vazirani. "Random Generation of Combinatorial Structures from a Uniform Distribution". In: *Theoretical Computer Science* 43 (1986), pp. 169–188. DOI: 10.1016/0304-3975(86)90174-X (cit. on pp. 4, 5, 18, 19, 32).

[90] Chenfanfu Jiang, Siyuan Qi, Yixin Zhu, Siyuan Huang, Jenny Lin, Lap-Fai Yu, Demetri Terzopoulos, and Song-Chun Zhu. "Configurable 3D Scene Synthesis and 2D Image Rendering with Per-pixel Ground Truth Using Stochastic Grammars". In: *International Journal of Computer Vision* 126.9 (2018), pp. 920–941. DOI: 10.1007/s11263-018-1103-5 (cit. on p. 164).

[91] Matthew Johnson-Roberson, Charles Barto, Rounak Mehta, Sharath Nittur Sridhar, Karl Rosaen, and Ram Vasudevan. "Driving in the Matrix: Can virtual worlds replace human-generated annotations for real world tasks?" In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. May 2017, pp. 746–753. DOI: 10.1109/ICRA.2017.7989092 (cit. on pp. 161, 163–165, 167, 168, 170, 172, 174, 178).

[92] Alexander Jung. *imgaug*. 2018. URL: https://github.com/aleju/imgaug (cit. on p. 177).

[93] Sampath Kannan, Z. Sweedyk, and Steve Mahaney. "Counting and Random Generation of Strings in Regular Languages". In: *6th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Jan. 1995, pp. 551–557. URL: https://dl.acm.org/citation.cfm?id=313651.313803 (cit. on pp. 4, 5, 19, 36, 52, 78).

[94] Narendra Karmarkar. "A new polynomial-time algorithm for linear programming". In: *Combinatorica* 4.4 (1984), pp. 373–396. DOI: 10.1007/BF02579150 (cit. on p. 50).

[95] Sindhu Karthikeyan and Mikhail Nesterenko. "RFID Security without Extensive Cryptography". In: *3rd ACM Workshop on Security of Ad Hoc and Sensor Networks (SASN)*. Nov. 2005, pp. 63–67. DOI: 10.1145/1102219.1102229 (cit. on p. 181).

[96] Jack Kelly and William Knottenbelt. "The UK-DALE dataset, domestic appliance-level electricity demand and whole-house demand from five UK homes". In: *Scientific Data* 2.150007 (2015). DOI: 10.1038/sdata.2015.7 (cit. on pp. 152, 153).

[97] Leonid G. Khachiyan. "A Polynomial Algorithm in Linear Programming". In: *Doklady Akademii Nauk SSSR* 244.5 (1979), pp. 1093–1096. URL: http://mi.mathnet.ru/eng/dan42319 (cit. on p. 50). Translated as "A Polynomial Algorithm in Linear Programming". In: *Soviet Mathematics Doklady* 20.1 (1979), pp. 191–194.

[98] Dania El-Khechen, Muriel Dulieu, John Iacono, and Nikolaj van Omme. "Packing 2×2 unit squares into grid polygons is NP-complete". In: *21st Annual Canadian Conference on Computational Geometry (CCCG)*. Aug. 2009, pp. 33–36. URL: http://cccg.ca/proceedings/2009/cccg09_09.pdf (cit. on p. 125).

[99] Nathan Koenig and Andrew Howard. "Design and Use Paradigms for Gazebo, an Open-Source Multi-Robot Simulator". In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Sept. 2004, pp. 2149–2154. DOI: 10.1109/IROS.2004.1389727. URL: http://gazebosim.org/ (cit. on pp. 86, 136).

[100] Dexter Kozen. "Lower Bounds for Natural Proof Systems". In: *18th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. Oct. 1977, pp. 254–266. DOI: 10.1109/SFCS.1977.16 (cit. on p. 51).

[101] Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. "Temporal-Logic-Based Reactive Mission and Motion Planning". In: *IEEE Transactions on Robotics* 25.6 (2009), pp. 1370–1381. DOI: 10.1109/TRO.2009.2030225 (cit. on pp. 55, 127, 139).

[102] Tejas D. Kulkarni, Pushmeet Kohli, Joshua B. Tenenbaum, and Vikash K. Mansinghka. "Picture: A probabilistic programming language for scene perception". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2015, pp. 4390–4399. DOI: 10.1109/CVPR.2015.7299068 (cit. on p. 164).

[103] Marta Kwiatkowska, Gethin Norman, and David Parker. "PRISM 4.0: Verification of Probabilistic Real-Time Systems". In: *23rd International Conference on Computer Aided Verification (CAV)*. July 2011, pp. 585–591. DOI: 10.1007/978-3-642-22110-1_47. URL: https://www.prismmodelchecker.org/ (cit. on pp. 145, 148, 153).

[104] Marc Langheinrich. "A survey of RFID privacy approaches". In: *Personal and Ubiquitous Computing* 13.6 (2009), pp. 413–421. DOI: 10.1007/s00779-008-0213-4 (cit. on p. 181).

[105] Per Larsen, Stefan Brunthaler, Lucas Davi, Ahmad-Reza Sadeghi, and Michael Franz. *Automated Software Diversity*. Synthesis Lectures on Information Security, Privacy, & Trust. Morgan & Claypool Publishers, 2015. DOI: 10.2200/S00686ED1V01Y201512SPT014 (cit. on pp. 2, 180).

[106] Edward A. Lee and Alberto L. Sangiovanni-Vincentelli. "A framework for comparing models of computation". In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 17.12 (1998), pp. 1217–1229. DOI: 10.1109/43.736561 (cit. on p. 142).

[107] Wenchao Li, Alessandro Forin, and Sanjit A. Seshia. "Scalable specification mining for verification and diagnosis". In: *47th Design Automation Conference (DAC)*. July 2010, pp. 755–760. DOI: 10.1145/1837274.1837466 (cit. on p. 125).

[108] Xiaodan Liang, Zhiting Hu, Hao Zhang, Chuang Gan, and Eric P. Xing. *Recurrent Topic-Transition GAN for Visual Paragraph Generation*. 2017. arXiv: 1703.07022 (cit. on p. 163).

[109] Joerg Liebelt and Cordelia Schmid. "Multi-view object class detection with a 3D geometric model". In: *Computer Vision and Pattern Recognition (CVPR)*. 2010, pp. 1688–1695. DOI: 10.1109/CVPR.2010.5539836 (cit. on p. 163).

[110] Jacques-Louis Lions, Lennart Lübeck, Jean-Luc Fauquembergue, Gilles Kahn, Wolfgang Kubbat, Stefan Levedag, Leonardo Mazzini, Didier Merle, and Colin O'Halloran. *ARIANE 5 Flight 501 Failure*. Ariane 501 Inquiry Board report. July 19, 1996. URL: https://esamultimedia.esa.int/docs/esa-x-1819eng.pdf (cit. on p. 1).

[111] Changliu Liu, Tomer Arnon, Christopher Lazarus, Clark W. Barrett, and Mykel J. Kochenderfer. *Algorithms for Verifying Deep Neural Networks*. 2019. arXiv: 1903.06758 (cit. on p. 2).

[112] Fredrik Lundh and Python Software Foundation contributors. *ElementTree*. 2019. URL: https://docs.python.org/3.7/library/xml.etree.elementtree.html (cit. on p. 87).

[113] Gilmore R. Lundquist, Vishwath Mohan, and Kevin W. Hamlen. "Searching for software diversity: attaining artificial diversity through program synthesis". In: *2016 New Security Paradigms Workshop (NSPW)*. Sept. 2016, pp. 80–91. DOI: 10.1145/3011883.3011891 (cit. on p. 5).

[114] Alexis C. Madrigal. "Inside Waymo's Secret World for Training Self-Driving Cars". In: *The Atlantic* (Aug. 23, 2017). URL: https://www.theatlantic.com/technology/archive/2017/08/inside-waymos-secret-testing-and-simulation-facilities/537648/ (cit. on pp. 2, 93).

[115] Marco Marchesi. *Megapixel Size Image Creation using Generative Adversarial Networks*. 2017. arXiv: 1706.00082 (cit. on p. 163).

[116] Henry Massalin. "Superoptimizer - A Look at the Smallest Program". In: *2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Oct. 1987, pp. 122–126. DOI: 10.1145/36177.36194 (cit. on p. 83).

[117] René Mazala. "Infinite Games". In: *Automata, Logics, and Infinite Games*. Ed. by Erich Grädel, Wolfgang Thomas, and Thomas Wilke. Berlin, Heidelberg: Springer, 2002. Chap. 2, pp. 23–38. ISBN: 978-3-540-36387-3. DOI: 10.1007/3-540-36387-4_2 (cit. on pp. 57, 67).

[118] Bruce McKenzie. *Generating Strings at Random from a Context Free Grammar*. Technical Report TR-COSC 10/97. University of Canterbury, 1997. URL: http://hdl.handle.net/10092/11231 (cit. on pp. 19, 37, 40, 67).

[119] Kuldeep S. Meel. "Constrained Counting and Sampling: Bridging the Gap Between Theory and Practice". PhD thesis. Rice University, 2017. URL: http://hdl.handle.net/1911/105480 (cit. on p. 19).

[120] Kuldeep S. Meel, Moshe Y. Vardi, Supratik Chakraborty, Daniel J. Fremont, Sanjit A. Seshia, Dror Fried, Alexander Ivrii, and Sharad Malik. "Constrained Sampling and Counting: Universal Hashing Meets SAT Solving". In: *2016 AAAI Workshop on Beyond NP*. Feb. 2016. URL: http://www.aaai.org/ocs/index.php/WS/AAAIW16/paper/view/12618 (cit. on p. 19).

[121] Lorenz Meier, Dominik Honegger, and Marc Pollefeys. "PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms". In: *IEEE International Conference on Robotics and Automation (ICRA) 2015*. May 2015, pp. 6235–6240. DOI: 10.1109/ICRA.2015.7140074. URL: https://px4.io/ (cit. on p. 136).

[122] Olivier Michel. "Webots: Professional Mobile Robot Simulation". In: *International Journal of Advanced Robotic Systems* 1.1 (2004), pp. 39–42. URL: https://cyberbotics.com/ (cit. on pp. 86, 87, 95, 165).

[123] Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. "BLOG: Probabilistic Models with Unknown Objects". In: *19th International Joint Conference on Artificial Intelligence (IJCAI)*. Aug. 2005, pp. 1352–1359. URL: http://ijcai.org/Proceedings/05/Papers/1546.pdf (cit. on pp. 84, 108, 121, 124, 164).

[124] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. "DeepFool: A Simple and Accurate Method to Fool Deep Neural Networks". In: *Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 2574–2582. DOI: 10.1109/CVPR.2016.282 (cit. on p. 163).

[125] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc J. Van Gool. "Procedural modeling of buildings". In: *ACM Trans. Graph.* 25.3 (2006), pp. 614–623. DOI: 10.1145/1141911.1141931 (cit. on pp. 92, 164).

[126] Masakazu Nasu and Namio Honda. "Mappings Induced by PGSM-Mappings and Some Recursively Unsolvable Problems of Finite Probabilistic Automata". In: *Information and Control* 15.3 (1969), pp. 250–273. DOI: 10.1016/S0019-9958(69)90449-5 (cit. on p. 36).

[127] Yehuda Naveh, Michal Rimon, Itai Jaeger, Yoav Katz, Michael Vinov, Eitan Marcus, and Gil Shurek. "Constraint-Based Random Stimuli Generation for Hardware Verification". In: *21st AAAI Conference on Artificial Intelligence*. July 2006, pp. 1720–1727. URL: http://www.aaai.org/Library/AAAI/2006/aaai06-287.php (cit. on p. 164).

[128] Truong Nghiem, Sriram Sankaranarayanan, Georgios E. Fainekos, Franjo Ivančić, Aarti Gupta, and George J. Pappas. "Monte-Carlo Techniques for Falsification of Temporal Properties of Non-Linear Hybrid Systems". In: *13th ACM International Conference on Hybrid Systems: Computation and Control (HSCC)*. Apr. 2010, pp. 211–220. DOI: 10.1145/1755952.1755983 (cit. on p. 178).

[129] Anh Mai Nguyen, Jason Yosinski, and Jeff Clune. "Deep neural networks are easily fooled: High confidence predictions for unrecognizable images". In: *Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 427–436. DOI: 10.1109/CVPR.2015.7298640 (cit. on p. 163).

[130] Aditya V. Nori, Chung-Kil Hur, Sriram K. Rajamani, and Selva Samuel. "R2: An Efficient MCMC Sampler for Probabilistic Programs". In: *28th AAAI Conference on Artificial Intelligence*. July 2014, pp. 2476–2482. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8192 (cit. on pp. 121, 124).

[131] Lynne E. Parker. "Multiple Mobile Robot Systems". In: *Springer Handbook of Robotics*. Ed. by Bruno Siciliano and Oussama Khatib. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 921–941. ISBN: 978-3-540-30301-5. DOI: 10.1007/978-3-540-30301-5_41 (cit. on p. 138).

[132] Terence Parr. *The Definitive ANTLR 4 Reference*. Dallas: Pragmatic Bookshelf, 2014. ISBN: 978-1-93435-699-9. URL: https://www.antlr.org/ (cit. on p. 87).

[133] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. "DeepXplore: Automated Whitebox Testing of Deep Learning Systems". In: *Symposium on Operating Systems Principles (SOSP)*. Oct. 2017, pp. 1–18. DOI: 10.1145/3132747.3132785 (cit. on p. 163).

[134] Avi Pfeffer. "IBAL: A Probabilistic Rational Programming Language". In: *17th International Joint Conference on Artificial Intelligence (IJCAI), Volume I*. Aug. 2001, pp. 733–740. URL: https://www.ijcai.org/Proceedings/01/IJCAI-2001-i.pdf (cit. on pp. 84, 92, 108, 125).

[135] Amir Pnueli. "The Temporal Logic of Programs". In: *18th Annual Symposium on Foundations of Computer Science (FOCS)*. Oct. 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32 (cit. on p. 79).

[136] Amir Pnueli and Roni Rosner. "On the Synthesis of a Reactive Module". In: *16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Jan. 1989, pp. 179–190. DOI: 10.1145/75277.75293 (cit. on pp. 26, 55).

[137] David Portugal and Rui P. Rocha. "A Survey on Multi-robot Patrolling Algorithms". In: *2nd Doctoral Conference on Computing, Electrical and Industrial Systems (DoCEIS)*. Feb. 2011, pp. 139–146. DOI: 10.1007/978-3-642-19170-1_15 (cit. on p. 127).

[138] Emil L. Post. "A variant of a recursively unsolvable problem". In: *Bulletin of the American Mathematical Society* 52.4 (1946), pp. 264–268. DOI: 10.1090/S0002-9904-1946-08555-9 (cit. on p. 41).

[139] Michael O. Rabin. "Probabilistic Automata". In: *Information and Control* 6.3 (1963), pp. 230–245. DOI: 10.1016/S0019-9958(63)90290-0 (cit. on p. 36).

[140] Lawrence R. Rabiner. "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition". In: *Proceedings of the IEEE* 77.2 (Feb. 1989), pp. 257–286. DOI: 10.1109/5.18626 (cit. on pp. 142, 148).

[141] Laminar Research. *X-Plane 11*. 2019. URL: https://www.x-plane.com/ (cit. on pp. 87, 165).

[142] Stephan R. Richter, Vibhav Vineet, Stefan Roth, and Vladlen Koltun. "Playing for Data: Ground Truth from Computer Games". In: *14th European Conference on Computer Vision (ECCV)*. Oct. 2016, pp. 102–118. DOI: 10.1007/978-3-319-46475-6_7 (cit. on p. 168).

[143] Daniel Ritchie. "Probabilistic programming for procedural modeling and design". PhD thesis. Stanford University, 2016. URL: https://purl.stanford.edu/vh730bw6700 (cit. on p. 164).

[144] Daniel Ritchie. "Quicksand: A Lightweight Embedding of Probabilistic Programming for Procedural Modeling and Design". In: *3rd NIPS Workshop on Probabilistic Programming*. 2014. URL: https://dritchie.github.io/pdf/qs.pdf (cit. on p. 164).

[145] German Ros, Vladlen Koltun, Felipe Codevilla, and Antonio M. López. *CARLA Autonomous Driving Challenge*. 2019. URL: https://carlachallenge.org/ (cit. on p. 178).

[146] German Ros, Laura Sellart, Joanna Materzyńska, David Vázquez, and Antonio M. López. "The SYNTHIA Dataset: A Large Collection of Synthetic Images for Semantic Segmentation of Urban Scenes". In: *Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 3234–3243. DOI: 10.1109/CVPR.2016.352 (cit. on p. 163).

[147] Robert Rowe. *Machine Musicianship*. MIT Press, 2001. ISBN: 9780262182065 (cit. on p. 6).

[148] Aitor Ruano. *DeepGTAV*. 2017. URL: https://github.com/aitorzip/DeepGTAV (cit. on p. 169).

[149] Stuart Russell, Daniel Dewey, and Max Tegmark. "Research Priorities for Robust and Beneficial Artificial Intelligence". In: *AI Magazine* 36.4 (2015). DOI: 10.1609/aimag.v36i4.2577 (cit. on pp. 2, 161).

[150] Feras A. Saad, Marco F. Cusumano-Towner, Ulrich Schaechtle, Martin C. Rinard, and Vikash K. Mansinghka. "Bayesian Synthesis of Probabilistic Programs for Automatic Data Modeling". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019), 37:1–37:32. DOI: 10.1145/3290350 (cit. on p. 125).

[151] Dorsa Sadigh, Sanjit A. Seshia, and Mona Gupta. "Automating exercise generation: a step towards meeting the MOOC challenge for embedded systems". In: *2012 Workshop on Embedded and Cyber-Physical Systems Education (WESE)*. Oct. 2012, p. 2. DOI: 10.1145/2530544.2530546 (cit. on p. 180).

[152] Nasser Saheb-Djahromi. "Probabilistic LCF". In: *7th Symposium on Mathematical Foundations of Computer Science (MFCS)*. Sept. 1978, pp. 442–451. DOI: 10.1007/3-540-08921-7_92 (cit. on pp. 109, 110).

[153] Tiago Sak, Jacques Wainer, and Siome Klein Goldenstein. "Probabilistic Multiagent Patrolling". In: *19th Brazilian Symposium on Artificial Intelligence (SBIA)*. Oct. 2008, pp. 124–133. DOI: 10.1007/978-3-540-88190-2_18 (cit. on p. 127).

[154] Eric Schkufza, Rahul Sharma, and Alex Aiken. "Stochastic Superoptimization". In: *18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Mar. 2013, pp. 305–316. DOI: 10.1145/2451116.2451150 (cit. on p. 83).

[155] Sanjit A. Seshia, Dorsa Sadigh, and S. Shankar Sastry. *Towards Verified Artificial Intelligence*. 2016. arXiv: 1606.08514 (cit. on pp. 2, 161).

[156] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando de Freitas. "Taking the Human Out of the Loop: A Review of Bayesian Optimization". In: *Proceedings of the IEEE* 104.1 (Jan. 2016), pp. 148–175. DOI: 10.1109/JPROC.2015.2494218 (cit. on p. 178).

[157] Yasser Shoukry, Pierluigi Nuzzo, Alberto L. Sangiovanni-Vincentelli, Sanjit A. Seshia, George J. Pappas, and Paulo Tabuada. "SMC: Satisfiability Modulo Convex Programming". In: *Proceedings of the IEEE* 106.9 (2018), pp. 1655–1679. DOI: 10.1109/JPROC.2018.2849003 (cit. on p. 131).

[158] Rohit Singh, Sumit Gulwani, and Sriram K. Rajamani. "Automatically Generating Algebra Problems". In: *26th AAAI Conference on Artificial Intelligence*. July 2012. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/view/5133 (cit. on p. 180).

[159] Armando Solar-Lezama. "Program Synthesis By Sketching". PhD thesis. University of California, Berkeley, Dec. 2008. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-177.html (cit. on p. 159).

[160] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. "Combinatorial Sketching for Finite Programs". In: *12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Oct. 2006, pp. 404–415. DOI: 10.1145/1168857.1168907 (cit. on p. 159).

[161] Mate Soos and Kuldeep S. Meel. "BIRD: Engineering an Efficient CNF-XOR SAT Solver and Its Applications to Approximate Model Counting". In: *33rd AAAI Conference on Artificial Intelligence*. Jan. 2019, pp. 1592–1599. URL: https://aaai.org/ojs/index.php/AAAI/article/view/3974 (cit. on p. 19).

[162] Michael Stark, Michael Goesele, and Bernt Schiele. "Back to the Future: Learning Shape Models from 3D CAD Data". In: *British Machine Vision Conference (BMVC)*. 2010, pp. 1–11. DOI: 10.5244/C.24.106 (cit. on p. 163).

[163] Sam Staton, Hongseok Yang, Frank Wood, Chris Heunen, and Ohad Kammar. "Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints". In: *31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. July 2016, pp. 525–534. DOI: 10.1145/2933575.2935313 (cit. on p. 121).

[164] Larry Stockmeyer. "The Complexity of Approximate Counting". In: *15th Annual ACM Symposium on Theory of Computing (STOC)*. Apr. 1983, pp. 118–126. DOI: 10.1145/800061.808740 (cit. on p. 19).

[165] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007. ISBN: 0321446119 (cit. on pp. 4, 37, 78, 93, 163, 164).

[166] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. *Intriguing properties of neural networks*. 2013. arXiv: 1312.6199 (cit. on pp. 163, 178).

[167] Eric Thorn, Shawn Kimmel, and Michelle Chaka. *A Framework for Automated Driving System Testable Cases and Scenarios*. Tech. rep. DOT HS 812 623. National Highway Traffic Safety Administration, Sept. 2018. URL: https://www.nhtsa.gov/sites/nhtsa.dot.gov/files/documents/13882-automateddrivingsystems_092618_v1a_tag.pdf (cit. on p. 125).

[168] Nikolai Tillmann and Jonathan de Halleux. "Pex—White Box Test Generation for .NET". In: *2nd International Conference on Tests and Proofs (TAP)*. Apr. 2008, pp. 134–153. DOI: 10.1007/978-3-540-79124-9_10 (cit. on p. 83).

[169] Nikolai Tillmann and Wolfram Schulte. "Parameterized Unit Tests". In: *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Sept. 2005, pp. 253–262. DOI: 10.1145/1081706.1081749 (cit. on p. 83).

[170] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. "Domain randomization for transferring deep neural networks from simulation to the real world". In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Sept. 2017, pp. 23–30. DOI: 10.1109/IROS.2017.8202133 (cit. on pp. 161, 163).

[171] Seinosuke Toda. "PP is as Hard as the Polynomial-Time Hierarchy". In: *SIAM Journal on Computing* 20.5 (1991), pp. 865–877. DOI: 10.1137/0220053 (cit. on pp. 33, 44).

[172] Claire Tomlin, John Lygeros, and Shankar Sastry. "Computing Controllers for Nonlinear Hybrid Systems". In: *2nd International Workshop on Hybrid Systems: Computation and Control (HSCC)*. Mar. 1999, pp. 238–255. DOI: 10.1007/3-540-48983-5_22 (cit. on p. 57).

[173] Hazem Torfah and Martin Zimmermann. "The Complexity of Counting Models of Linear-time Temporal Logic". In: *Acta Informatica* 55.3 (2018), pp. 191–212. DOI: 10.1007/s00236-016-0284-z (cit. on p. 79).

[174] Leslie G. Valiant. "The Complexity of Computing the Permanent". In: *Theoretical Computer Science* 8.2 (1979), pp. 189–201. DOI: 10.1016/0304-3975(79)90044-6 (cit. on pp. 14, 19).

[175] Leslie G. Valiant and Vijay V. Vazirani. "NP Is as Easy as Detecting Unique Solutions". In: *17th Annual ACM Symposium on Theory of Computing (STOC)*. May 1985, pp. 458–463. DOI: 10.1145/22145.22196 (cit. on p. 19).

[176] Rafael Valle. "Data Hallucination, Falsification and Validation using Generative Models and Formal Methods". PhD thesis. University of California, Berkeley, 2018. URL: http://www.cnmat.berkeley.edu/publications/data-hallucination-falsification-and-validation-using-generative-models-and-formal (cit. on p. 6).

[177] Rafael Valle, Alexandre Donzé, Daniel J. Fremont, Ilge Akkaya, Sanjit A. Seshia, Adrian Freed, and David Wessel. "Specification Mining for Machine Improvisation with Formal Specifications". In: *Computers in Entertainment* 14.3 (2016), 6:1–6:20. DOI: 10.1145/2967504 (cit. on pp. 6, 125).

[178] Rafael Valle, Daniel J. Fremont, Ilge Akkaya, Alexandre Donzé, Adrian Freed, and Sanjit A. Seshia. "Learning and Visualizing Music Specifications Using Pattern Graphs". In: *17th International Society for Music Information Retrieval Conference (ISMIR)*. Aug. 2016, pp. 192–198. URL: https://wp.nyu.edu/ismir2016/wp-content/uploads/sites/2294/2016/07/280_Paper.pdf (cit. on p. 6).

[179] David Vázquez, Antonio M. López, Javier Marín, Daniel Ponsa, and David Gerónimo. "Virtual and Real World Adaptation for Pedestrian Detection". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36.4 (Apr. 2014), pp. 797–809. DOI: 10.1109/TPAMI.2013.163 (cit. on p. 163).

[180] Persistence of Vision Raytracer Pty. Ltd. *POV-Ray Scene Description Language*. 2013. URL: https://www.povray.org/documentation/3.7.0/r3_3.html (cit. on p. 164).

[181] Bill Vlasic and Neal E. Boudette. "Self-Driving Tesla Was Involved in Fatal Crash, U.S. Says". In: *The New York Times* (June 30, 2016). URL: https://nyti.ms/29dJjPp (cit. on p. 1).

[182] Israel A. Wagner, Michael Lindenbaum, and Alfred M. Bruckstein. "MAC Versus PC: Determinism and Randomness as Complementary Approaches to Robotic Exploration of Continuous Unknown Domains". In: *International Journal of Robotics Research* 19.1 (2000), pp. 12–31. DOI: 10.1177/02783640022066716 (cit. on pp. 2, 127).

[183] Daisuke Wakabayashi. "Self-Driving Uber Car Kills Pedestrian in Arizona, Where Robots Roam". In: *The New York Times* (Mar. 19, 2018). URL: https://nyti.ms/2u3QDYx (cit. on p. 1).

[184] Herbert S. Wilf. "A Unified Setting for Sequencing, Ranking, and Selection Algorithms for Combinatorial Objects". In: *Advances in Mathematics* 24.3 (1977), pp. 281–291. DOI: 10.1016/0001-8708(77)90059-7 (cit. on pp. 18, 35, 40, 67).

[185] Jeannette M. Wing. "A specifier's introduction to formal methods". In: *Computer* 23.9 (1990), pp. 8–22. DOI: 10.1109/2.58215 (cit. on p. 1).

[186] Patrick Henry Winston. *Artificial Intelligence*. 2nd edition. Addison-Wesley Series in Computer Science. Addison-Wesley, 1984. ISBN: 0201082594 (cit. on p. 122).

[187] Sebastien C. Wong, Adam Gatt, Victor Stamatescu, and Mark D. McDonnell. "Understanding Data Augmentation for Classification: When to Warp?" In: *International Conference on Digital Image Computing: Techniques and Applications (DICTA)*. Nov. 2016, pp. 1–6. DOI: 10.1109/DICTA.2016.7797091 (cit. on p. 163).

[188] Frank Wood, Jan-Willem van de Meent, and Vikash K. Mansinghka. "A New Approach to Probabilistic Programming Inference". In: *17th International Conference on Artificial Intelligence and Statistics (AISTATS)*. Apr. 2014, pp. 1024–1032. URL: http://proceedings.mlr.press/v33/wood14.html (cit. on pp. 121, 124).

[189] Bichen Wu, Forrest N. Iandola, Peter H. Jin, and Kurt Keutzer. "SqueezeDet: Unified, Small, Low Power Fully Convolutional Neural Networks for Real-Time Object Detection for Autonomous Driving". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. July 2017, pp. 446–454. DOI: 10.1109/CVPRW.2017.60 (cit. on pp. 164, 169).

[190] Yi-Chin Wu, Vasumathi Raman, Blake C. Rawlings, Stéphane Lafortune, and Sanjit A. Seshia. "Synthesis of Obfuscation Policies to Ensure Privacy and Utility". In: *Journal of Automated Reasoning* 60.1 (2018), pp. 107–131. DOI: 10.1007/s10817-017-9420-x (cit. on p. 181).

[191] Weiming Xiang, Patrick Musau, Ayana A. Wild, Diego Manzanas Lopez, Nathaniel Hamilton, Xiaodong Yang, Joel A. Rosenfeld, and Taylor T. Johnson. *Verification for Machine Learning, Autonomy, and Neural Networks Survey*. 2018. arXiv: 1810.01989 (cit. on p. 2).

[192] Yan Xu, Ran Jia, Lili Mou, Ge Li, Yunchuan Chen, Yangyang Lu, and Zhi Jin. "Improved relation classification by deep recurrent neural networks with data augmentation". In: *26th International Conference on Computational Linguistics (COLING)*. Dec. 2016, pp. 1461–1470. URL: https://www.aclweb.org/anthology/C16-1138 (cit. on p. 163).

[193] Shun-Zheng Yu. "Hidden semi-Markov models". In: *Artificial Intelligence* 174.2 (2010), pp. 215–243. DOI: 10.1016/j.artint.2009.11.011 (cit. on pp. 142, 144).

[194] Kim Zetter. "Inside the Cunning, Unprecedented Hack of Ukraine's Power Grid". In: *Wired* (Mar. 3, 2016). URL: https://www.wired.com/2016/03/inside-cunning-unprecedented-hack-ukraines-power-grid/ (cit. on p. 1).