Automated Quantitative Software Verification

by

Mark Kattenbelt

A thesis submitted for the degree of
Doctor of Philosophy

Trinity College, Oxford
Trinity Term 2010

Abstract

Many software systems exhibit probabilistic behaviour, either added explicitly, to improve performance or to break symmetry, or implicitly, through interaction with unreliable networks or faulty hardware. When employed in safety-critical applications, it is important to rigorously analyse the behaviour of these systems. This can be done with a formal verification technique called *model checking*, which establishes *properties* of systems by algorithmically considering all execution scenarios. In the presence of probabilistic behaviour, we consider *quantitative* properties such as "*the worst-case probability that the airbag fails to deploy within 10ms*", instead of *qualitative* properties such as "*the airbag eventually deploys*". Although many model checking techniques exist to verify *qualitative* properties of software, *quantitative* model checking techniques typically focus on manually derived models of systems and cannot directly verify software.

In this thesis, we present two quantitative model checking techniques for probabilistic software. The first is a quantitative adaptation of a successful model checking technique called *counter-example guided abstraction refinement* which uses *stochastic two-player games* as abstractions of probabilistic software. We show how to achieve abstraction and refinement in a probabilistic setting and investigate theoretical extensions of stochastic two-player game abstractions. Our second technique *instruments* probabilistic software in such a way that existing, non-probabilistic software verification methods can be used to compute bounds on quantitative properties of the original, uninstrumented software.

Our techniques are the first to target real, compilable software in a probabilistic setting. We present an experimental evaluation of both approaches on a large range of case studies and evaluate several extensions and heuristics. We demonstrate that, with our methods, we can successfully compute quantitative properties of real network clients comprising approximately 1,000 lines of complex ANSI-C code — the verification of such software is far beyond the capabilities of existing quantitative model checking techniques.

Acknowledgements

Firstly, I would like to express my gratitude to Marta Kwiatkowska for giving me the opportunity to pursue a doctorate study under her supervision as well as for her invaluable support and guidance during my studies. I would also like to thank Gethin Norman and David Parker — their support has been absolutely instrumental in completing this thesis.

I have also been fortunate position to receive support and guidance from to Michael Huth, Daniel Kroening and Josée Desharnais and a special thanks goes to them. Finally, I would like thank the many collegues and visitors who have made the last few years very enjoyable and the EPSRC for funding my studies.

Table of Contents

Chapter 1

Introduction

Software plays an increasingly important role in our society. We entrust software to perform safety-critical tasks in systems such as cars, aircraft and medical equipment, as well as highly responsible tasks such as managing our finances and stock markets. It is crucial to ensure that the software in these systems conforms to its requirements, as failing to do so may have far-reaching financial consequences and may even lead to the loss of life.

The prevalent practice in industry is to look for violations of these requirements by conducting a large number of tests. We recognise the importance of testing. However, because it generally is not feasible to test *every* scenario, we cannot *guarantee* conformance to requirements through testing alone. A long-standing aspiration of computer science is to better this practice through *formal verification techniques* — techniques with which we *can* make rigorous guarantees about conformance to requirements.

An established methodology in formal verification is *model checking* [CE81, QS82]. Model checking is a collective name for algorithmic techniques that establish formal *properties* of systems by systematically exploring *all* possible execution scenarios. Tools that implement such algorithms are called *model checkers*. By "property" we mean formally defined *characteristics* of systems, such as

- "*the program does not dereference null-pointers*" or
- "*the procedure eventually triggers a* `BRAKE` *signal*".

We do not restrict to Boolean-valued characteristics that are either "true" or "false". In fact, the properties we consider in this thesis are typically *real*-valued. We use the terminology that Boolean-valued properties, like the examples above, are *qualitative* properties and real-valued properties are *quantitative* properties. Some examples of quantitative properties are:

- "*the minimum probability of the procedure eventually triggering a* `BRAKE` *signal*",
- "*the maximum probability of establishing a network connection*" and
- "*the minimum expected number of control signals sent before a timeout*".

When considering *qualitative* properties, model checking equates to *verifying* or *refuting* the validity of a property. In contrast, model checking *quantitative* properties typically amounts to *computing* a value. Through computing quantitative properties, we can trivially verify or refute requirements such as:

- "*the probability that the procedure fails to trigger a* `BRAKE` *signal is at most* 10^{-9}".

Whereas *qualitative* properties help establish conformance to *functional* requirements, *quantitative* properties help to establish *non-functional* requirements such as performance, reliability or robustness requirements.[1]

Originally, model checking techniques focused on verifying *qualitative* properties of hand-written, formal *models* of systems [Pnu77, CE81, QS82]. Evidently, verifying a hand-written model of a system is only meaningful if the model accurately reflects the behaviour of the system. Through a number of innovations in, e.g., data structures [McM92], abstraction methods [GS97, Kur94, CGJ+00] and decision procedures [BCCY99, HJMM04, JM05, McM06], it is now possible to model check software directly from its source code, eliminating any concerns regarding the inaccuracy of models. The success of software model checking is demonstrated by many success stories in this area [VHG+02, BR01, BCLR04, HJMS03, CCG+04, CKSY05, CKL04, CPR06].

Moreover, there are now mature methods for computing *quantitative* properties. The focus of this thesis is on *probabilistic* model checking, where properties that relate to the

[1] A more detailed categorisation of *functional* and *non-functional* requirements can be found in most software engineering textbooks (see, e.g., [LL02, Chapter 4]).

probabilistic behaviour of systems are considered. Taking into account this behaviour is essential to verify, say, robustness requirements of systems that are prone to *failures* or performance requirements for systems that employ randomisation. Probabilistic model checking is a mature field with applications in communication protocols [DKNP06, KNS03], randomised distributed algorithms [KNS01, KN02], security [Ste06, Shm02] and systems biology [HKN+08].

The fields of *software model checking* and *probabilistic model checking* are currently, to a great extent, separate. That is, software model checkers focus almost exclusively on verifying *non-probabilistic* properties of software and probabilistic model checkers focus exclusively on verifying manually prepared models of systems. There are obvious benefits to a marriage of the two fields. That is, combining these techniques would allow us to establish quantitative properties of software which, in turn, may help ensure that, say, safety-critical systems conform to certain *non-functional* requirements. Applying quantitative analyses to software can also help us gain insight into the probabilistic behaviour of software and may help detect anomalies or trends that may go undiscovered with qualitative analyses.

In this thesis, we endeavour to close the gap between software model checking and probabilistic model checking. As a platform for doing so, we develop model checking techniques to compute quantitative properties of what we call *probabilistic software* — computer programs that are subject to probabilistic behaviour. We remark that, in most of the programs we consider, probabilistic behaviour is introduced by the interaction of the program with the environment, as opposed to being explicitly introduced by the programmer. We make a point of targeting *real* software — computer programs that can be compiled and executed using conventional compiler suites with little or no adjustments — as opposed to programs written in less expressive, formal modelling languages.

To provide probabilistic programs with the ability to specify probabilistic behaviour, we provide a library of *quantitative functions*, partially depicted in Figure 1.1. These functions exist purely for the purposes of verification — we do not expect real programs to use these functions. We *do* argue that for many programs it is meaningful to temporarily insert these quantitative functions into the source code for the purposes of model checking.

FUNCTION PROTOTYPE	INFORMAL SEMANTICS
`bool coin(int n, int d)`	Returns `tt` or `ff` with *probability* `n`/`d` and $1 - \texttt{n}/\texttt{d}$, respectively.
`int uniform(int ub)`	Returns an `int` in the range $[0, \texttt{ub}[$ *uniformly at random*.
`int ndet(int ub)`	Returns an *arbitrary* `int` in the range $[0, \texttt{ub}[$.
`void cost()`	Marks a control-flow location as a *cost* location.
`void target()`	Marks a control-flow location as a *target* location.

FIGURE 1.1: Examples of special functions that are at the disposal of probabilistic ANSI-C programs.

To illustrate this, suppose `out_p()` is a function that writes data to some unreliable hardware that returns 0 upon a failure. If we have statistical evidence suggesting that calls to `out_p()` fail every one out of ten times on average, then it could be meaningful to replace a call to this function with a call to `coin(9,10)`. Once replaced, we effectively model check the program under the *assumption* that the hardware fails with this frequency.

The properties we consider are also specified through the functions in Figure 1.1. In particular, we use `target()` and `cost()` to mark control-flow locations as *target* and *cost* locations. Informally, the quantitative properties we consider are

- "*the minimum probability of eventually calling* `target()`",
- "*the maximum probability of eventually calling* `target()`",
- "*the minimum expected number of times* `cost()` *is called*", and
- "*the maximum expected number of times* `cost()` *is called*".

We argue that by putting `target()` and `cost()` in meaningful places of programs we are able to specify a wide range of quantitative properties of programs in a relatively straightforward way. We remark that, while simple to specify, model checking non-trivial properties such as these is generally undecidable [Ric53] and, hence, we cannot expect to develop verification techniques that will work for all probabilistic programs. We therefore focus on developing techniques that work well on selected classes of programs in practice.

We propose two techniques for verifying these properties. Our first technique is inspired by the success of automated *abstraction-refinement* methods in software model checking [CGJ+00, BR01, HJMS03, CCG+04, CKSY05]. With these methods, increas-

ingly precise *abstract models* of programs are considered in an abstraction-refinement loop. In Chapter 5, we propose an instantiation of such an abstraction-refinement loop that considers stochastic *game abstractions* [KNP06] as abstract models of *probabilistic* software. We discuss how to construct such abstractions directly from source code and how to realise refinement when dealing with software that is subject to probabilistic behaviour. The approach in this chapter is fully implemented and we present extensive experimental results over a large range of case studies. We also discuss various heuristics and optimisations employed in the implementation.

Our abstraction-refinement technique is subject to certain limitations. As an example, due to the methods we use to abstract and refine probabilistic programs, we are restricted in the types of probabilistic behaviour that we can handle. Moreover, constructing game abstractions is often computationally expensive, affecting the scalability of our approach. A cause of both limitations is that it is not clear how to *approximate* the transition function of game abstractions. Most abstraction-based software model checkers employ such approximations to improve scalability (see, e.g., [DD01, BMR01, CKSY05, JM05, KS06]). In Chapter 6 we address this problem by proposing an improved game-based abstraction framework.

Our second technique — which we call *instrumentation-based* verification — reduces a *quantitative* verification problem to a *qualitative* one. More specifically, this technique, presented in Chapter 7, *instruments* probabilistic programs in such a way that we can compute quantitative properties of the original probabilistic program by verifying several qualitative properties of the instrumented program. The main motivation for pursuing this approach is that, in our experience, adapting existing qualitative verification techniques for non-probabilistic programs to deal with probabilistic behaviour requires fundamental changes to the underlying technique. This, in turn, adversely affects the effectiveness and scalability of the verification technique. For example, with our abstraction-refinement technique, due to the presence of probabilistic behaviour, we were unable to use state-of-the-art interpolation-based refinement methods such as [HJMM04, JM05, McM06]. In contrast, with our instrumentation-based technique, we reason about quantitative properties by verifying qualitative properties of *non-probabilistic* programs. This allows us

to *directly* use state-of-the-art methods in non-probabilistic software verification — like interpolation-based refinement — for the purposes of computing quantitative properties of probabilistic programs. The instrumentation-based technique is also fully implemented and we again present extensive experimental results to evaluate it.

We will show that we are able to compute quantitative properties of real network programs of approximately 1,000 lines of complex ANSI-C code with *both* techniques — the verification of such programs is far beyond the capabilities of existing quantitative verification techniques. Firstly, these programs have an extremely large data space that is far larger than existing finite-state probabilistic verification tools can handle. To illustrate this, observe that a program comprising just two 32-bit integer variables already has a data space of 2^{64} ($\approx 1.8 \cdot 10^{19}$) states. One of the largest case studies that has been verified with finite-state probabilistic verification tools is the Bluetooth model in [DKNP06], where a state space of 10^{10} is reported. Secondly, these programs use complex programming constructs such as functions, pointers, function pointers, arrays, structs and bit-level arithmetic, whereas existing verification tools for probabilistic systems target models in simpler formal modelling languages in which most of these programming constructs cannot be used.

Other Publications Selected parts of this thesis have been published as co-authored papers. A model-level abstraction-refinement technique using game abstractions was presented in [KKNP10] and is joint work with Marta Kwiatkowska, Gethin Norman and David Parker. The author contributed to the development of the refinement procedure and the development of case studies.

The abstraction-refinement approach in Chapter 5 was presented in [KKNP09] and is joint work Marta Kwiatkowska, Gethin Norman and David Parker. The author contributed to abstraction and refinement procedures in this approach and was responsible for the implementation and experimental evaluation of this approach.[2] The abstraction procedure in this chapter is partially based on a method for constructing game abstractions for models specified in a guarded command language, presented in [KKNP08]. This is also joint work with Marta Kwiatkowska, Gethin Norman and David Parker. The au-

[2] Excluding the model checker for game abstractions.

thor contributed to the development of this abstraction method and was responsible for its implementation and experimental evaluation.

The extension of game abstractions in Chapter 6 was presented in [KH09] and is joint work with Michael Huth. The author contributed to the development of this framework and is solely responsible for the details presented in Chapter 6.

The instrumentation-based approach in Chapter 7 is joint work with Daniel Kroening, Marta Kwiatkowska, Gethin Norman and David Parker. The author contributed to the development of this approach and is solely responsible for the details presented in Chapter 7, as well as the implementation of this approach and the experimental evaluation.

Chapter 2

Related Work

In this chapter, we review the literature that is relevant to this thesis. Central to our work is a formal verification technique called *model checking*, which was independently proposed by Clarke and Emerson [CE81] and Quielle and Sifakis [QS82]. Model checking methods establish properties of systems by algorithmically considering all execution scenarios of the system. There is a large body of work on model checking, focusing on different kinds of systems. This chapter is structured as follows: in Section 2.1 we first discuss model checking methods for *software* systems then, in Section 2.2, we discuss model checking approaches for *probabilistic* systems.

2.1 Software Verification

In this section, we discuss *automated* software verification methods. We start with *model checking* methods that employ *abstraction refinement* in Section 2.1.1. Section 2.1.2 then considers some alternative approaches to software verification. All methods in this section focus on *qualitative* properties of *non-probabilistic* software — we discuss methods for *probabilistic* systems in Section 2.2.

2.1.1 Automated Abstraction Refinement

Verifying programs by directly analysing their low-level semantics is not necessarily practical for many classes of programs. Therefore, a substantial amount of research in software

verification is concerned with technologies and tools that enable us to establish properties of programs by using *abstraction*.

Abstraction In model checking, abstraction is typically realised by constructing *abstract models* of programs which are themselves subjected to model checking. Various abstraction frameworks have been suggested in literature. Arguably, the most prominent framework for abstraction in model checking is *existential abstraction* [CGL94]. Existential abstractions are normal transition systems that over-approximate the behaviours of the program under consideration. Due to the nature of existential abstractions, every path of a program corresponds to a path in the abstract model. Hence, if a property holds for *all* paths of the abstract model, then this property must also hold for every path of the program. Intuitively, this means we can *verify*, e.g., *safety* properties — properties that can be refuted through a finite path of the program. However, *refutation* of these properties cannot be achieved via existential abstractions alone and is typically achieved via analyses of abstract counter-examples. Existential abstractions are employed in many prominent software model checkers, including SLAM [BR01], BLAST [HJMS03], MAGIC [CCG+04] and SATABS [CKSY05].

The inability to directly refute safety properties by model checking abstract models has led to the consideration of so-called *three-valued* or *multi-valued* abstractions frameworks, through which we can directly verify and refute safety properties. Most notable such frameworks are the *modal* and *mixed abstractions* of [LT88, DGG97]. In these frameworks, the abstract models have two transition functions — one for over-approximation and one for under-approximation. The added ability to under-approximate behaviours of the program allows us to refute properties by model checking abstract models. This eliminates the need to analyse counter-examples for the purposes of refutation. These abstraction frameworks have also been applied to software in [GC06].

There are also various other abstraction frameworks that, to the best of our knowledge, have not yet been used in software verification. We mention extensions of modal and mixed abstractions in, e.g., [LX90, SG06], alternating abstractions in [AHKV98], abstraction frameworks for turn-based games in [HJM03, dAGJ04, dAR07b] and abstraction frameworks that employ ranking functions, fairness or tree automata in [KP00, DN04,

Nam03, DN04, DN05].

Constructing abstractions In software model checking, we typically only consider abstract models that are abstractions of the program *by construction*. The main focus in software model checking is on existential abstractions induced by *predicates*. Essentially, these abstractions do not touch the control-flow of programs and abstract the data space of programs by keeping track of the validity of a finite set of predicates [GS97]. The process of constructing such abstractions is the subject of much research.

In [GS97, SS99, DDP99, BR01], the construction of abstractions was done via calls to general-purpose theorem provers. In practice, this is often expensive. More recently, many other abstraction methods have been suggested. We mention techniques based on SAT solvers [LBC03, CKSY04], symbolic data structures [LBC03, CCF+07], SMT solvers [LNO06] and proof-based methods [JM05, LBC05, KS06]. In terms of performance, these techniques typically perform better than the method in, say, [GS97]. In practice, however, the time required to compute abstractions of programs is still worst-case exponential in the number of predicates [KS06]. To mitigate this, almost all abstraction-based model checkers over-approximate the transition relation of existential abstractions (see, e.g., [DD01, BMR01, BPR03, CKSY05, JM05, KS06]). According to [KS06], this is done in all prominent abstraction-based software model checkers except for MAGIC.

Automated abstraction refinement Independent of the abstraction framework that is used, there are typically many abstract models one could construct for any given program. In practice, we need abstractions that are both small enough to be model checked efficiently and precise enough to be able to verify or refute the property under consideration. The main challenge in abstraction-based software model checking is to find good abstractions *automatically*.

A recognised methodology to automatically find abstractions is what we call the *abstraction-refinement paradigm*. With this paradigm we consider increasingly precise abstract models in an *abstraction-refinement loop*. We start with an imprecise abstraction that is cheap to construct. Then, in subsequent iterations of the loop, we consider increasingly precise abstractions until we have either verified or refuted the property un-

der consideration. The key step in this abstraction-refinement loop is the *refinement step* which, given an abstraction that is too imprecise, selects a more precise abstract model of the program.

The most prominent abstraction-refinement methodology, employed by model checkers such as SLAM [BR01], BLAST [HJMS03], MAGIC [CCG+04] and SATABS [CKSY05], is that of *counter-example guided abstraction refinement* (CEGAR) [CGJ+00]. This abstraction-refinement methodology employs existential abstractions induced by predicates. We construct an abstract model in each iteration of the abstraction-refinement loop. By model checking the abstraction, we may establish that the abstract model satisfies the safety property under consideration. In this case, we can conclude the program also satisfies it. Otherwise, we try and refute the safety property by taking an abstract counter-example and mapping it to a concrete counter-example. If there is no corresponding concrete counter-example, then the abstract counter-example is said to be *spurious*. The key idea of CEGAR is that, when verification and refutation fails, we refine the abstraction by eliminating a spurious abstract counter-examples. This elimination can be realised by adding predicates which prevent the spurious abstract counter-examples from occurring in the refined abstraction. In practice, this is either achieved by taking weakest preconditions [CGJ+00, BR01] or by using interpolating decision procedures [HJMM04].

In practice, most CEGAR model checkers approximate the transition function of abstract models. In this case, the principal cause of imprecision in the abstract model may not be caused by missing predicates, but may be caused by an imprecise abstract transition function instead. To deal with this kind of imprecision, an additional level of refinement is needed. This kind of refinement is discussed in, e.g., [DD01, BMR01, JM05].

An alternative to CEGAR is the abstraction-refinement approach suggested in [SG07]. This approach uses modal abstractions instead of existential abstractions. Because this abstract model both over and under-approximates the possible behaviours, a model check on abstract models both over and under-approximates the validity of the property under consideration. This means we can both verify and refute properties by model checking abstractions. Due to the extended range of properties we can deal with under this abstraction framework, however, it is no longer the case that every property can be refuted

with a simple finite path of the program. Therefore, instead of using counter-examples to refine abstractions, the refinement step in [SG07] analyses the current abstract model and finds the states of the abstract model that are responsible for the current imprecision. It then applies refinements locally to these states. To our knowledge, this method has not yet been used to verify software.

There are various other abstraction-refinement methods that are relevant to this thesis. Most notably, we mention abstraction-refinement methods for solving turn-based games [HJM03, dAR07b]. Although the semantic models and verification problems involved are strict generalisations of those usually considered in software model checking, like the method in [SG07], these methods have not been applied in the context of software verification.

2.1.2 Other Software Verification Techniques

Besides abstraction-refinement methods there are various other automated software verification methods. In this section, for completeness, we will discuss *search-based model checking*, *bounded model checking* and *abstract interpretation*, respectively.

Search-based model checking A recognised model checking method is the automata-based approach introduced in [VW86]. This method checks whether all executions of a transition system satisfy a certain qualitative property by reducing this verification problem to checking the emptiness of a language accepted by an automaton. In [CVWY90], it was shown how this emptiness check can be achieved via a nested depth-first search. This search-based model checking method has successfully been applied to software (see, e.g., [HS00, HD01, VHG+02, MPC+02, CW02, AQR+04, God05]). For many programs, however, the automata generated by this method have an extremely large or infinite state space and, in practice, an exhaustive search over such automata is not always feasible.

Bounded model checking Another model checking method, called *bounded model checking*, was introduced by Armin Biere in [BCCY99], and has been applied to software model checkers such as CBMC [CKL04, AMP06]. The key idea in this method is to try and *refute* safety properties by considering increasingly long paths of the program as po-

tential counter-examples to these properties. All paths of a certain length are symbolically represented with a formula in propositional logic or a formula in a first-order theory in such a way that the formula is satisfiable if and only if there is a counter-example of this length. A SAT or SMT solver is then used to decide whether the formula is satisfiable. The bounded model checking method is effective in *refuting* qualitative safety properties that have shallow counter-examples, but is not as effective in *verifying* safety properties.

Lazy abstraction with interpolants Another model checking method is described in [McM06]. Instead of directly constructing abstract models, this model checking approach sets out to find inductive invariants of the program through repeated applications of an interpolating decision procedure — decision procedures that produce proofs. Essentially, the model checker calls an interpolating decision procedure for many successive control-flow paths to a "bad" location. If one such path is feasible then a counter-example to a safety property has been found. When infeasible, the decision procedure generates a proof which can used to augment the inductive invariants. Whether this model checking approach is effective depends on whether the decision procedure that is used is able to guess the loop invariants of programs. We discuss this method in more detail in Chapter 7.

Abstract interpretation Another prominent way in which abstraction is employed in software verification is through *abstract interpretation* [CC77, CR79]. Abstract interpretation provides a mathematical framework through which we can execute a program abstractly. This abstract execution can be used to verify safety properties and has been successfully applied to software in tools such as ASTRÉE [CCF+05]. The main disadvantage of abstract interpretation is that, unlike model checking, there is no provision to improve the quality of the abstraction through refinements. The emphasis of abstract interpretation is to directly abstract the actual *computations* of programs instead of constructing abstract models of the program and model checking them. A link between abstract interpretation and abstraction-based model checking that we will use in our thesis is that the computation we perform when model checking an abstract model is often an abstract interpretation of the computation we perform when model checking the program directly.

2.2 Probabilistic Verification

We now discuss verification techniques and tools for *probabilistic* systems and, in particular, probabilistic model checking methods. We focus on systems that exhibit both *non-deterministic* and *probabilistic* behaviour. We argue that, in the context of software verification, the ability to deal with non-determinism is important. For example, non-determinism can be used to model calls to library functions for which no source code is available. Markov Decision Process (MDP) are models which naturally capture both non-deterministic and probabilistic behaviour. Our main discussion is therefore on the verification of systems with MDP semantics.

We remark that sometimes MDPs are defined to be *deterministic* in the sense that each non-deterministic choice is labelled with a distinct action label. The MDPs we use in this thesis are not deterministic in this sense. We also do not use action labels. Our models are closer to the *probabilistic automata* of [Seg95]. However, we only use one action label. Besides MDPs, various other models are prominent in probabilistic verification. Notably, many verification methods deal with models that do not exhibit any non-deterministic behaviour. There are also models for modelling continuous-time or real-time behaviour in systems. We discuss relevant work on these formalisms where appropriate.

Like in software verification, the use of abstraction has been the topic of much reseach in probabilistic verification. To structure this section, we first discuss probabilistic verification methods that do not use abstraction in Section 2.2.1, and then discuss abstraction methods for probabilistic systems in Section 2.2.2.

2.2.1 Probabilistic Verification without Abstraction

In this section, we discuss verification techniques and tools for probabilistic systems that do not employ abstraction. We start with what we call *probabilistic model checking* and then discuss *probabilistic equivalence checking*.

Probabilistic model checking In this thesis, we consider *probabilistic reachability properties* and *cost properties* for MDPs. For probabilistic reachability properties, we

are interested in computing the minimum and maximum probability of reaching a set of states in the MDP, whereas for cost properties we are interested in the the expected total cost we incur when executing the MDP. Model checking algorithms for these properties were first considered in [CY90, BdA95, CY98] and [Put94, dA99], respectively. The algorithms in [CY90, BdA95, CY98] consist of both graph-based algorithms and a numeric computation using linear programming. As an alternative to linear programming, for reasons of scalability, many tools use an approximation algorithm called *value iteration* [Put94]. In [Par02], it is shown that an effective way to implement probabilistic model checking algorithms for MDPs is value iteration via symbolic data structures.

We are aware of several tools that implement these methods: PRISM [HKNP06], LiQuor [CB06], ProbDiViNe [BBv+08], RAPTURE [DJJL01, DJJL02, JDL02] and PASS [WZH07, HWZ08, HHWZ10]. Due to their use of abstraction techniques, we will discuss RAPTURE and PASS in the next section.

The model checker PRISM employs the symbolic value iteration method described in [Par02]. Models in PRISM are specified in a simple compositional guarded command language. Besides MDPs, PRISM can also deal with models that have continuous-time or real-time behaviours. The model checkers LiQuor and ProbDiViNe implement the verification methods in [CY90, CY98]. A main feature of ProbDiViNe is that it implements these methods in a distributed fashion. Like PRISM, the MDPs in ProbDiViNe are specified in a simple guarded command language. LiQuor, however, considers models specified in PROBMELA, a probabilistic adaptation of the process meta-language accepted by the popular non-probabilistic model checker SPIN [Hol03]. The model checker MRMC [KZH+09] is unable to deal with systems with MDP semantics, but can verify systems without non-determinism, or non-deterministic systems with continuous-time semantics.

The simplicity of the modelling languages used by PRISM, LiQuor and ProbDiViNe makes it impractical to directly target software systems with these finite-state model checkers. In [ZvB10], however, probabilistic adaptation of Java Pathfinder is used to extract models from Java programs. Essentially, the state space of probabilistic Java programs is traversed and, every so often, the model checker MRMC [KZH+09] is used to model check the state space that has been explored so far. We mention that non-

deterministic behaviour is not supported in this approach. Moreover, although this approach deals with real, compilable probabilistic software, we argue that, for many programs, the state space is too large to traverse explicitly. The experimental evaluation in [ZvB10] is limited and there is little evidence to suggest this method scales beyond small probabilistic programs.

Probabilistic counter-examples Counter-examples to probabilistic safety properties are not usually individual paths to "bad" states. Instead, they comprise a *set* of paths [HK07]. Due to their complex nature, prominent probabilistic verification tools such as PRISM or MRMC do not generate counter-examples. However, algorithmic methods to generate probabilistic counter-examples were introduced in [HK07]. In [WBB09], it was proposed to enumerate paths of probabilistic counter-examples via non-probabilistic bounded model checking methods. In [HK07, WBB09], only fully probabilistic models are considered. However, [HK07] has been adapted to deal with MDPs in [AL09].

Another approach is taken in [FHT08], where bounded model checking methods are directly applied to probabilistic systems. In this approach, instead of using SAT or SMT, a probabilistic extension of SMT, called SSMT, is used. With SSMT, one can directly encorporate a notion of probability when encoding the behaviour of a model in a first-order formula. In this way, a decision procedure for SSMT can be used to decide whether there is a probabilistic counter-example of a certain structure, akin to non-probabilistic bounded model checking methods. It should be noted that, in [FHT08], the models under consideration are hybrid systems. Hybrid systems mix continuous-time and discrete-time semantics and strictly subsume MDPs.

Probabilistic equivalence checking An automated approach for checking the equivalence of probabilistic programs is discussed in [LMOW08]. The programs that are considered are specified in a simple C-like language that is restrictive enough to ensure that the equivalence check is decidable. This approach is implemented in a tool called APEX. It is possible to use APEX to compute the probabilistic reachability properties used in this thesis. On certain models and properties, APEX can significantly outperform finite-state model checkers such as PRISM. Unfortunately, APEX is not able to deal with

non-determinism. Moreover, the programs considered by APEX are still some distance removed from real, compilable software.

2.2.2 Abstraction of Probabilistic Systems

Like in non-probabilistic verification, the key to verifying increasingly complicated *probabilistic* systems is through the use of *abstraction*. There are many ways in which we can employ abstraction. In Section 2.2.2, we first discuss probabilistic analogues of the abstraction refinement approach discussed in Section 2.1.1. Then, in Section 2.2.2 and Section 2.2.2, we discuss *probabilistic predicate transformers* and *probabilistic abstract interpretation*, respectively.

Probabilistic abstraction refinement At the heart of most abstraction techniques used in model checking are simulation preorders, which are also called *abstraction* or *refinement* preorders, depending on context. A notion of simulation specifically devised for probabilistic systems was first introduced in [JL91]. This simulation preorder builds on a notion of bisimulation for probabilistic systems that was introduced in [LS91]. Both [LS91] and [JL91] considered MDPs that are deterministic in the sense that each non-deterministic choice is labelled with a distinct action label. In [SL94], notions of simulation and bisimulation were introduced for probabilistic automata, which are, like our MDPs, not deterministic in this sense. An important insight in [SL94] is that, for probabilistic systems, it is natural to allow the abstract model to simulate behaviours of the concrete system via probabilistic combinations of behaviours, using so-called *combined transitions*.

A simulation preorder called *strong probabilistic simulation*, which uses combined transitions, is proposed in [SL94]. If we interpret the strong probabilistic simulation of [SL94] as an abstraction preorder then it is, in effect, the probabilistic analogue of the *existential abstractions* in [CGL94]. It shares the typical characteristics of existential abstractions in that abstract models over-approximate the possible behaviours of the systems they abstract. Because of this, we can only obtain a one-sided non-trivial bound on the quantitative properties of MDPs we consider in this thesis via these abstractions. For example, for probabilistic reachability properties, these abstract models yield a lower

bound on the minimum reachability probability and an upper bound on the maximum reachability probability of the MDP that is being abstracted. The abstraction preorders in [SL94] are at the heart of various abstraction-refinement tools for MDPs, including RAPTURE [DJJL01, DJJL02, JDL02] and PASS [WZH07, HWZ08, HHWZ10], which we will discuss next.

In RAPTURE, instead of actually computing probabilistic reachability propeties, we decide whether the minimum or maximum probability of reaching a set of states in an MDP is above or below some user-defined threshold. The MDPs are specified in a simple CSP-based automata language. Prior to constructing any abstractions, RAPTURE performs a reachability analysis on the MDP. It then constructs successively precise abstract models in an abstraction-refinement loop. These models are strong probabilistic simulations of the system by construction. The abstract models are constructed via operations on symbolic data structures. Then, following [CY90, CY98, BdA95], the probabilistic reachability probabilities of an abstract model are computed via a reduction to linear programming. If we are interested in the minimum reachability probability and the lower bound on the minimum probability is above the user-defined threshold or, if we are interested in the maximum probability and the upper bound is below the threshold, then the abstraction-refinement loop terminates. This may not always be possible. Given the over-approximating nature of the abstract MDPs, if the actual minimum reachability probability of the system is below the user-defined threshold (or if the maximum reachability probability is above it) then no such abstract model exists. RAPTURE has an additional check in place that tests whether the abstraction is a probabilistic bisimulation of the system. If this is the case then the reachability probabilities of the abstract MDP coincide with those of the concrete MDP. It is only through this check that we can decide whether the minimum probability is below the threshold or the maximum probability is above it. If the abstract model is not precise enough to establish whether the threshold is met, and the abstract model is not a bisimulation of the system under consideration, then RAPTURE refines the abstract model. Refinements are realised by splitting states of the abstract model. Essentially, RAPTURE looks for abstract states that abstract states that are not bisimilar because they induce different abstract transitions in the abstract model.

It then splits the abstract state to ensure that, in the next abstraction, these states are no longer abstracted by the same abstract state. We remark that both the abstraction and the refinement procedures in RAPTURE are defined on the level of individual states and transitions, as opposed to on the language-level.

A second tool based on strong probabilistions simulations is PASS [WZH07, HWZ08, HHWZ10]. The focus of this tool is to decide whether maximum reachability probabilities of MDPs are above or below a certain probability threshold. The MDPs considered in PASS are specified as PRISM models, augmented with the ability to define variables with infinite ranges. Unlike RAPTURE, abstract models in PASS are induced by language-level predicates, akin to predicate abstractions in software verification [GS97]. Strong probabilistic simulations of the original MDP are constructed using probabilistic adaptations of the SAT/SMT-based abstraction methods in [LBC03, CKSY04, LNO06]. Like in RAPTURE, if the abstract model yields an upper bound that is below the threshold then the abstraction-refinement loop is finished. If this is not the case, then, in keeping with CEGAR [CGJ+00], PASS analyses counter-examples to try and establish if the maximum reachability probability exceeds the user-specified threshold. In a probabilistic setting, counter-examples can be viewed as a set of paths [HK07, AL09]. In cases where the analysis fails, spurious paths of the counter-example are eliminated using interpolation-based refinement methods [HJMM04]. In contrast to RAPTURE, the abstraction and refinement procedures in PASS work directly on the language-level representation of MDPs. For example, instead of splitting individual abstract states, the refinement step in PASS introduces predicates.

More recent versions of PASS [WZ10] use the *game-based* abstractions of [KNP06]. In their ability to both under and over-approximate the possible behaviours of systems, these game-based abstractions are a probabilistic analogue of three-valued abstraction frameworks such as [LT88, DGG97]. Game-based abstractions yield both non-trivial lower and upper bounds for both minimum and maximum reachability probabilities. We will discuss game-based abstraction in more detail in, e.g., Chapter 3. Because game-based abstractions are more expensive to construct, in game-based PASS, the abstractions that are constructed are actually approximations of game abstractions, called *parallel abstrac-*

tions. Unlike MDP-based PASS, the version of PASS that uses game-based abstractions is also able to consider minimum reachability probabilities and, because the abstractions provide both lower and upper bounds, this version of PASS no longer involves counter-examples. The refinement procedure in [WZ10] follows the lines of three-valued abstraction-refinement methods such as [SG07] — it those states of the abstract model that are responsible for the difference in the lower and upper bounds. In [WZ10], these refinements do not take into account that imprecisions may be introduced by the approximating nature of parallel abstractions. It is reported in [WZ10] that, compared to MDP-based PASS, game-based PASS is applicable to more properties, is generally faster and generally finds smaller abstractions.

Besides RAPTURE and PASS, a third abstraction-refinement method for MDPs is suggested in [CV10]. Like PASS, this method is presented as a direct probabilistic adaptation of CEGAR. We remark that, to our knowledge, this approach is currently purely theoretic and does not feature an implementation.

For completeness we also mention abstraction frameworks with abstract models in which probabilities have been replaced with intervals [Hut05, FLW06, KKLW07]. To the best of our knowledge, none of these interval-based approaches has been applied to systems that are non-deterministic. We are also unaware of any applications of interval-based abstractions in the context of abstraction refinement for MDPs.

Magnifying lens abstraction Another relevant abstraction technique is that of *magnifying lens abstraction* [dAR07a]. Instead of considering abstract models of systems, abstraction in this approach is directly applied to the value iteration algorithms used to compute quantitative properties of MDPs. The state space of the system is partitioned into regions and, during value iteration, the value of all but one "magnified" region is represented with a single value. By continuously changing the magnified region one can compute the quantitative properties under consideration without ever having to represent the system in its entirety. Automatic methods to refine the partion into regions are also discussed in [dAR07a]. To our knowlegde, this abstraction technique has not yet been applied to, say, probabilistic software, where it is impractical to deal with the low-level MDP semantics directly.

Probabilistic predicate transformers The mathematical machinery through which we classically reason about sequential, non-probabilistic programs, e.g. [Flo67, Hoa69, Dij75], has also been adapted to a probabilistic setting in [MM05, dHdV02]. Essentially, this work provides a mathematical framework to reason about probabilistic programs specified in a simple probabilistic guarded command language (pGCL). Recently, there have been a number of automated techniques based on the work of [MM05]. We mention [KMMM10], which proposes automated quantitative invariant generation of pGCL programs via quantitative adaptations of invariant generation methods such as [CSS03]. An advantage of [KMMM10] is that it is possible to verify programs in a parameterised fashion. We can analyse programs in which, say, a failure probability is left unspecified. We also mention the work in [BW10, NM10] where a quantitative analogue of predicate abstraction is introduced, called random variable abstraction. However, these approaches are currently not fully automated. We remark that the case studies considered in this thread of work are typically relatively small and are some distance removed from being real, compilable software.

Probabilistic abstract interpretation Another approach to employ abstraction in probabilistic systems is through probabilistic adaptations of abstract interpretation. One such adaptation is proposed in [Mon00, Mon01]. Through the probabilistic analysis in this approach it is possible to obtain upper bounds on, e.g., the maximum reachability properties considered in this thesis. A downside is that, like normal abstract interpretation, there is no guarantee as to how precise this upper bound is, nor is there any automated method to obtain more precise upper bounds. Although the methods in [Mon00, Mon01] can be applied to software systems, we are unaware of any experiments on substantial programs. Another probabilistic adaptation of abstract interpretation-based approach is that of [DPW00, DPW01]. The main benefit of the approach in [DPW00, DPW01] is that it is possible to measure how precise the analysis is. However, this approach does not have the necessary machinery to deal with non-determinism. Finally we mention that the approach of [DPW00, DPW01] has also been extended to programs with continuous-time semantics in [Smi08].

Chapter 3

Background

In this chapter, we discuss the required background material for this thesis. We start, in Section 3.1, by establishing some basic notation and terminology. Section 3.2 then introduces some necessary definitions. In Section 3.3, we define the mathematical models we will use to model probabilistic programs. Finally, in Section 3.4, we discuss an abstraction method for such models.

3.1 Notation & Terminology

We write $\mathbb{N}$, $\mathbb{R}$, $\mathbb{Q}$ and $\mathbb{B}$ to denote *naturals*, *reals*, *rationals* and *Booleans*, respectively. We write `tt` and `ff` to denote "true" and "false" and write $[0,1]$, $[0,\infty[$, and $[0,\infty]$ to denote the unit interval, the non-negative reals or the non-negative reals extended with positive infinity, respectively. For a set S, we write $\mathbb{P}S$ to denote the powerset of S and, for notational convenience, we write $\overline{\mathbb{P}}S$ to denote $\mathbb{P}S \setminus \emptyset$ — i.e. the set of all non-empty subsets of S. For sets S_1, S_2 we write $S_1 \uplus S_2$ for the disjoint union of S_1 and S_2.

We will use AP to denote a fixed, finite set of *atomic propositions*. We will use these propositions in formal models of systems. We will mostly limit our attention to the atomic proposition $\mathbf{F} \in \mathrm{AP}$, which we use to mark target states of systems.

Countable sums & weights For a countable index set I and a family of values $\{r_i\}_{i \in I}$ in $[0,\infty]$ we we often use a countable sum $\sum_{i \in I} r_i$. We interpret addition over the non-

negative reals extended with infinity in the obvious way. Formally, as we are dealing with non-negative terms only, we define the result of the sum $\sum_{i \in I} r_i$ as:

$$\sup_{n \to \infty} \left(\sum_{i=0}^{n} r_i \right) .$$

This supremum yields a value in $[0, \infty]$ and may be infinite. It is also simple to show that this value is independent of the ordering of the terms. We call a countable, indexed family of non-negative reals, $\{w_i\}_{i \in I}$, a *family of weights* if and only if $w_i \in [0, 1]$ for every $i \in I$ and $\sum_{i \in I} w_i = 1$.

Relations & Functions Let S_1, S_2 and S_3 be sets, let $R \subseteq S_1 \times S_2$ and $R' \subseteq S_2 \times S_3$ be relations and let $f : S_1 \to S_2$ and $f' : S_2 \to S_3$ be functions. We write $R' \circ R$ and $f' \circ f$ to denote the compositions of these relations and functions, respectively. We write R^{-1} for the relational inverse of R. The function f corresponds to the relation:

$$\{\langle s_1, s_2 \rangle \in S_1 \times S_2 \mid f(s_1) = s_2\} .$$

We write f^{-1} to denote the relational inverse of this relation. For a set $S_1' \subseteq S_1$, we write $R(S_1')$ to mean the image of S_1' in R, i.e.

$$R(S_1') = \{s_2 \in S_2 \mid \exists s_1' \in S_1' : \langle s_1', s_2 \rangle \in R\} .$$

Analogously we write $f(S_1')$ to denote the image of S_1 under the relation corresponding to f. For all $s_1 \in S_1$, we write $R(s_1)$ to mean $R(\{s_1\})$.

We call the relation $R \subseteq S_1 \times S_2$ *left-total* if $|R(s_1)| \geq 1$ for all $s_1 \in S_1$ and *right-unique* if $|R(s_1)| \leq 1$ for all $s_1 \in S_1$. We call R *right-total* and *left-unique* when R^{-1} is left-total and right-unique, respectively. In addition to this, when $S_1 = S_2$, we will call R *reflexive*, *transitive*, *anti-symmetric*, *total* or a *preorder*, a *partial order*, a *total order* or an *equivalence relation* in accordance with standard definitions.

Probability distributions Let S be a set and let $\lambda : S \to [0, 1]$ be a real-valued function on s. We write $\text{SUPP}(\lambda)$ to denote the *support* of λ, i.e. the subset of S comprising

those elements that yield a non-zero value in λ. Formally, we have $\text{SUPP}(\lambda) = \{s \in S \mid \lambda(s) > 0\}$. We call λ a *discrete probability distribution* (or "distribution") if $\text{SUPP}(\lambda)$ is countable and $\sum_{s \in \text{SUPP}(\lambda)} \lambda(s) = 1$. For every element $s \in S$, we write $[s]$ to mean the *point distribution* on s — i.e. the distribution with $[s](s) = 1$. We let $\mathbb{D}S$ denote the set of all distributions on S.

A distribution $\lambda \in \mathbb{D}S$ can be written as a countable sum $\sum_{i \in I} w_i \cdot [s_i]$ for some countable family of weights, $\{w_i\}_{i \in I}$, and some countable family of elements of S, $\{s_i\}_{i \in I}$.

For notational convenience we sometimes perform arithmetic directly on distributions. This arithmetic is applied pointwise. That is, for $\lambda_1, \lambda_2 \in \mathbb{D}S$ we write, say, $\frac{1}{3}\lambda_1 + \frac{2}{3}\lambda_2$, to mean the distribution which yields $\frac{1}{3} \cdot \lambda_1(s) + \frac{2}{3} \cdot \lambda_2(s)$ for every $s \in S$.

In similar spirit, if S_1 and S_2 are sets such that $S_1 \subset S_2$ then we will sometimes interpret a distribution $\lambda_1 \in \mathbb{D}S_1$ as a distribution on S_2 where the probability of all elements in $S_2 \setminus S_1$ is 0. Finally, for distributions $\lambda_1 \in \mathbb{D}S_1$ and $\lambda_2 \in \mathbb{D}S_2$, we write $\text{JOIN}(\lambda_1, \lambda_2)$ for the product distribution in $\mathbb{D}(S_1 \times S_2)$ such that $(\text{JOIN}(\lambda_1, \lambda_2))(s_1, s_2) = \lambda(s_1) \cdot \lambda_2(s_2)$ for all $\langle s_1, s_2 \rangle \in S_1 \times S_2$.

3.2 Preliminaries

In this section, we will introduce some preliminary definitions. We start by defining *weight functions* in Section 3.2.1. Then, in Section 3.2.2 and 3.2.3 we will introduce some necessary basic concepts from *probability theory* and *lattice theory*, respectively.

3.2.1 Weight Functions

When relating the behaviour of two systems, we often use a relation over the respective state spaces to formally relate the systems. For probabilistic systems it is often useful to relate probability distributions instead. To this end, we now discuss a way to lift relations on states to relations over distributions. This definition was introduced in [JL91].

Definition 3.1. *Let S_1 and S_2 be sets and let $R \subseteq S_1 \times S_2$ be a relation. The relation $\mathcal{L}(R) \subseteq \mathbb{D}S_1 \times \mathbb{D}S_2$ contains $\langle \lambda_1, \lambda_2 \rangle \in \mathbb{D}S_1 \times \mathbb{D}S_2$ iff there is function $\delta : S_1 \times S_2 \to [0, 1]$*

— *called a* weight function — *satisfying all of the following conditions:*

$$\lambda_1(s_1) = \sum_{s_2 \in S_2} \delta(s_1, s_2) \qquad (\textit{for all } s_1 \in S_1) \tag{3.1}$$

$$\lambda_2(s_2) = \sum_{s_1 \in S_1} \delta(s_1, s_2) \qquad (\textit{for all } s_2 \in S_2) \tag{3.2}$$

$$\langle s_1, s_2 \rangle \notin R \;\Rightarrow\; \delta(s_1, s_2) = 0\,. \qquad (\textit{for all } s_1 \in S \textit{ and } s_2 \in S_2) \tag{3.3}$$

We remark that Definition 3.1 is known to be equivalently definable via either *network flows* [Bai96] or *capacities* [Des99, Seg06]. We will not use this equivalence and point the interested reader to [Bai96, DLT08] and [Zha09, Lemma 4.2.1].

For certain proofs, we will rely on simple properties of $\mathcal{L}(R)$ to hold. We introduce these properties here:

Lemma 3.2. *Let S_1, S_2 and S_3 be sets and let $R, R' \subseteq S_1 \times S_2$ and $R'' \subseteq S_2 \times S_3$ be relations over these sets. The following statements hold:*

(i) *If R is left or right-total then so is $\mathcal{L}(R)$.*

(ii) *If R is left or right-unique then so is $\mathcal{L}(R)$.*

(iii) *$\mathcal{L}(R^{-1})$ equals $\mathcal{L}(R)^{-1}$.*

(iv) *$R \subseteq R'$ implies $\mathcal{L}(R) \subseteq \mathcal{L}(R')$.*

(v) *$\mathcal{L}(R'') \circ \mathcal{L}(R)$ is contained in $\mathcal{L}(R'' \circ R)$.*

(vi) *Suppose I is a countable index set and $\{\lambda_i^1\}_{i \in I}$ and $\{\lambda_i^2\}_{i \in I}$ are families of distributions in $\mathbb{D}S_1$ and $\mathbb{D}S_2$, respectively, with $\langle \lambda_i^1, \lambda_i^2 \rangle \in \mathcal{L}(R)$ for each $i \in I$, then for every family of weights $\{w_i\}_{i \in I}$, we have that $\langle \sum_{i \in I} w_i \cdot \lambda_i^1, \sum_{i \in I} w_i \cdot \lambda_i^2 \rangle \in \mathcal{L}(R)$, also.*

Proof. See Section A.1.1 on page 207. □

3.2.2 Probability Theory

We now introduce some basic concepts from probability and measure theory. We refer the interested reader to [Bil86] for details. We will discuss sigma algebras, probability

measures and random variables. We start by defining sigma algebras:

Definition 3.3 (Sigma algebra). *Let Ω be a set. A* sigma algebra *$\mathscr{F}$ on Ω is a non-empty* set of subsets *of Ω satisfying the following conditions:*

(i) $\Omega \in \mathscr{F}$,

(ii) $F \in \mathscr{F}$ *implies* $\Omega \setminus F \in \mathscr{F}$ *and*

(iii) If $\{F_i\}_{i \in I}$ *is a countable family of sets in* $\mathscr{F}$, *then* $(\cup_{i \in I} F_i) \in \mathscr{F}$, *also.*

The set Ω is called a *sample space*. A sigma algebra $\mathscr{F}$ on Ω defines which sets of samples in Ω are *measurable*. To each measurable set of samples (or each "*event*"), we assign a probability. A *probability measure* assigns a probability to each event, $F \in \mathscr{F}$, as follows:

Definition 3.4 (Probability measure). *Let Ω be a sample space and let $\mathscr{F}$ be a sigma algebra on Ω. A* probability measure *on $\langle \Omega, \mathscr{F} \rangle$ is a function, $\mathbf{Pr} : \mathscr{F} \to [0, 1]$, such that all of the following conditions hold:*

(i) $\mathbf{Pr}(\emptyset) = 0$,

(ii) $\mathbf{Pr}(\Omega) = 1$ *and*

(iii) If $\{F_i\}_{i \in I}$ *is a countable family of pairwise disjoint sets in* $\mathscr{F}$, *then* $\mathbf{Pr}(\cup_{i \in I} F_i) = \sum_{i \in I} \mathbf{Pr}(F_i)$.

A tuple $\langle \Omega, \mathscr{F}, \mathbf{Pr} \rangle$ is called a *probability space* if Ω is a sample space, $\mathscr{F}$ is a sigma algebra on Ω, and $\mathbf{Pr}$ is a probability measure on $\langle \Omega, \mathscr{F} \rangle$. With a probability space we can measure the probability of each $F \in \mathscr{F}$ by taking the probability $\mathbf{Pr}(F)$.

Finally we introduce *random variables*:

Definition 3.5 (Random variable). *Let $\langle \Omega, \mathscr{F}, \mathbf{Pr} \rangle$ be probability space. A* random variable *is a function $X : \Omega \to [0, \infty[$. The* expected value *of X in $\langle \Omega, \mathscr{F}, \mathbf{Pr} \rangle$, denoted $\mathbb{E}(X)$, is defined by the integral $\int_\Omega X d\mathbf{Pr}$.*

For more details on expectations and random variables we refer to [Bil86]. For simplicity we ignore issues concerning measurability and convergence — all random variables

we consider are trivially measurable and have converging expectations.

3.2.3 Lattices, Galois Connections & Fixpoints

In this section we will introduce some standard definitions, including *complete lattices*, *Galois connections* and *fixpoints*. For more details we refer to [DP02, NNH05].

Definition 3.6 (Complete lattice). *A* lattice *is a tuple* $\langle S, \leq \rangle$ *where* S *is a non-empty set and* $\leq \subseteq S \times S$ *is a partial order on* S*. We say a lattice is* complete *if every subset* $S' \subseteq S$ *has a* supremum, $\sup S'$*, in* S *as well as an* infimum, $\inf S'$*, in* S*.*

An example of a complete lattice is $\langle \mathbb{B}, \leq \rangle$ with $\leq = \{\langle \mathtt{ff}, \mathtt{ff} \rangle, \langle \mathtt{ff}, \mathtt{tt} \rangle, \langle \mathtt{tt}, \mathtt{tt} \rangle\}$. Other examples of complete lattices include closed intervals over the extended reals, such as $\langle [0,1], \leq \rangle$ and $\langle [0,\infty], \leq \rangle$, using the standard order on $\mathbb{R}$. We introduce two ways to *construct* complete lattices (see Appendix A.2 in [NNH05] for details and proofs).

Definition 3.7. *If* S *is a set and* $\langle X, \leq \rangle$ *is a complete lattice then we write* $\langle S \to X, \leq \rangle$ *for the complete lattice comprising all functions from* S *to* X *such that* $f \leq g$ *for two functions* $f, g : S \to X$ *if and only if* $f(s) \leq g(s)$ *for all* $s \in S$*.*

We will use, say, $\langle S \to [0,1], \leq \rangle$, to order mappings from states to probabilities. We also introduce the following lattice construction:

Definition 3.8. *Let* $\langle S, \leq \rangle$ *be a complete lattice, we will write* $\langle S \times S, \leq \rangle$ *to denote the complete lattice for which* $\langle l, u \rangle \leq \langle l', u' \rangle$ *if and only if* $l \leq l'$ *and* $u' \leq u$*.*

The intuition is that lattice elements, $\langle l, u \rangle \in S \times S$, are essentially intervals on S. However, these intervals are not necessarily consistent — the lower bound may be strictly greater than the upper bound. Given a lattice $\langle S \times S, \leq \rangle$ we write $\mathrm{LB}, \mathrm{UB} : S \times S \to S$ for the projection functions which, for every $\langle l, u \rangle \in S \times S$, $\mathrm{LB}(\langle l, u \rangle)$ and $\mathrm{UB}(\langle l, u \rangle)$ yield the lower bound, l, and the upper bound, u, respectively.

We remark that, if we apply Definition 3.8 to Booleans, i.e. if we consider the lattice $\langle \mathbb{B} \times \mathbb{B}, \leq \rangle$, then we end up with a lattice over Belnap values [Bel77], i.e.

- $\langle \mathtt{tt}, \mathtt{tt} \rangle$ equates to *definitely* $\mathtt{tt}$,
- $\langle \mathtt{ff}, \mathtt{ff} \rangle$ equates to *definitely* $\mathtt{ff}$,
- $\langle \mathtt{ff}, \mathtt{tt} \rangle$ equates to *either* $\mathtt{tt}$ or $\mathtt{ff}$ ("don't know") and
- $\langle \mathtt{tt}, \mathtt{ff} \rangle$ equates to *both* $\mathtt{tt}$ and $\mathtt{ff}$ ("inconsistent").

These Belnap values are frequently used for representing the validity of qualitative properties when they are evaluated on *abstract* models. In the same spirit we will use, say, $\langle [0,1] \times [0,1], \leq \rangle$ and $\langle [0,\infty] \times [0,\infty], \leq \rangle$ to describe the value of *quantitative* properties evaluated on abstract models.

We now show how we can relate two complete lattices:

Definition 3.9 (Galois connection). *Let $\langle A, \sqsubseteq \rangle$ and $\langle B, \preceq \rangle$ be complete lattices and let $\mathrm{L} : A \to B$ and $\mathrm{U} : B \to A$ be monotone functions. We call $\langle A, \sqsubseteq \rangle \underset{\mathrm{L}}{\overset{\mathrm{U}}{\leftrightarrows}} \langle B, \preceq \rangle$ a* Galois connection *if and only if $a \sqsubseteq \mathrm{U}(\mathrm{L}(a))$ and $\mathrm{L}(\mathrm{U}(b)) \preceq b$ for every $a \in A$ and $b \in B$.*

In a Galois connection, $\langle A, \sqsubseteq \rangle \underset{\mathrm{L}}{\overset{\mathrm{U}}{\leftrightarrows}} \langle B, \preceq \rangle$, we call L and U the *lower* and *upper adjunct*, respectively. For some proofs we need some very specific Galois connections. The following definition is a slight generalisation of the Galois connection found in [WZ10]:

Lemma 3.10. *Let S and $\hat{S}$ be sets, let $\langle L, \leq \rangle$ be a complete lattice and let $\mathscr{R} \subseteq \hat{S} \times S$ be a relation. Moreover, let $\alpha^+ : (S \to L) \to (\hat{S} \to L)$ and $\gamma^+ : (\hat{S} \to L) \to (S \to L)$ be functions defined, for every $v : S \to L$, $\hat{v} : \hat{S} \to L$, $s \in S$ and $\hat{s} \in \hat{S}$, as*

$$\alpha^+(v)(\hat{s}) = \sup\{v(s) \mid s \in \mathscr{R}(\hat{s})\} \quad \text{and} \quad \gamma^+(\hat{v})(s) = \inf\{\hat{v}(\hat{s}) \mid \hat{s} \in \mathscr{R}^{-1}(s)\} .$$

We have that $\langle S \to L, \leq \rangle \underset{\alpha^+}{\overset{\gamma^+}{\leftrightarrows}} \langle \hat{S} \to L, \leq \rangle$ is a Galois connection.

Proof. See Section A.1.2 on page 210. □

We will later use this lemma where S and $\hat{S}$ are state spaces and L is either $[0,1]$ or $[0,\infty]$. We will also need the dual of Lemma 3.10:

Corollary 3.11. *Let S and $\hat{S}$ be sets, let $\langle L, \leq \rangle$ be a complete lattice and let $\mathscr{R} \subseteq \hat{S} \times S$ be a relation. Moreover, let $\alpha^- : (S \to L) \to (\hat{S} \to L)$ and $\gamma^- : (\hat{S} \to L) \to (S \to L)$ be*

functions defined, for every $v : S \to L$, $\hat{v} : \hat{S} \to L$, $s \in S$ *and* $\hat{s} \in \hat{S}$, *as*

$$\alpha^-(v)(\hat{s}) = \inf\{v(s) \mid s \in \mathscr{R}(\hat{s})\} \quad \textit{and} \quad \gamma^-(\hat{v})(s) = \sup\{\hat{v}(\hat{s}) \mid \hat{s} \in \mathscr{R}^{-1}(s)\} .$$

We have that $\langle \hat{S} \to L, \leq \rangle \underset{\gamma^-}{\overset{\alpha^-}{\leftrightarrows}} \langle S \to L, \leq \rangle$ *is a Galois connection.*

Proof. Application of Lemma 3.10 on $\mathscr{R}^{-1}$ (with the lattices interchanged). □

Let $\langle A, \sqsubseteq \rangle$ be a complete lattice and let $f : A \to A$ a function on A. We call $a \in A$ a *fixpoint* of f if and only if $f(a) = a$. When f is monotone, using Knaster-Karski's fixpoint theorem, there is a *least* fixpoint of f in $\langle A, \sqsubseteq \rangle$, denoted $\mathrm{LFP}(f)$ [Tar55]. Our main reason for introducing Galois connections is that, under some conditions, Galois connections preserve least fixpoints:

Lemma 3.12 (Fixpoint preservation). *Let* $\langle A, \sqsubseteq \rangle \underset{\mathrm{L}}{\overset{\mathrm{U}}{\leftrightarrows}} \langle B, \preceq \rangle$ *be a Galois connection and let* $f : A \to A$ *and* $g : B \to B$ *be monotone functions such that* $\mathrm{L} \circ f \circ \mathrm{U} \sqsubseteq g$, *then:*

$$\mathrm{LFP}(f) \sqsubseteq \mathrm{U}(\mathrm{LFP}(g)) \quad \textit{and} \quad \mathrm{L}(\mathrm{LFP}(f)) \preceq \mathrm{LFP}(g)$$

Proof. See, e.g., [NNH05, Lemma 4.42]. □

We remark that this fixpoint preservation result is at the core of abstract interpretation (see, e.g., [CC77]). Akin to [WZ10], we will use this fixpoint preservation to prove the soundness of an abstraction framework.

3.3 Markov Decision Processes

In this section we introduce Markov Decision Processes (MDPs) — a mathematical model we will use to describe the semantics of probabilistic programs. We first define MDPs in Section 3.3.1. Then, in Section 3.3.2, we define some qualitative and quantitative properties for systems with MDP semantics. Finally, in Section 3.20, we define a notion of simulation on MDPs.

3.3.1 Definition of Markov Decision Processes

A mathematical formalism in which we capture the semantics of *probabilistic* software clearly needs to be able to model *probabilistic* behaviours. However, we also argue that the ability to deal with *non-determinism* is essential. With non-determinism, we can model programs statements whose precise semantics cannot be directly determined from the source code alone, and may depend on, say, the compiler. For example, if a local variable is left uninitialised, then we cannot be sure of its value and, in our formal model, we typically assign a value to it non-deterministically. Non-determinism is also a very useful tool for modelling calls to library functions for which the source code is unavailable.

In this section we introduce Markov Decision Processes, which have a natural ability to model systems that exhibit both *non-deterministic* and *probabilistic* behaviour.

Definition 3.13 (Markov decision process). *A* Markov decision process *(MDP) M is a tuple, $\langle S, I, T, L, R\rangle$, where*

- *S is a countable set of* states*,*
- *$I \subseteq S$ is a non-empty set of* initial states*,*
- *$T : S \to \overline{\mathbb{P}}\mathbb{D}S$ is a* transition function*,*
- *$L : S \times AP \to \mathbb{B}$ is a* propositional labelling function*, and*
- *$R : S \to [0, \infty[$ is a* cost labelling function*.*

The transition function, T, maps *states* to *sets* of *distributions* over states. We require that the set $T(s)$ is non-empty and countable for every $s \in S$. We will also assume that, for every $s \in S$, the distributions $\lambda \in T(s)$ yield a *rational* probability $\lambda(s') \in [0, 1] \cap \mathbb{Q}$ for every $s' \in S$. We remark that our MDPs do not have action labels. The propositional labelling function, L, assigns to all states $s \in S$ and atomic propositions $a \in$ AP a Boolean value $L(s, a)$ indicating the validity of the proposition a in state s. The cost labelling function, R, assigns a non-negative, real-valued *cost*, $R(s)$, to each state $s \in S$.

A *transition* of M is a tuple $\langle s, \lambda, s'\rangle \in S \times \mathbb{D}S \times S$ and comprises a *source state*, s, a *distribution*, λ, and a *target state*, s', such that λ is an element of $T(s)$ and s has a non-zero probability in λ. The choice of a distribution λ from the set $T(s)$ is a *non-*

deterministic choice. In contrast, the choice of a successor state, s', given a distribution, λ, is *probabilistic* and occurs with probability $\lambda(s')$. We will usually write $s \xrightarrow{\lambda} s'$ to denote a transition $\langle s, \lambda, s' \rangle$ and we will sometimes write $s \to s'$ to denote $s \xrightarrow{[s']} s'$ — i.e. when λ is the point distribution, $[s']$.

A *path* describes an execution of an MDP. Formally, a path of M is a finite sequence in $S \times (\mathbb{D}S \times S)^*$ or an infinite sequence in $S \times (\mathbb{D}S \times S)^\omega$ comprising a finite or infinite number of transitions. We will write $\mathit{FinPath}_M$ and $\mathit{InfPath}_M$ to denote the set of all finite and infinite paths of M, respectively. Given a set of paths and a state $s \in S$ or a set of states $S' \subseteq S$, we add the superscript "s" or "S" to the path set to denote a restriction to paths that originate from s or S', respectively. For example, $\mathit{InfPath}^I_M$ is the set of all infinite paths of M, starting from M's initial states.

Given a finite path, $\pi \in \mathit{FinPath}_M$, we denote by $\textsc{Last}(\pi)$ the *last* state of π and by $|\pi|$ the number of transitions in π. For an arbitrary, potentially infinite path:

$$\pi = s_0 \xrightarrow{\lambda_0} s_1 \xrightarrow{\lambda_1} s_2 \xrightarrow{\lambda_2} \dots \ ,$$

and $i \in \mathbb{N}$ we write π^i to denote the $(i{+}1)$-th state of π (i.e. s_i), and we write $\textsc{Trans}(\pi, i)$ to denote the $(i{+}1)$-th transition of π (i.e. $s_i \xrightarrow{\lambda_i} s_{i+1}$). Moreover, we write $\textsc{Pref}(\pi, i)$ to denote the prefix of π of length i.

Paths effectively resolve both probabilistic and non-deterministic choice. To reason about probabilistic properties, however, we need to consider what happens to M when only non-deterministic choice is resolved. To this end, we introduce *strategies*.[1] Formally, a *strategy* of M is a function $\sigma : \mathit{FinPath}_M \to \mathbb{D}\mathbb{D}S$ which, for every *finite* path $\pi \in \mathit{FinPath}_M$, resolves the non-deterministic choice in $\textsc{Last}(\pi)$ by providing a probability distribution $\sigma(\pi) \in \mathbb{D}(T(\textsc{Last}(\pi)))$ over the available distributions in $T(\textsc{Last}(\pi))$. A strategy is *pure* if it yields a *point distribution* for all paths. We write Strat_M and $\mathit{PureStrat}_M$ to denote the set of *all* strategies and all *pure* strategies of M, respectively.

A path, π, is *consistent* with a strategy σ when for every $i < |\pi|$ the transition $\textsc{Trans}(\pi, i) = s_i \xrightarrow{\lambda_i} s_{i+1}$ is such that the probability $\sigma(\textsc{Pref}(\pi, i))(\lambda_i)$ is positive. For

[1] Elsewhere, strategies are also often called "policies", "adversaries" or "schedulers".

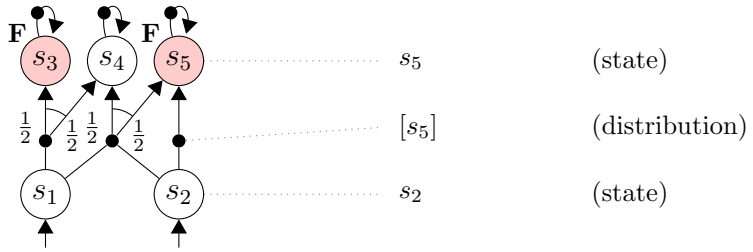


FIGURE 3.1: An MDP M with two target states, s_3 and s_5.

a strategy $\sigma \in \mathit{Strat}_M$, we add the subscript "$\sigma$" to a set of paths to restrict this set to those paths that are consistent with σ. For example, $\mathit{FinPath}_{M,\sigma}$ denotes the set of finite paths of M that are consistent with σ.

Finally, M is said to be *finite* if both its state space, S, is finite and $|T(s)|$ is finite for every $s \in S$. Given two MDPs M and M' we write $M \uplus M'$ for their disjoint union, defined in the standard way. We call M *non-probabilistic* if $T(s)$ contains only point distributions for every $s \in S$. We use non-probabilistic MDPs to model non-probabilistic systems. We do this to avoid notational overhead. We denote with MDP the class of *all* MDPs.

Depicting MDPs In figures, we mark initial states of M with a small incoming arrow. A state is depicted with a big, unfilled circle whereas a distribution is depicted as a small, filled circle. There is an arrow from a distribution $\lambda \in \mathbb{D}S$ to every state $s \in \textsc{Supp}(\lambda)$ labelled with the probability $\lambda(s)$ (we omit "1"). There is also a line from a state $s \in S$ to every distribution in $T(s)$. Finally, if for some state $s \in S$ and some atomic proposition $a \in \mathrm{AP}$ we have $L(s, a)$ then we depict a label "a" next to s. If a is the special atomic proposition $\mathbf{F} \in \mathrm{AP}$, then we also highlight the state. We do not depict costs in figures.

We illustrate the definition of MDPs with an example:

Example 3.14. *Consider an MDP* $M = \langle S, I, T, L, R \rangle$ *with*

- $S = \{s_1, s_2, s_3, s_4, s_5\}$,
- $I = \{s_1, s_2\}$,

– *T is such that, for every $s \in S$, we have*

$$T(s) = \begin{cases} \{\frac{1}{2}[s_3] + \frac{1}{2}[s_4], \frac{1}{2}[s_4] + \frac{1}{2}[s_5]\} & \text{if } s = s_1 \ , \\ \{\frac{1}{2}[s_4] + \frac{1}{2}[s_5], [s_5]\} & \text{if } s = s_2 \ , \text{ and} \\ \{[s]\} & \text{otherwise} \ . \end{cases}$$

– *L is such that, for every $s \in S$ and $a \in AP$, we have*

$$L(s,a) = \begin{cases} \texttt{tt} & \text{if } s \in \{s_3, s_5\} \text{ and } a = \mathbf{F} \ , \text{ and} \\ \texttt{ff} & \text{otherwise} \ . \end{cases}$$

– *$R(s) = 0$ for all $s \in S$.*

We depict M in Figure 3.1. Recall that $\{[s_4]\}$ denotes a singleton set comprising a point distribution on s_4.

3.3.2 Properties of Markov Decision Processes

In this section, we will define some qualitative and quantitative properties of MDPs. We formalise these properties as functions from MDPs to values. To do this we fix, say, the atomic proposition to label target states in the definition of the property. However, our presentation can easily be extended to include properties parameterised by, say, an atomic proposition or a temporal logic formula.

Non-probabilistic reachability We first consider *qualitative* properties $Reach^-, Reach^+ : \text{MDP} \to \mathbb{B}$ which, for a given MDP M, focus on the ability to reach a set of states in M:

Definition 3.15 (Non-probabilistic reachability). *Let $Reach^-, Reach^+ : \text{MDP} \to \mathbb{B}$ be the qualitative properties which, for every MDP $M = \langle S, I, T, L, R \rangle$, are defined as*

$$Reach^-(M) = \forall \pi \in InfPath_M^I : \exists i \in \mathbb{N} : L(\pi^i, \mathbf{F}) \ ,$$
$$Reach^+(M) = \exists \pi \in InfPath_M^I : \exists i \in \mathbb{N} : L(\pi^i, \mathbf{F}) \ .$$

For Reach^{-}, the proposition $\mathbf{F} \in \mathrm{AP}$ typically marks a set of "good" states such that, when $\mathit{Reach}^{-}(M)$ is true, we can conclude that, in M, we will always eventually reach a good state. In contrast, for Reach^{+}, the proposition $\mathbf{F}$ often marks a set of "bad" states such that, if $\neg\mathit{Reach}^{+}(M)$, then we can conclude no bad states are reachable or, equivalently, that M is *safe*. If M is not safe, then this is witnessed by a *counter-example* — a finite path $s_0 \xrightarrow{\lambda_0} \ldots \xrightarrow{\lambda_{n-1}} s_n$ such that $s_0 \in I$ and $L(s_n, \mathbf{F})$. The property Reach^{+} can be verified with most software model checkers. We will call Reach^{-} and Reach^{+} the "*non-probabilistic liveness property*" and "*non-probabilistic safety property*", respectively.

In addition to non-probabilistic reachability properties, for MDPs we are typically interested in characterising the *probability* of reaching states satisfying $\mathbf{F}$. We therefore now consider the probabilistic analogues of non-probabilistic reachability properties.

Probabilistic reachability Next, we will introduce quantitative properties on MDPs. That is, we will define *probabilistic* reachability properties Prob^{-}, $\mathit{Prob}^{+} : \mathrm{MDP} \to [0, 1]$. Probabilities can only be defined in an MDP $M = \langle S, I, T, L, R \rangle$ once all the non-determinism in M is resolved by a strategy. Informally, under a strategy $\sigma \in \mathit{Strat}_M$, the probability of a finite path of

$$\pi = s_0 \xrightarrow{\lambda_0} \ldots \xrightarrow{\lambda_n} s_{n+1} \in \mathit{FinPath}_{M,\sigma}$$

is defined as the product

$$\textsc{Prob}_{M,\sigma}(\pi) = \prod_{i=0}^{|\pi|-1} \sigma(\textsc{Pref}(\pi, i))(\lambda_i) \cdot \lambda_i(s_{i+1}) \ .$$

That is, for every transition $s_i \xrightarrow{\lambda_i} s_{i+1}$ we take into account the probability of choosing to transition from s_i to λ_i under the strategy σ and the probability of transitioning from λ_i to s_{i+1}. For pure strategies $\sigma \in \mathit{PureStrat}_M$, the term $\sigma(\textsc{Pref}(\pi, i))(\lambda_i)$ is always trivially 1. We sometimes write $\textsc{Prob}_M(\pi)$ instead of $\textsc{Prob}_{M,\sigma}(\pi)$ when it is clear we are considering pure strategies.

For a fixed initial state $s \in I$ and strategy $\sigma \in \mathit{Strat}_M$ we define for every *finite* path

$\pi \in \mathit{FinPath}^s_{M,\sigma}$ the *cylinder set* $\textsc{Cyl}^s_{M,\sigma}(\pi) \subseteq \mathit{InfPath}^s_{M,\sigma}$ — the set of all *infinite* paths that start with s, are consistent with σ and that have π as a prefix.

For a fixed initial state $s \in I$ and strategy $\sigma \in \mathit{Strat}_M$ we can construct a probability space $\langle \mathit{InfPath}^s_{M,\sigma}, \mathscr{F}^s_{M,\sigma}, \mathbf{Pr}^s_{M,\sigma} \rangle$ on infinite paths of M with standard methods (see, e.g., [KSK76]). Our probability space is such that the sigma algebra, $\mathscr{F}^s_{M,\sigma}$, contains $\textsc{Cyl}^s_{M,\sigma}(\pi)$ for every *finite* path $\pi \in \mathit{FinPath}^s_{M,\sigma}$ and such that the measure, $\mathbf{Pr}^s_{M,\sigma}$, is such that

$$\mathbf{Pr}^s_{M,\sigma}(\textsc{Cyl}^s_{M,\sigma}(\pi)) = \textsc{Prob}_{M,\sigma}(\pi)$$

for every $\pi \in \mathit{FinPath}^s_{M,\sigma}$.

Through the probability measure, $\mathbf{Pr}^s_{M,\sigma}$, we can quantify the likelihood of certain behaviours of M. That is, we can take *sets* of infinite paths $\Pi \subseteq \mathit{InfPath}^s_{M,\sigma}$ and compute the measure $\mathbf{Pr}^s_{M,\sigma}(\Pi)$. We remark that all sets of paths we consider in this thesis are known to be measurable [Var85].

We are now finally in a position to define a probabilistic safety and liveness property by quantifying over all possible initial states and strategies as follows:

Definition 3.16 (Probabilistic reachability). *Let* $\mathit{Prob}^-, \mathit{Prob}^+ : \mathrm{MDP} \to [0,1]$ *be the quantitative properties which, for every MDP* $M = \langle S, I, T, L, R \rangle$*, are defined as*

$$\mathit{Prob}^-(M) = \inf_{s \in I, \sigma \in \mathit{Strat}_M} \mathbf{Pr}^s_{M,\sigma}(\{\pi \in \mathit{InfPath}^s_{M,\sigma} \mid \exists i \in \mathbb{N} : L(\pi^i, \mathbf{F})\})$$
$$\mathit{Prob}^+(M) = \sup_{s \in I, \sigma \in \mathit{Strat}_M} \mathbf{Pr}^s_{M,\sigma}(\{\pi \in \mathit{InfPath}^s_{M,\sigma} \mid \exists i \in \mathbb{N} : L(\pi^i, \mathbf{F})\}) \,.$$

We will call Prob^- and Prob^+ the "*probabilistic liveness property*" and "*probabilistic safety property*", respectively. The properties $\mathit{Prob}^-(M)$ and $\mathit{Prob}^+(M)$ of M and guarantee that the *probability* of reaching a state satisfying $\mathbf{F}$ in M is at least $\mathit{Prob}^-(M)$ and at most $\mathit{Prob}^+(M)$, respectively. We now illustrate probabilistic reachability properties via an example.

Example 3.17. *Reconsider Example 3.14 and the MDP* M *depicted in Figure 3.1. We have that* $\mathit{Prob}^-(M) = \frac{1}{2}$ *and* $\mathit{Prob}^+(M) = 1$*. To see that* $\mathit{Prob}^+(M) = 1$*, consider the*

initial state s_2 *and a strategy* $\sigma \in \mathit{Strat}_M$ *which, given the path* s_2*, yields the distribution* $[[s_2]]$*. Under this strategy we have that*

$$\mathbf{Pr}^{s_2}_{M,\sigma}(\{\pi \in \mathit{InfPath}^{s_2}_{M,\sigma} \mid \exists i \in \mathbb{N} : L(\pi^i, \mathbf{F})\}) = \mathbf{Pr}^{s_2}_{M,\sigma}(\textsc{Cyl}^{s_2}_{M,\sigma}(s_2 \to s_5)) = 1 \ .$$

As $\mathit{Prob}^+(M)$ *is the supremum of this value over all initial states and strategies of* M *it must be the case that* $\mathit{Prob}^+(M) = 1$*. It is also straightforward to see that there is an initial state and a strategy of* M *that reach the target state with probability* $\frac{1}{2}$*, and that no lower probability is achievable.*

To actually compute probabilistic reachability properties, we can use the model checking algorithms in [CY90, BdA95, CY98]. These algorithms first perform a precomputation using graph-based algorithms and then obtain the required reachability probabilities for each state of the MDP using linear programming. For efficiency reasons, instead of linear programming, probabilistic verification tools frequently compute the probabilities by using an iterative approximation method called *value iteration* [Put94]. In practice, an "explicit" representation of states in the value iteration algorithm does not scale to large MDPs. Therefore, in tools such as PRISM [HKNP06], symbolic data structures are used in the value iteration algorithm instead [Par02].

We remark that the computation of probabilistic safety and liveness properties is at least as hard as the verification of non-probabilistic safety and liveness properties on non-probabilistic MDPs. Informally, for *any* MDP M we can establish the validity of $\mathit{Reach}^+(M)$ by computing $\mathit{Prob}^+(M)$ and checking whether it is greater than 0. Moreover, for a *non-probabilistic* MDP M, we can establish that $\mathit{Reach}^-(M)$ holds by checking whether $\mathit{Prob}^-(M) = 1$.

Cost properties We also introduce quantitative properties, $\mathit{Cost}^-, \mathit{Cost}^+ : \mathrm{MDP} \to [0, \infty]$, which characterise the total expected *cost* incurred in an MDP M. Our definition follows the notion of *expected total cost* in [Put94, Section 5.1]. We use the same probability spaces introduced for the quantitative reachability properties but, instead of measuring probabilities, we consider the expectation of random variables. For a strategy

$\sigma \in \mathit{Strat}_M$ and initial state $s \in I$ let us define for every $i \in \mathbb{N}$ the random variable $X^i_{M,\sigma} : \mathit{InfPath}^s_{M,\sigma} \to [0, \infty[$ as the function defined as

$$X^i_{M,\sigma}(\pi) = R(\pi^i)$$

for every $\pi \in \mathit{InfPath}^s_{M,\sigma}$. Intuitively, the random variable $X^i_{M,\sigma}$ yields the *cost* of the $(i+1)$-th state in R of each path. These random variables are trivially measurable. We define the expected cost properties through these random variables:

Definition 3.18 (Expected total cost). *Let Cost^-, $\mathit{Cost}^+ : \mathrm{MDP} \to [0, \infty]$ be the quantitative properties which, for every MDP $M = \langle S, I, T, L, R \rangle$, are defined as*

$$\mathit{Cost}^-(M) = \inf_{s \in I, \sigma \in \mathit{Strat}_M} \left(\sum_{i \in \mathbb{N}} \mathbb{E}\left(X^i_{M,\sigma}\right) \right) \quad \textit{and}$$

$$\mathit{Cost}^+(M) = \sup_{s \in I, \sigma \in \mathit{Strat}_M} \left(\sum_{i \in \mathbb{N}} \mathbb{E}\left(X^i_{M,\sigma}\right) \right) .$$

Recall from our discussion on countable sums that, formally, the sums in this definition are suprema over all partial sums in $[0, \infty]$. Total expected cost is always defined over infinite paths. Hence, this total cost property will normally only be finite under a strategy σ if, under σ, with probability 1 the MDP ends up remaining in states that yield 0 in R.

We can compute cost properties in a similar fashion to how we compute probabilistic reachability properties. We refer to [Put94, dA99] for further discussion on cost properties. In the remainder of this section we will discuss another characteristics of probabilistic reachability properties.

Reachability probabilities as sums The sets of infinite paths measured in the probabilistic reachability properties of Definition 3.16 are disjoint unions of a countable number of cylinder sets. In practice this means that the probability can be defined by a countable sum. Using such a sum is convenient in many proofs:

Lemma 3.19. *Let $M = \langle S, I, T, L, R \rangle$ be an MDP. Let $\mathbf{F}\text{-}\mathit{FinPath}_M \subseteq \mathit{FinPath}_M$ be precisely the set of finite paths, $\pi \in \mathit{FinPath}_M$, for which $L(\textsc{Last}(\pi), \mathbf{F})$ holds and $\neg L(\pi^i, \mathbf{F})$*

for all $i < |\pi|$. For a fixed initial state $s \in I$ and a fixed strategy $\sigma \in Strat_M$ we have:

$$\mathbf{Pr}^s_{M,\sigma}(\{\pi \in InfPath^s_{M,\sigma} \mid \exists i \in \mathbb{N} : L(\pi^i, \mathbf{F})\}) = \sum_{\pi \in \mathbf{F}\text{-}FinPath^s_{M,\sigma}} \textsc{Prob}_{M,\sigma}(\pi) \ .$$

Proof. See Section A.1.3 on page 211. □

3.3.3 Stuttering Simulations

In this section, we define a notion of simulation on non-probabilistic MDPs based on the divergence blind stuttering equivalence in [DNV95]. The only difference between our definition and [DNV95] is that we do not restrict to equivalence relations.

Definition 3.20 (Stuttering simulation). *Let $M = \langle S, I, T, L, R\rangle$ be a non-probabilistic MDP and let $\mathscr{R} \subseteq S \times S$ be a relation. We call $\mathscr{R}$ a* stuttering simulation *on M if and only if for all $\langle s_0, t_0\rangle \in \mathscr{R}$ the following conditions hold:*

(i) $L(s_0, a) = L(t_0, a)$ *for all* $a \in AP$,

(ii) $\forall s_0 \to s_1 : \exists t_0 \to \ldots \to t_n \in FinPath^s_M : \langle s_1, t_n\rangle \in \mathscr{R}$ *and* $\forall i < n : \langle s_0, t_i\rangle \in \mathscr{R}$.

We remark that, in Definition 3.20, we allow the case where $n = 0$. We will later show that certain stuttering simulations on disjoint unions of MDPs, $M \uplus M'$, preserve $Reach^+$.

3.4 Game-based Abstractions of Markov Decision Processes

In this section, we introduce an abstraction framework for MDPs. This framework uses *two-player stochastic games* to to describe abstractions of MDPs and was originally introduced in [KNP06]. We start, in Section 3.4.1, by defining two-player stochastic games. Section 3.4.2 then introduces quantitative properties for these games. Finally, in Section 3.4.3, we show how to use two-player stochastic games as abstractions of MDPs.

3.4.1 Stochastic Two-player Games

In this thesis, we often refer to *stochastic two-player games* as *games*, meaning the simple stochastic turn-based games on two players as defined in [Sha53, Con92]. In this thesis, we name the players A and player C (instead of, say, player 1 and player 2) to emphasise the roles of the players in a game that is used as an abstraction of an MDP (we will discuss this in Section 3.4.3). In our games, player A transitions, player C transitions and probabilistic transitions strictly alternate.

Definition 3.21 (Stochastic two-player game). *A* stochastic two-player game G *is a tuple,* $\langle S, I, T, L, R\rangle$*, where*

- S *is a countable set of* states*,*
- $I \subseteq S$ *is a non-empty set of* initial states*,*
- $T : S \to \overline{\mathbb{P}}\overline{\mathbb{P}}\mathbb{D}S$ *is a* transition function*,*
- $L : S \times AP \to \mathbb{B} \times \mathbb{B}$ *and*
- $R : S \to [0, \infty[\times [0, \infty[$.

In comparison to MDPs, the transition function, T, has an additional level of choice — T maps states to non-empty, countable *sets* of non-empty, countable *sets* of *probability distributions* over S. Moreover, the propositional labelling function, L, and the cost labelling function, R, take values in $\mathbb{B} \times \mathbb{B}$ and $[0, \infty[\times [0, \infty[$ instead of $\mathbb{B}$ and $[0, \infty[$, respectively. These should be interpreted as values in the lattices $\langle \mathbb{B} \times \mathbb{B}, \leq\rangle$ and $\langle [0, \infty] \times [0, \infty], \leq\rangle$ (see Definition 3.8). This allows the validity of propositions (and the costs incurred in states) to be approximated.

The tuple G implicitly encodes a turn-based game with two players. The state space of this game comprises *player* A *states*, *player* C *states* and *probabilistic states*. Intuitively, player A states correspond to the states of G (i.e. S), player C states are sets of distributions over S (i.e. $\overline{\mathbb{P}}\mathbb{D}S$), and probabilistic states are distributions over S (i.e. $\mathbb{D}S$). That is, probabilistic states are distributions over player A states and player C states are sets of probabilistic states.

The *transitions* in G strictly alternate between *player* A *transitions*, *player* C *transi-*

tions and *probabilistic transitions*. There is a player A transition from a player A state, $s \in S$, to a player C state $\Lambda \in \overline{\mathbb{P}}\mathbb{D}S$, denoted $s \to \Lambda$, if and only if $\Lambda \in T(s)$. Analogously, there is a player C transition from a player C state, $\Lambda \in \overline{\mathbb{P}}\mathbb{D}S$, to a probabilistic state, $\lambda \in \mathbb{D}S$, denoted $\Lambda \to \lambda$, if and only if $\lambda \in \Lambda$. Finally, there is a probabilistic transition from a probabilistic state, $\lambda \in \mathbb{D}S$, to a player A state, $s \in S$, denoted $\lambda \to s$, if and only if $\lambda(s) > 0$.

Analogous to paths in MDPs, a *play* of G is a strictly alternating sequence of player A, player C and probabilistic transitions that starts with a player A state. For a finite play, π, we denote by $|\pi|$ the number of player A transitions in π. We denote by $\mathit{InfPlays}_G$ the set of all infinite plays of G. For all player A states, $s \in S$ and $S' \subseteq S$, we add the superscripts "s" and "S'" to sets of plays to restrict to those plays that start from s and S', respectively. We denote with π^i the $(i{+}1)$-th player A state of G.

In games, the two types of non-determinism are resolved with different types of strategies. A *player A strategy* of G is a mapping from finite plays that end in a player A state to distributions over available player C states. That is, suppose π is a finite play of G that ends with a player A state $s \in S$, then a player A strategy, σ_{A}, will yield a distribution $\sigma_{\mathsf{A}}(\pi) \in \mathbb{D}(T(s))$ on the sets of distributions in $T(s)$. We will call the non-deterministic choice between player C states "*player A non-determinism*". We let $\mathit{Strat}_G^{\mathsf{A}}$ be the set of all player A strategies.

Similarly, a *player C strategy* of G is a mapping from finite plays that end in a player C state to distributions over available probabilistic states in this player C state. That is, suppose π is a finite play of G that ends with a player C state $\Lambda \in \overline{\mathbb{P}}\mathbb{D}S$, then a player C strategy, σ_{C}, will yield a distribution $\sigma_{\mathsf{C}}(\pi) \in \mathbb{D}\Lambda$ on the distributions in Λ. We will call the non-deterministic choice between probabilistic states "*player C non-determinism*". We let $\mathit{Strat}_G^{\mathsf{C}}$ be the set of all player C strategies.

A play of G is consistent with a player A (player C) strategy if all player A (player C) transitions have a non-zero probability with this strategy. We add subscripts "σ_{A}" and "σ_{C}" to sets of paths to restrict to those paths that are consistent with σ_{A} and σ_{C}, respectively.

We call G *finitely branching for player A* if and only if the set $T(s)$ is finite for every

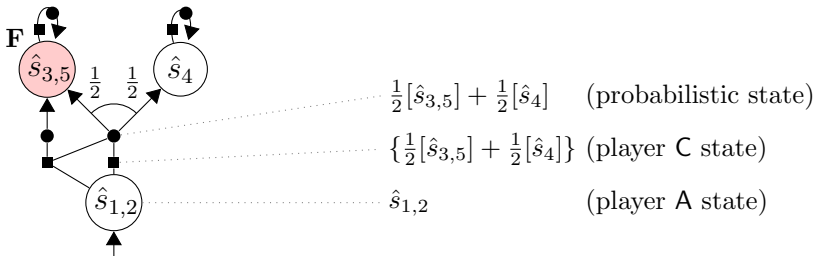


FIGURE 3.2: A two player stochastic game with a single target state, $\hat{s}_{3,5}$.

$s \in S$. For two games, G_1 and G_2, we write $G_1 \uplus G_2$ for the game that is the disjoint union of G_1 and G_2, defined in the standard way. We let GAME be the class of all games.

Depicting Games In figures, we mark initial states of G with a small incoming arrow. A player A state is depicted with a big, unfilled circle, a player C state is depicted with a small, filled square, and a distribution, corresponds to a small, filled circle. Like for MDPs, there is an arrow from a distribution $\lambda \in \mathbb{D}S$ to every state $s \in \text{SUPP}(\lambda)$ labelled with the probability $\lambda(s)$ (we omit "1"). Moreover, there is a line from every player C state, $\Lambda \in \overline{\mathbb{P}}\mathbb{D}S$, to every distribution $\lambda \in \Lambda$ and a line from each player A state, $s \in S$, to every player C state $\Lambda \in T(s)$. We will only depict games for which $L(s,a)$ is either $\langle \mathtt{tt}, \mathtt{tt} \rangle$ or $\langle \mathtt{ff}, \mathtt{ff} \rangle$ for every $s \in S$ and $a \in \text{AP}$. In the former case we write "a" next to the state s. If a is the special atomic proposition $\mathbf{F} \in \text{AP}$, then we also highlight the state. We do not depict costs in figures.

Example 3.22. *Consider the game* $\hat{G} = \langle \hat{S}, \hat{I}, \hat{T}, \hat{L}, \hat{R} \rangle$ *with*

- $\hat{s} = \{\hat{s}_{1,2}, \hat{s}_{3,5}, \hat{s}_4\}$,
- $\hat{I} = \{\hat{s}_{1,2}\}$,
- $\hat{T}$ *is such that, for every* $\hat{s} \in \hat{S}$*, we have*

$$\hat{T}(\hat{s}) = \begin{cases} \{\{\frac{1}{2}[\hat{s}_{3,5}] + \frac{1}{2}[\hat{s}_4]\}, \{\frac{1}{2}[\hat{s}_{3,5}] + \frac{1}{2}[\hat{s}_4], [\hat{s}_{3,5}]\}\} & \text{if } \hat{s} = \hat{s}_{1,2}\text{ , and} \\ \{\{[\hat{s}]\}\} & \text{otherwise .} \end{cases}$$

– $\hat{L}$ *is such that, for every* $\hat{s} \in \hat{S}$ *and* $a \in AP$*, we have*

$$\hat{L}(\hat{s}, a) = \begin{cases} \langle \mathtt{tt}, \mathtt{tt} \rangle & \text{if } \hat{s} \in \{\hat{s}_{3,5}\} \text{ and } a = \mathbf{F} \text{ , and} \\ \langle \mathtt{ff}, \mathtt{ff} \rangle & \text{otherwise .} \end{cases}$$

– $\hat{R}(\hat{s}) = \langle 0, 0 \rangle$ *for all* $\hat{s} \in \hat{s}$.

An example of a play in $FinPlay_{\hat{G}}^{\hat{s}_{1,2}}$ *is*

$$\hat{s}_{1,2} \to \{(\tfrac{1}{2}[\hat{s}_{3,5}] + \tfrac{1}{2}[\hat{s}_4])\} \to (\tfrac{1}{2}[\hat{s}_{3,5}] + \tfrac{1}{2}[\hat{s}_4]) \to \hat{s}_{3,5} \to \{[\hat{s}_{3,5}]\} \to [\hat{s}_{3,5}] \to \hat{s}_{3,5} \text{ .}$$

We have depicted $\hat{G}$ *in Figure 3.2.*

3.4.2 Properties of Games

For games, we will restrict our attention to the *quantitative* properties $Prob^-$, $Prob^+$, $Cost^-$ and $Cost^-$. Whereas for MDPs these properties yield a single real value, for games these properties will yield tuples from $[0,1] \times [0,1]$ and $[0,\infty] \times [0,\infty]$ (see Definition 3.8).

Probabilistic reachability If all non-determinism in a game, G, is resolved via a player A strategy $\sigma_{\mathsf{A}} \in Strat_G^{\mathsf{A}}$ and a player C strategy $\sigma_{\mathsf{C}} \in Strat_G^{\mathsf{C}}$ then, for every initial state s of G, we can construct a probability space $\langle InfPlays^s_{G,\sigma_{\mathsf{A}},\sigma_{\mathsf{C}}}, \mathscr{F}^s_{G,\sigma_{\mathsf{A}},\sigma_{\mathsf{C}}}, \mathbf{Pr}^s_{G,\sigma_{\mathsf{A}},\sigma_{\mathsf{C}}} \rangle$ in the same way we did for MDPs. Unlike for MDPs, we introduce some additional definitions. For every game $G = \langle S, I, T, L, R \rangle$ and state $s \in S$ let us define the abbreviations

$$Prob^{--}(G,s) = \inf_{\sigma_{\mathsf{A}},\sigma_{\mathsf{C}}} \mathbf{Pr}^s_{M,\sigma_{\mathsf{A}},\sigma_{\mathsf{C}}}(\{\pi \in InfPlays^s_{G,\sigma_{\mathsf{A}},\sigma_{\mathsf{C}}} \mid \exists i \in \mathbb{N} : \mathrm{LB}(L(\pi^i, \mathbf{F}))\}) \text{ ,}$$

$$Prob^{+-}(G,s) = \sup_{\sigma_{\mathsf{A}}} \inf_{\sigma_{\mathsf{C}}} \mathbf{Pr}^s_{M,\sigma_{\mathsf{A}},\sigma_{\mathsf{C}}}(\{\pi \in InfPlays^s_{G,\sigma_{\mathsf{A}},\sigma_{\mathsf{C}}} \mid \exists i \in \mathbb{N} : \mathrm{UB}(L(\pi^i, \mathbf{F}))\}) \text{ ,}$$

$$Prob^{-+}(G,s) = \inf_{\sigma_{\mathsf{A}}} \sup_{\sigma_{\mathsf{C}}} \mathbf{Pr}^s_{M,\sigma_{\mathsf{A}},\sigma_{\mathsf{C}}}(\{\pi \in InfPlays^s_{G,\sigma_{\mathsf{A}},\sigma_{\mathsf{C}}} \mid \exists i \in \mathbb{N} : \mathrm{LB}(L(\pi^i, \mathbf{F}))\}) \text{ ,}$$

$$Prob^{++}(G,s) = \sup_{\sigma_{\mathsf{A}},\sigma_{\mathsf{C}}} \mathbf{Pr}^s_{M,\sigma_{\mathsf{A}},\sigma_{\mathsf{C}}}(\{\pi \in InfPlays^s_{G,\sigma_{\mathsf{A}},\sigma_{\mathsf{C}}} \mid \exists i \in \mathbb{N} : \mathrm{UB}(L(\pi^i, \mathbf{F}))\}) \text{ .}$$

Here, σ_{A} and σ_{C} range over player A and player C strategies in $\mathit{Strat}_G^{\mathsf{A}}$ and $\mathit{Strat}_G^{\mathsf{C}}$, respectively. We can now define probabilistic reachability properties on games:

Definition 3.23 (Probabilistic reachability). *Let Prob^-, $\mathit{Prob}^+ : \mathrm{GAME} \to [0,1] \times [0,1]$ be quantitative properties which yield, for every $G = \langle S, I, T, L, R \rangle$, the tuples*

$$\mathit{Prob}^-(G) = \langle \inf_{s \in I} \mathit{Prob}^{--}(G,s),\ \inf_{s \in I} \mathit{Prob}^{+-}(G,s) \rangle \ , \text{ and}$$
$$\mathit{Prob}^+(G) = \langle \sup_{s \in I} \mathit{Prob}^{-+}(G,s),\ \sup_{s \in I} \mathit{Prob}^{++}(G,s) \rangle \ .$$

Probabilistic reachability properties on games yield a value in $[0,1] \times [0,1]$ — a tuple comprising a lower bound and an upper bound — instead of a single probability in $[0,1]$. The lower bounds are obtained by taking infima over all player A strategies and the upper bounds by taking suprema. For a fixed player A strategy we also measure different sets of plays for the lower and upper bound. That is, for the lower bound we measure a play only when $\mathbf{F}$ is *definitely* true in some player A state of this play. The proposition $\mathbf{F}$ is definitely true in a state s if the lower bound $\mathrm{LB}(L(s,\mathbf{F}))$ is true. For the upper bound, we only require that a state *possibly* satisfies $\mathbf{F}$ using the upper bound of L.

We illustrate probabilistic reachability properties on games through an example.

Example 3.24. *Reconsider Example 3.22 and the game $\hat{G}$ depicted in Figure 3.2. We have that $\mathit{Prob}^-(\hat{G}) = \langle \frac{1}{2}, \frac{1}{2} \rangle$ and $\mathit{Prob}^+(\hat{G}) = \langle \frac{1}{2}, 1 \rangle$. Let us explain $\mathit{Prob}^+(\hat{G})$. Suppose player A and player C cooperate. If player A uses a strategy, $\hat{\sigma}_{\mathsf{A}} \in \mathit{Strat}_{\hat{G}}^{\mathsf{A}}$, which, given the play $\hat{s}_{1,2}$, picks the player C state $\{[\hat{s}_{3,5}], \frac{1}{2}[\hat{s}_{3,5}] + \frac{1}{2}[\hat{s}_4]\}$ with probability one, and player C uses a strategy, $\hat{\sigma}_{\mathsf{C}} \in \mathit{Strat}_{\hat{G}}^{\mathsf{C}}$, which, given the play $\hat{s}_{1,2} \to \{[\hat{s}_{3,5}], \frac{1}{2}[\hat{s}_{3,5}] + \frac{1}{2}[\hat{s}_4]\}$ transitions to $[\hat{s}_{3,5}]$ with probability one, then*

$$\mathbf{Pr}^{\hat{s}_{1,2}}_{\hat{G},\hat{\sigma}_{\mathsf{A}},\hat{\sigma}_{\mathsf{C}}}(\{\hat{\pi} \in \mathit{InfPlays}^{\hat{s}_{1,2}}_{\hat{G},\hat{\sigma}_{\mathsf{A}},\hat{\sigma}_{\mathsf{C}}} \mid \exists i \in \mathbb{N} : \mathrm{UB}(\hat{L}(\hat{\pi}^i, \mathbf{F}))\}) =$$
$$\mathbf{Pr}^{\hat{s}_{1,2}}_{\hat{G},\hat{\sigma}_{\mathsf{A}},\hat{\sigma}_{\mathsf{C}}}(\mathrm{CYL}^{\hat{s}_{1,2}}_{\hat{G},\hat{\sigma}_{\mathsf{A}},\hat{\sigma}_{\mathsf{C}}}(\hat{s}_{1,2} \to \{[\hat{s}_{3,5}], \tfrac{1}{2}[\hat{s}_{3,5}] + \tfrac{1}{2}[\hat{s}_4]\} \to [\hat{s}_{3,5}] \to \hat{s}_{3,5})) = 1 \ .$$

Because the upper bound of $\mathit{Prob}^+(\hat{G})$ is a supremum over all initial states and player A and player C strategies, we have that $\mathrm{UB}(\mathit{Prob}^+(\hat{G})) = 1$. To see that $\mathrm{LB}(\mathit{Prob}^+(\hat{G})) = \frac{1}{2}$

observe that, independent of the strategy player A *followed in* $\hat{s}_{1,2}$*, player* C *can always transition to the distribution* $\frac{1}{2}[\hat{s}_{3,5}] + \frac{1}{2}[\hat{s}_4]$*. This means that the target state,* $\hat{s}_{3,5}$*, can always be reached with probability* $\frac{1}{2}$*.*

Cost properties We will now define cost properties on games. Let $G = \langle S, I, T, L, R\rangle$ be a game. Let $\sigma_{\mathsf{A}} \in \mathit{Strat}_G^{\mathsf{A}}$ be a player A strategy of G, let $\sigma_{\mathsf{C}} \in \mathit{Strat}_G^{\mathsf{C}}$ be a player C strategy of G and let $s \in I$ be an intial state of G. For every $i \in \mathbb{N}$ we define the random variables $X^{-,i}_{G,\sigma_{\mathsf{A}},\sigma_{\mathsf{C}}}, X^{+,i}_{G,\sigma_{\mathsf{A}},\sigma_{\mathsf{C}}} : \mathit{InfPlays}^s_{G,\sigma_{\mathsf{A}},\sigma_{\mathsf{C}}} \to [0,\infty[$ as the functions which yield

$$X^{-,i}_{G,\sigma_{\mathsf{A}},\sigma_{\mathsf{C}}}(\pi) = \mathrm{LB}(R(\pi^i)) \text{ and } X^{+,i}_{G,\sigma_{\mathsf{A}},\sigma_{\mathsf{C}}}(\pi) = \mathrm{UB}(R(\pi^i)) \,,$$

for every $\pi \in \mathit{InfPlays}^s_{G,\sigma_{\mathsf{A}},\sigma_{\mathsf{C}}}$. These random variables yield the lowest possible and highest possible cost value for a play that is consistent with R.

Akin to probabilistic reachability, we introduce some additional definitions for games. For every game $G = \langle S, I, T, L, R\rangle$ and state $s \in S$ let us define the abbreviations

$$\begin{aligned}
\mathit{Cost}^{--}(G,s) &= \inf_{\sigma_{\mathsf{A}},\sigma_{\mathsf{C}}} \left(\sum_{i\in\mathbb{N}} \mathbb{E}\left(X^{-,i}_{G,\sigma_{\mathsf{A}},\sigma_{\mathsf{C}}}\right)\right) \,,\\
\mathit{Cost}^{+-}(G,s) &= \sup_{\sigma_{\mathsf{A}}}\inf_{\sigma_{\mathsf{C}}} \left(\sum_{i\in\mathbb{N}} \mathbb{E}\left(X^{+,i}_{G,\sigma_{\mathsf{A}},\sigma_{\mathsf{C}}}\right)\right) \,,\\
\mathit{Cost}^{-+}(G,s) &= \inf_{\sigma_{\mathsf{A}}}\sup_{\sigma_{\mathsf{C}}} \left(\sum_{i\in\mathbb{N}} \mathbb{E}\left(X^{-,i}_{G,\sigma_{\mathsf{A}},\sigma_{\mathsf{C}}}\right)\right) \,,\\
\mathit{Cost}^{++}(G,s) &= \sup_{\sigma_{\mathsf{A}},\sigma_{\mathsf{C}}} \left(\sum_{i\in\mathbb{N}} \mathbb{E}\left(X^{+,i}_{G,\sigma_{\mathsf{A}},\sigma_{\mathsf{C}}}\right)\right) \,.
\end{aligned}$$

Here, σ_{A} and σ_{C} range over player A and player C strategies in $\mathit{Strat}_G^{\mathsf{A}}$ and $\mathit{Strat}_G^{\mathsf{C}}$, respectively. We are now in a position to define the expected total cost on games:

Definition 3.25 (Expected total cost). *Let* Cost^-*,* $\mathit{Cost}^+ : \mathrm{GAME} \to [0,1] \times [0,1]$

be the qualitative properties which, for every game $G = \langle S, I, T, L, R\rangle$, are defined as

$$Cost^{-}(G) = \langle \inf_{s\in I} Cost^{--}(G,s),\ \inf_{s\in I} Cost^{+-}(G,s)\rangle\ ,\ \textit{and}$$
$$Cost^{+}(G) = \langle \sup_{s\in I} Cost^{-+}(G,s),\ \sup_{s\in I} Cost^{++}(G,s)\rangle\ .$$

We remark that the symbolic value iteration algorithms we use to compute probabilistic reachability and cost properties of MDPs generalise to algorithms we can use to compute these properties for games (see also [Con93]).

Finally, having defined properties on games, we consider an alternative method to characterise properties of games.

Fixpoint characterisations Similarly to, say, [WZ10, BBKO10], we use fixpoints as an alternative way to define properties on games. We will use these fixpoint characterisations in proofs. Consider the functions, $\mathrm{P}_G^{--}, \mathrm{P}_G^{+-}, \mathrm{P}_G^{-+}, \mathrm{P}_G^{++} : (S \to [0,1]) \to (S \to [0,1])$, which are defined, for every $v : S \to [0,1]$ and $s \in S$, as

$$\mathrm{P}_G^{--}(v)(s) = \max\left\{\mathrm{LB}(L(s,\mathbf{F})),\ \inf_{s\to\Lambda\to\lambda}\left(\textstyle\sum_{s'\in\mathrm{SUPP}(\lambda)}\lambda(s')\cdot v(s')\right)\right\}\ ,$$
$$\mathrm{P}_G^{+-}(v)(s) = \max\left\{\mathrm{UB}(L(s,\mathbf{F})),\ \sup_{s\to\Lambda}\inf_{\Lambda\to\lambda}\left(\textstyle\sum_{s'\in\mathrm{SUPP}(\lambda)}\lambda(s')\cdot v(s')\right)\right\}\ ,$$
$$\mathrm{P}_G^{-+}(v)(s) = \max\left\{\mathrm{LB}(L(s,\mathbf{F})),\ \inf_{s\to\Lambda}\sup_{\Lambda\to\lambda}\left(\textstyle\sum_{s'\in\mathrm{SUPP}(\lambda)}\lambda(s')\cdot v(s')\right)\right\}\ ,$$
$$\mathrm{P}_G^{++}(v)(s) = \max\left\{\mathrm{UB}(L(s,\mathbf{F})),\ \sup_{s\to\Lambda\to\lambda}\left(\textstyle\sum_{s'\in\mathrm{SUPP}(\lambda)}\lambda(s')\cdot v(s')\right)\right\}\ .$$

Here, we interpret, say, $\mathrm{LB}(L(s,\mathbf{F}))$ as 1 if it is `tt` and as 0 if it is `ff`. In [BBKO10] it is proven that the least fixpoints of these functions relate to the probabilistic reachability properties that we have defined on games.

Lemma 3.26. *Let $G = \langle S, I, T, L, R\rangle$ be a game and let $s \in S$ be an arbitrary state of*

G. We have:

$$Prob^{--}(G,s) = (\mathrm{LFP}(\mathrm{P}_G^{--}))(s) \quad \textit{and} \quad Prob^{+-}(G,s) = (\mathrm{LFP}(\mathrm{P}_G^{+-}))(s) \quad \textit{and}$$
$$Prob^{-+}(G,s) = (\mathrm{LFP}(\mathrm{P}_G^{-+}))(s) \quad \textit{and} \quad Prob^{++}(G,s) = (\mathrm{LFP}(\mathrm{P}_G^{++}))(s)\ .$$

Proof. See [BBKO10, Proof of Theorem 3.1]. □

We remark that the result in Lemma 3.26 makes no assumptions on G. In particular, games need not be finitely branching to satisfy this lemma.

3.4.3 Game Abstractions

In [KNP06] it was suggested to use stochastic two-player games as a formalism to describe abstractions of MDPs. A game abstraction of an MDP $M = \langle S, I, T, L, R\rangle$ is induced by an abstraction function $\alpha : S \to \hat{S}$ from states of M to abstract states, $\hat{S}$. An equivalent definition can be achieved by partitioning M's state space. We first describe how α can be lifted to an abstraction *function*:

Definition 3.27. *Let $M = \langle S, I, T, L, R\rangle$ be an MDP, let $\hat{S}$ be an abstract state space, and let $\alpha : S \to \hat{S}$ be an abstraction function. We lift the abstraction function α to a function $\alpha_{\mathbb{D}} : \mathbb{D}S \to \mathbb{D}(\alpha(S))$, which maps distributions over S to distributions over α's co-domain, $\alpha(S)$, by letting*

$$\alpha_{\mathbb{D}}(\lambda) = \sum_{s \in \mathrm{SUPP}(\lambda)} \lambda(s) \cdot [\alpha(s)]$$

for each $\lambda \in \mathbb{D}S$.

We sometimes write α to denote $\alpha_{\mathbb{D}}$ when it is unambiguous to do so. We remark that an equivalent definition of Definition 3.27 can be achieved via Definition 3.1. That is, let $R \subseteq \alpha(S) \times S$ be the relation induced by α:

$$R = \{\langle \hat{s}, s\rangle \in \alpha(S) \times S \mid \alpha(s) = \hat{s}\}\ .$$

Because R is both right-total and left-unique, by Lemma 3.2, $\mathcal{L}(R)$ is also right-total and left-unique. This means that for every $\lambda \in \mathbb{D}S$ there is precisely one $\hat{\lambda} \in \mathbb{D}(\alpha(s))$ for which we have $\langle \lambda, \hat{\lambda} \rangle \in \mathcal{L}(R)$. It is easy to verify that this $\hat{\lambda}$ is $\alpha_{\mathbb{D}}(\lambda)$ as defined in Definition 3.27 for every $\lambda \in \mathbb{D}S$.

We are now in a position to define game abstraction:

Definition 3.28 (Game abstraction). *Let $M = \langle S, I, T, L, R \rangle$ be an MDP and $\alpha : S \to \hat{S}$ be an abstraction function. We let $\alpha(M)$ be the game $\langle \hat{S}, \hat{I}, \hat{T}, \hat{L}, \hat{R} \rangle$ with*

- $\hat{S} = \alpha(S)$,
- $\hat{I} = \alpha(I)$,
- $\hat{T}(\hat{s}) = \{\{\alpha(\lambda) \mid \lambda \in T(s)\} \mid s \in \alpha^{-1}(\hat{s})\}$ *for all* $\hat{s} \in \hat{S}$,
- $\hat{L}(\hat{s}, a) = \inf_{s \in \alpha^{-1}(\hat{s})} \langle L(s, a), L(s, a) \rangle$ *for all* $\hat{s} \in \hat{S}$ *and* $a \in AP$, *and*
- $\hat{R}(\hat{s}) = \inf_{s \in \alpha^{-1}(\hat{s})} \langle R(s), R(s) \rangle$ *for all* $\hat{s} \in \hat{S}$.

The state space of $\alpha(M)$ is $\alpha(S)$ — i.e. those elements of $\hat{s} \in \hat{S}$ that correspond to some state in $s \in S$ via α. We do this such that we do not have to define the transitions for states of $\hat{S}$ that do not have corresponding concretisations.

The key idea behind $\alpha(M)$'s transition function, $\hat{T}$, is to separate the non-determinism that arises from abstraction from the non-determinism that occurs in M. We do this by attributing these different types of non-deterministic choice to player A and player C in $\alpha(M)$, respectively.

Informally, the role of player A is to resolve non-determinism that is caused by grouping concrete states — the name "A" is short for "abstraction". That is, in a player A state $\hat{s} \in \alpha(S)$, there is a player C state $\hat{\Lambda} \in \hat{T}(\hat{s})$ that corresponds to $T(s)$ — i.e. $\hat{\Lambda} = \{\alpha(\lambda) \mid \lambda \in T(s)\}$ — for every concrete state $s \in S$ that $\hat{s}$ abstracts. Effectively, player A in $\hat{s}$ picks a state in $\alpha^{-1}(\hat{s})$. In reality, of course, many states $s \in \alpha^{-1}(\hat{s})$ may induce the same player C state in $\hat{T}(\hat{s})$, meaning a player A choice in $\alpha(M)$ corresponds to a *set* of states of M.

The role of player C is to resolve non-determinism in the *concrete* model — "C" is an abbreviation for "concrete".. Let $\hat{\Lambda} = \{\alpha(\lambda) \mid \lambda \in T(s)\}$ be a player C state corresponding

to an MDP state $s \in S$. In $\hat{\Lambda}$, player C has the choice of $\alpha(\lambda) \in \mathbb{D}\hat{S}$ — i.e. an abstracted version of the distribution λ — for every non-deterministic choice $\lambda \in T(s)$ in the concrete model. This means player C resolves the non-determinism in the concrete state $s \in S$. Again, many distributions $\lambda \in T(s)$ may induce the same abstract distribution in $\hat{\Lambda}$, and hence a player C transition of $\hat{G}$ corresponds to a *set* of distributions in $T(s)$.

In the definition of $\hat{L}$ and $\hat{R}$ we take infima in $\langle [0,1] \times [0,1], \leq \rangle$ and $\langle [0,\infty] \times [0,\infty], \leq \rangle$, respectively (see Definition 3.8). Our definition of games does not allow $\hat{R}$ to yield bounds that are infinite, and hence, to ensure that all games obtained with Definition 3.28 are well-defined, when we apply game abstraction we will implicitly assume that there is a bound $r \in \mathbb{R}$ such that $R(s) \leq r$ for all $s \in S$.

Example 3.29. *Consider again the MDP M depicted in Figure 3.1. Consider abstraction function $\alpha : \{s_1, s_2, s_3, s_4, s_5\} \to \{\hat{s}_{1,2}, \hat{s}_{3,5}, \hat{s}_4\}$ with $\alpha(s_1) = \alpha(s_2) = \hat{s}_{1,2}$, $\alpha(s_3) = \alpha(s_5) = \hat{s}_{3,5}$ and $\alpha(s_4) = \hat{s}_4$. Suppose $\alpha(M) = \langle \hat{S}, \hat{I}, \hat{T}, \hat{L}, \hat{R} \rangle$. By definition $\alpha(M)$ we have, say:*

$$\begin{aligned} \hat{T}(\hat{s}_{1,2}) &= \{\{\alpha(\lambda) \mid \lambda \in T(s)\} \mid s \in \{s_1, s_2\}\} \\ &= \{\{\tfrac{1}{2}[s_{3,5}] + \tfrac{1}{2}[s_4]\}, \{\tfrac{1}{2}[s_{3,5}] + \tfrac{1}{2}[s_4], [s_5]\}\} \,. \end{aligned}$$

The game $\alpha(M)$ is depicted in Figure 3.2.

The relation between MDPs and their game abstractions enables us to approximate quantitative properties of MDPs via games:

Theorem 3.30 (Soundness of game abstraction). *Let $M = \langle S, I, T, L, R \rangle$ be an MDP and let $\alpha : S \to S$ be an abstraction function. Suppose the game abstraction $\alpha(M) = \langle \hat{S}, \hat{I}, \hat{T}, \hat{L}, \hat{R} \rangle$ is such that, for every $\hat{s} \in \hat{S}$, we have that $\hat{L}(\hat{s}, \mathbf{F})$ and $\hat{R}(\hat{s})$ are precise, i.e. $\hat{L}(\hat{s}, \mathbf{F}) = \langle b, b \rangle$ and $\hat{R}(\hat{s}) = \langle r, r \rangle$ for some $b \in \mathbb{B}$ and $r \in [0, \infty[$. For all $Prop \in \{Prob^-, Prob^+, Cost^-, Cost^+\}$ we have*

$$Prop(\alpha(M)) \leq \langle Prop(M), Prop(M) \rangle \,,$$

where $\leq$ is the order as described in Definition 3.8.

Proof. Follows from [KNP06, Theorem 12]. See also [KKNP10, Theorem 1]. □

We finalise our discussion on game abstractions with an example.

Example 3.31. *Reconsider the game M depicted in Figure 3.1 and its game abstraction $\alpha(M)$ in Figure 3.2 as discussed in Example 3.29. We have that*

$$Prob^{-}(\alpha(M)) = \langle \tfrac{1}{2}, \tfrac{1}{2} \rangle \leq \langle Prob^{-}(M), Prob^{-}(M) \rangle \quad \text{and}$$
$$Prob^{+}(\alpha(M)) = \langle \tfrac{1}{2}, 1 \rangle \leq \langle Prob^{+}(M), Prob^{+}(M) \rangle \ .$$

By definition of $\leq$ this means that $Prob^{-}(M) \in [\frac{1}{2}, \frac{1}{2}]$ and $Prob^{+}(M) \in [\frac{1}{2}, 1]$. That is, by computing, say, $Prob^{-}(\alpha(M))$ we can approximate $Prob^{+}(M)$.

Chapter 4

Probabilistic Software

In this section we introduce *probabilistic software*. Probabilistic software comprises computer programs that are subject to randomness. We first present the formal definition of *probabilistic programs* in Section 4.1. Then, in Section 4.2, we discuss various examples. Finally, in Section 4.3, we consider *weakest preconditions* for probabilistic programs.

4.1 Probabilistic Programs

We first define the *data space* of probabilistic programs in Section 4.1.1. We then formally define probabilistic programs in Section 4.1.2. Finally, in Section 4.1.3, we define the MDP semantics of these programs.

4.1.1 Variables & Data Space

The data space of a program is induced by a finite set of *variables* Var. Each variable $var \in Var$ of a program has a type $\textsc{Type}(var)$. The type, $\textsc{Type}(var)$, is formally defined as the *set of values* that var can assume. We do not explicitly restrict the types of variables allowed in programs. In particular, we do not restrict to finite types — we do require $\textsc{Type}(var)$ to be countable for every variable $var \in Var$.

In practice, most variables types are ANSI-C primitives such as `bool`, `int` and `float`, which we represent with finite bit-vectors. We also admit proper mathematical types such

as $\mathbb{Q}$, $\mathbb{N}$ and $\mathbb{Z}$ and have direct support for structs, enums, pointers and, notably, finite and infinite arrays. If $var \in Var$ is an infinite array then, for any $val \in \textsc{Type}(var)$ and $i \in \mathbb{N}$, we denote with $val\,[i]$ the $(i{+}1)$-th element of val. We represent the program's heap with an infinite array. Although we can model heaps in this way, we note that our verification techniques are not tailored to deal with programs that use the heap intensively. That is, not all probabilistic programs we can specify can be verified with the verification techniques we present in Chapter 5 and 7. In particular, infinite arrays are not fully supported.

Given a set of variables, Var, we let $\mathscr{U}_{Var}$ denote the *data space* induced by the variable set Var — the set of all type-consistent mappings $u \in \mathscr{U}_{Var}$ from variables $var \in Var$ to values $u(var) \in \textsc{Type}(var)$. For an expression, e, over variables in Var and a data state $u \in \mathscr{U}_{Var}$ we write $u(e)$ to denote the value of e in the data state u. For a data state $u \in \mathscr{U}_{Var}$, a variable $var \in Var$ and a value $val \in \textsc{Type}(var)$, we write $u_{[var \mapsto val]}$ to denote the data state $u' \in \mathscr{U}_{Var}$ for which $u'(var) = val$ and which matches with u on every other variable in Var — i.e. $u(var') = u(var')$ for every other $var' \in Var \setminus \{var\}$.

4.1.2 Definition of Probabilistic Programs

The mathematical model we use to model probabilistic programs is tailored to modelling sequential, non-recursive probabilistic programs. That is, we require that the program is governed by a single control-flow graph. In practice, to obtain a single control-flow graph $\langle \mathscr{L}, \mathscr{E} \rangle$, we need to eliminate any function pointers in the program and inline any function calls. We use existing tools to do this for us [Kro10b].

Definition 4.1 (Probabilistic program). *A probabilistic program is a tuple* $P = \langle \langle \mathscr{L}, \mathscr{E} \rangle, \{\mathscr{L}_{\text{N}}, \mathscr{L}_{\text{P}}, \mathscr{L}_{\text{B}}\}, \ell_i, \mathscr{L}_{\text{T}}, \mathscr{L}_{\text{C}}, Var, Sem \rangle$, *where*

- $\langle \mathscr{L}, \mathscr{E} \rangle$ *is a finite directed* control-flow graph,
- $\{\mathscr{L}_{\text{N}}, \mathscr{L}_{\text{P}}, \mathscr{L}_{\text{B}}\}$ *partitions* $\mathscr{L}$ *into* assignment, probabilistic *and* branching *locations,*
- $\ell_i \in \mathscr{L}$ *is an* initial *location,*
- $\mathscr{L}_{\text{T}}, \mathscr{L}_{\text{C}} \subseteq \mathscr{L}$ *are* target *and* cost *locations,*

– *Var is a finite* set of variables *and*
– $\mathit{Sem} : \mathscr{E} \to (\mathscr{U}_{\mathit{Var}} \to \mathbb{PD}\, \mathscr{U}_{\mathit{Var}})$ *maps control-flow edges to their semantics.*

The control-flow graph $\langle \mathscr{L}, \mathscr{E} \rangle$ comprises a set of locations and an edge relation $\mathscr{E} \subseteq \mathscr{L} \times \mathscr{L}$. We require that $\mathscr{E}$ is left-total — i.e. that every control-flow location has an outgoing edge — and model program termination via self-loops. The location ℓ_i is the entry location of the program and the sets of locations $\mathscr{L}_{\mathrm{T}}$ and $\mathscr{L}_{\mathrm{C}}$ help us define properties of the program and identify locations where the target is reached or a cost is incurred, respectively.

The data space of P is the set $\mathscr{U}_{\mathit{Var}}$, induced by a finite set of variables *Var*. The semantics, *Sem*, comprises a mapping from data states to potentially empty, countable sets of distributions on the data space $\mathscr{U}_{\mathit{Var}}$ for every control-flow edge in $\mathscr{E}$.

The partition $\{\mathscr{L}_{\mathrm{N}}, \mathscr{L}_{\mathrm{P}}, \mathscr{L}_{\mathrm{B}}\}$ helps us define some additional requirements on the control-flow and semantics. Firstly, we assume that every location has a successor in $\mathscr{E}$. We also assume that control-flow branching only occurs in branching program locations — locations in $\mathscr{L} \setminus \mathscr{L}_{\mathrm{B}}$ have at most one successor in $\mathscr{E}$. For such locations $\ell \in \mathscr{L}_{\mathrm{P}} \cup \mathscr{L}_{\mathrm{N}}$ the set $\mathscr{E}(\ell)$ is always a singleton set. Hence, for convenience, we define a successor function $\textsc{Succ} : \mathscr{L}_{\mathrm{P}} \cup \mathscr{L}_{\mathrm{N}} \to \mathscr{L}$ which maps non-branching program locations to their successor in $\mathscr{E}$ — i.e. for all $\ell \in \mathscr{L}_{\mathrm{P}} \cup \mathscr{L}_{\mathrm{N}}$ we have that $\textsc{Succ}_{\mathscr{E}}(\ell)$ is such that $\mathscr{E}(\ell) = \{\textsc{Succ}_{\mathscr{E}}(\ell)\}$. Finally, we make some additional assumptions about *Sem*:

– All *assignment* locations, $\ell \in \mathscr{L}_{\mathrm{N}}$, are *non-probabilistic*. More specifically, for every $u \in \mathscr{U}_{\mathit{Var}}$, the set $\mathit{Sem}(\langle \ell, \textsc{Succ}_{\mathscr{E}}(\ell) \rangle)(u)$ is a non-empty set of point distributions.
– All *probabilistic* locations, $\ell \in \mathscr{L}_{\mathrm{P}}$, are *deterministic*. That is, for every $u \in \mathscr{U}_{\mathit{Var}}$, $\mathit{Sem}(\langle \ell, \textsc{Succ}_{\mathscr{E}}(\ell) \rangle)(u)$ is a set comprising exactly one distribution.
– For all *branching* locations, $\ell \in \mathscr{L}_{\mathrm{B}}$, for every $u \in \mathscr{U}_{\mathit{Var}}$, there is one $\ell' \in \mathscr{E}(\ell)$ with $\mathit{Sem}(\langle \ell, \ell' \rangle)(u) = \{[u]\}$ and $\mathit{Sem}(\langle \ell, \ell'' \rangle)(u) = \emptyset$ for all other $\ell'' \in \mathscr{E}(\ell)$.

Note that we satisfy another assumption by construction: for $\langle \ell, u \rangle \in \mathscr{L} \times \mathscr{U}_{\mathit{Var}}$ there is a *precisely one* successor location $\ell' \in \mathscr{E}(\ell)$ for which $\mathit{Sem}(\langle \ell, \ell' \rangle)(u)$ is non-empty.

Typically, with our definition, many assignments could be modelled with either assign-

ment locations or probabilistic locations. We use assignment locations wherever possible.

A program is called *non-probabilistic* if $\mathscr{L}_{\text{P}}$ is empty. We denote with PROG the class of all probabilistic programs.

4.1.3 MDP Semantics

In this section we define the semantics of probabilistic programs through a function $[\![\cdot]\!] : \text{PROG} \to \text{MDP}$. Due to our definition of probabilistic programs, this function is relatively straightforward to define:

Definition 4.2 (Semantics of probabilistic programs). *Let P be the probabilistic program $\langle\langle \mathscr{L}, \mathscr{E}\rangle, \{\mathscr{L}_{\text{N}}, \mathscr{L}_{\text{P}}, \mathscr{L}_{\text{B}}\}, \ell_i, \mathscr{L}_{\text{T}}, \mathscr{L}_{\text{C}}, Var, Sem\rangle$. We define P's MDP semantics, $[\![P]\!]$, as the MDP $\langle S, I, T, L, R\rangle$ where, for all $\langle \ell, u\rangle \in \mathscr{L} \times \mathscr{U}_{Var}$ and $a \in AP$, we have:*

- $S = \mathscr{L} \times \mathscr{U}_{Var}$,
- $I = \{\ell_i\} \times \mathscr{U}_{Var}$,
- $T(\langle \ell, u\rangle) = \{\textsc{Join}([\ell'], \lambda) \mid \langle \ell, \ell'\rangle \in \mathscr{E},\ \lambda \in Sem(\langle \ell, \ell'\rangle)(u)\}$,
- $L(\langle \ell, u\rangle, a)$ *is* `tt` *if* $(\ell \in \mathscr{L}_{\text{T}})$ *and* $(a = \mathbf{F})$ *and* `ff` *otherwise, and*
- $R(\langle \ell, u\rangle)$ *is* 1 *if* $\ell \in \mathscr{L}_{\text{C}}$ *and* 0 *otherwise.*

The state space of the MDP $[\![P]\!]$ comprises a control component, $\mathscr{L}$, and a data component, $\mathscr{U}_{Var}$. For a state $\langle \ell, u\rangle \in \mathscr{L} \times \mathscr{U}_{Var}$ the distributions available in $T(\langle \ell, u\rangle)$ are obtained by taking the union of the sets of distributions available at every outgoing control-flow edges in $\ell' \in \mathscr{E}(\ell)$. By construction, for every control-flow location and data state there is precisely one outgoing control-flow edge that has a non-empty set of distributions available. Note that we do not use the full power of the propositional labelling function, L. We only use the proposition $\mathbf{F} \in \text{AP}$, which is true in locations $\ell \in \mathscr{L}_{\text{T}}$ and ignore other propositions. Similarly, we only admit costs values of 0 and 1, depending on whether we are in a cost location $\ell \in \mathscr{L}_{\text{C}}$.

Properties Having defined the MDP semantics of probabilistic programs we are now finally in a position to formally state the verification problem we are targeting in this

```
bool c=0;

void main()
{
  while (!c)
  {
    cost();
    c=coin(1,2);
  }
  target();
}
```

FIGURE 4.1: The source code of the probabilistic program described in Example 4.3.

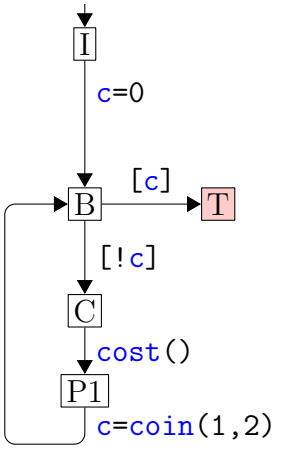


FIGURE 4.2: The annotated control-flow graph of the probabilistic program described in Example 4.3.

thesis. We are interested in quantitative properties of probabilistic programs, P, as defined through their MDP semantics $[\![P]\!]$. That is, we will set out to develop verification techniques which take a probabilistic program $P \in \text{PROG}$ and a property $Prop \in \{Prob^-, Prob^+, Cost^-, Cost^+\}$ and then compute the value $Prop([\![P]\!])$.

4.2 Examples

We will illustrate our definition of probabilistic programs with some examples. To aid presentation, we will not generally introduce probabilistic programs via their mathematical definition. Instead, we use a combination of source code (see, e.g., Figure 4.1) and annotated control-flow graphs (see, e.g., Figure 4.2).

The annotated control-flow graph essentially depicts $\langle \mathscr{L}, \mathscr{E} \rangle$. A small incoming arrow identifies the initial location, ℓ_i. The control-flow we depict deviates slightly from the one we define formally. Firstly, some locations have no outgoing control-flow edges. These are assignment locations $\ell \in \mathscr{L}_{\text{N}}$ in which there is a self-loop $\langle \ell, \ell \rangle \in \mathscr{E}$ that has trivial semantics in Sem — i.e. $Sem(\langle \ell, \ell \rangle)(u) = \{[u]\}$ for every $u \in \mathscr{U}_{Var}$. Secondly, we sometimes omit a control-flow edge from branching locations. Implicitly, there is a self-loop in such branching locations that is labelled with an expression that holds if and only if the expressions on all other branches do not hold.

TYPE	SYNTAX	INTERPRETATION
$\mathscr{L}_{\text{N}}$	var=e_1	Returns the singleton set $\{[u_{[var\mapsto u(e_1)]}]\}$.
	var=*	Returns the set $\{[u_{[var\mapsto val]}] \mid val \in \text{TYPE}(var)\}$.
	var=ndet(e_1)	Returns the set $\{[u_{[var\mapsto n]}] \mid n \in \mathbb{N}, n < u(e_1)\}$ if $u(e_1) > 0$ and $\{[u_{[var\mapsto 0]}]\}$, otherwise.
$\mathscr{L}_{\text{P}}$	var=coin(e_1,e_2)	Returns $\left\{\frac{u(e_1)}{u(e_2)} \cdot [u_{[var\mapsto \texttt{tt}]}] + \left(\frac{u(e_2)-u(e_1)}{u(e_2)}\right) \cdot [u_{[var\mapsto \texttt{ff}]}]\right\}$ if $u(e_2) > 0$ and $0 \leq u(e_1) \leq u(e_2)$ and $\{[u_{[var\mapsto \texttt{ff}]}]\}$ otherwise.
	var=uniform(e_1)	Returns $\left\{\sum_{n=0}^{u(e_1)-1} \frac{1}{u(e_1)} [u_{[var\mapsto n]}]\right\}$ if $u(e_1) > 0$ and $\{[u_{[var\mapsto 0]}]\}$, otherwise.
$\mathscr{L}_{\text{B}}$	[e_1]	Returns the set $\{[u]\}$ if $u(e_1)$ and $\emptyset$ if $\neg u(e_1)$.

FIGURE 4.3: We show some common syntax labels for control-flow edges $\langle \ell, \ell' \rangle \in \mathscr{E}$. We depict the type of the source location, the syntax and the set $Sem(\langle \ell, \ell' \rangle)(u)$ for a given data state $u \in \mathscr{U}_{Var}$.

Our treatment of $\mathscr{L}_{\text{C}}$ and $\mathscr{L}_{\text{T}}$ is not symmetrical. That is, as target locations are often final locations of the program, we do not depict the function call, "`target()`", in figures. Instead, the locations in the set $\mathscr{L}_{\text{T}}$ are highlighted. Locations in $\mathscr{L}_{\text{C}}$ *can* be identified with an outgoing edge labelled with "`cost()`".

The variables of a program are not formally introduced by annotated control-flow graphs but can be inferred from the corresponding source code listings or explanatory text. The semantics and remaining location types are also not introduced formally. However, we do provide an intuitive syntax labelling from which some of this information can be derived. In Figure 4.3 we show how to interpret some common edge labels. We mention that the fractions we use to define the semantics of probabilistic locations should be interpreted as proper fractions and not as C-style integer divisions. We also remark that this figure only gives exemplary definitions. In practice we are not restricted to this set of functions and we also allow, say, assignments where the left-hand side is not a variable. Defining the syntax of ANSI-C programs or the formal semantics of this syntax are non-trivial tasks that fall outside the scope of this thesis. We will assume that, for the most part, the syntax we use in control-flow graphs is self-explanatory.

We now provide various examples of probabilistic programs — both in terms of source code, annotated control-flow graph as well as the formal mathematical definition.

Example 4.3. *Consider the program in Figure 4.1. This program has one, Boolean-valued variable,* `c`*, which is initialised to* 0*. The program contains a loop. In the body of this loop the program first calls* `cost()` *and then assigns* 0 *or* 1 *to* `c` *with a probability of* $\frac{1}{2}$ *each. The function* `target()` *is called once* `c` *becomes 1. The annotated control-flow graph of this program is depicted in Figure 4.2. Formally, the program described here is* $P = \langle\langle \mathscr{L}, \mathscr{E}\rangle, \{\mathscr{L}_{\mathrm{N}}, \mathscr{L}_{\mathrm{P}}, \mathscr{L}_{\mathrm{B}}\}, \ell_i, \mathscr{L}_{\mathrm{T}}, \mathscr{L}_{\mathrm{C}}, \mathit{Var}, \mathit{Sem}\rangle$ *with*

$$\begin{aligned} \mathscr{L} &= \{I, B, C, P1, T\}, \text{ and} \\ \mathscr{E} &= \{\langle I, B\rangle, \langle B, C\rangle, \langle C, P1\rangle, \langle P1, B\rangle, \langle B, T\rangle, \langle T, T\rangle\}\ . \end{aligned}$$

We partition $\mathscr{L}$ *into assignment locations* $\mathscr{L}_{\mathrm{N}} = \{I, C, T\}$*, branching locations* $\mathscr{L}_{\mathrm{B}} = \{B\}$ *and probabilistic locations* $\mathscr{L}_{\mathrm{P}} = \{P1\}$*. The initial location,* ℓ_i*, is* I*, the cost locations are* $\mathscr{L}_{\mathrm{C}} = \{C\}$ *and the target locations are* $\mathscr{L}_{\mathrm{T}} = \{T\}$*. The set of variables, Var, consists of one variable,* `c`*, with* $\mathrm{TYPE}(\mathtt{c}) = \mathbb{B}$*. The semantics, Sem, are defined for every control-flow edge* $e \in \mathscr{E}$ *and every data state* $u \in \mathscr{U}_{\mathit{Var}}$ *as*

$$\mathit{Sem}(e)(u) = \begin{cases} \{[u_{[\mathtt{c}\mapsto\mathtt{ff}]}]\} & \text{if } e{=}\langle I, B\rangle, \\ \{[u]\} & \text{if } e{=}\langle B, C\rangle \wedge \neg u(\mathtt{c}), \text{ or } e{=}\langle B, T\rangle \wedge u(\mathtt{c}), \text{ or } e{=}\langle C, P1\rangle \text{ or } e{=}\langle T, T\rangle, \\ \emptyset & \text{if } e{=}\langle B, C\rangle \wedge u(\mathtt{c}), \text{ or } e{=}\langle B, T\rangle \wedge \neg u(\mathtt{c}), \\ \{\frac{1}{2}[u_{[\mathtt{c}\mapsto\mathtt{ff}]}] + \frac{1}{2}[u_{[\mathtt{c}\mapsto\mathtt{tt}]}]\} & \text{if } e = \langle P1, B\rangle\ . \end{cases}$$

Here, $\{[u]\}$ *and* $\{[u_{[\mathtt{c}\mapsto\mathtt{ff}]}]\}$*, say, denote singleton sets of point distributions on* $\mathscr{U}_{\mathit{Var}}$*. The semantics for* $P1$ *can be viewed as a probabilistic combination of the semantics of two non-probabilistic assignments* `c=1` *and* `c=0`*.*

Example 4.4. *Our next program, depicted in Figure 4.4, is slightly more involved and*

```
bool n, p1, p2;

void main()
{
  n=foo();

  p1=coin(4,10);
  p2=coin(4,10);

  if (n==(p1^p2))
    target();
}
```

FIGURE 4.4: The source code of the probabilistic program described in Example 4.4.

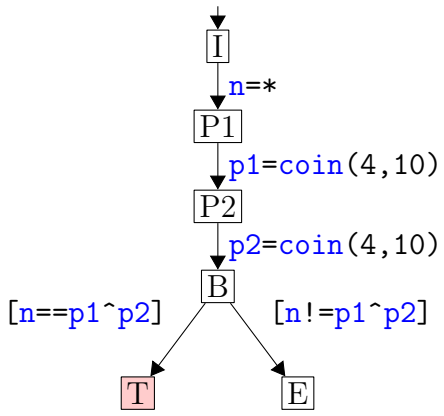


FIGURE 4.5: The annotated control-flow graph of the probabilistic program described in Example 4.4.

comprises three Boolean variables, `n`, `p1` *and* `p2`*. This program first calls a library function,* `foo()`*, and assigns the result to* `n`*. We will model this function call with a nondeterministic choice, written* `n=*`*. Next, the program assigns* 1 *to* `p1` *with probability* $\frac{4}{10}$ *and* 0 *to* `p1` *with probability* $\frac{6}{10}$ *and then repeats this probabilistic assignment for* `p2`*. Finally, the program checks if the exclusive or of* `p1` *and* `p2` *matches the value of* `n`*, and calls* `target()` *if this is the case. The annotated control-flow graph of this program is depicted in Figure 4.5. For this example we have*

$$\mathscr{L} = \{I, P1, P2, B, T, E\}, \text{ and}$$
$$\mathscr{E} = \{\langle I, P1\rangle, \langle P1, P2\rangle, \langle P2, B\rangle, \langle B, T\rangle, \langle B, E\rangle, \langle T, T\rangle, \langle E, E\rangle\} .$$

We partition $\mathscr{L}$ *with assignment locations* $\mathscr{L}_{\text{N}} = \{I, T, E\}$*, branching locations* $\mathscr{L}_{\text{B}} = \{B\}$ *and probabilistic locations* $\mathscr{L}_{\text{P}} = \{P1, P2\}$*. The initial location,* ℓ_i*, is* I*, the cost locations are* $\mathscr{L}_{\text{C}} = \emptyset$ *and the target locations are* $\mathscr{L}_{\text{T}} = \{T\}$*. The set* $Sem(e)(u)$ *is defined for every*

```
bool num_pkts()
{ return ndet(3); }

bool send_to()
{ return coin(1,10); }

void main()
{
 bool f=0;
 int p=num_pkts();

 while (!f&&p!=0)
 {
   f=send_to();
   p--;
 }
 if (f) target();
}
```

FIGURE 4.6: A probabilistic program P for which we want to compute the worst-case probability of failure — i.e. $Prob^{+}([\![P]\!])$.

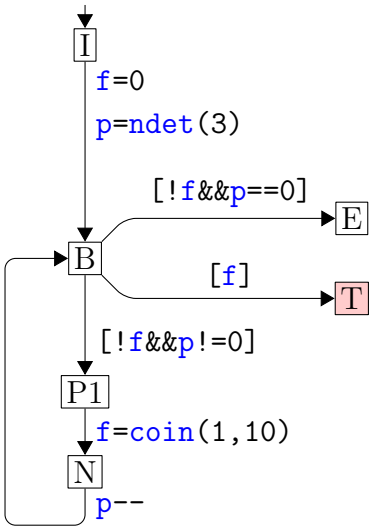


FIGURE 4.7: The control-flow graph of the program in Figure 4.6 with `num_pkts()` and `send_to()` inlined.

control-flow edge $e \in \mathscr{E}$ *and every data state* $u \in \mathscr{U}_{Var}$ *as*

$$
\begin{array}{ll}
\{[u_{[\mathtt{n}\mapsto\mathtt{ff}]}], [u_{[\mathtt{n}\mapsto\mathtt{tt}]}]\} & \textit{if } e{=}\langle I, P1\rangle, \\
\{\frac{6}{10}[u_{[\mathtt{p1}\mapsto\mathtt{ff}]}] + \frac{4}{10}[u_{[\mathtt{p1}\mapsto\mathtt{tt}]}]\} & \textit{if } e{=}\langle P1, P2\rangle, \\
\{\frac{6}{10}[u_{[\mathtt{p2}\mapsto\mathtt{ff}]}] + \frac{4}{10}[u_{[\mathtt{p2}\mapsto\mathtt{tt}]}]\} & \textit{if } e{=}\langle P2, B\rangle, \\
\{[u]\} & \textit{if } e{=}\langle B, T\rangle \wedge (u(\mathtt{n}) \Leftrightarrow (u(\mathtt{p1}) \not\Leftrightarrow u(\mathtt{p2}))), \textit{ or} \\
 & \quad e{=}\langle B, E\rangle \wedge (u(\mathtt{n}) \not\Leftrightarrow (u(\mathtt{p1}) \not\Leftrightarrow u(\mathtt{p2}))), \textit{ or} \\
 & \quad e{=}\langle T, T\rangle \textit{ or } e{=}\langle E, E\rangle, \\
\emptyset & \textit{if } e{=}\langle B, T\rangle \wedge (u(\mathtt{n}) \not\Leftrightarrow (u(\mathtt{p1}) \not\Leftrightarrow u(\mathtt{p2}))), \textit{ or} \\
 & \quad e{=}\langle B, E\rangle \wedge (u(\mathtt{n}) \Leftrightarrow (u(\mathtt{p1}) \not\Leftrightarrow u(\mathtt{p2}))) .
\end{array}
$$

In contrast to the previous example, the assignment location, I*, now has non-deterministic semantics, yielding a set of two point distributions.*

We illustrate MDP-semantics with another example.

Example 4.5. *Let* $P = \langle\langle \mathscr{L}, \mathscr{E}\rangle, \{\mathscr{L}_{\mathrm{N}}, \mathscr{L}_{\mathrm{P}}, \mathscr{L}_{\mathrm{B}}\}, \ell_i, \mathscr{L}_{\mathrm{T}}, \mathscr{L}_{\mathrm{C}}, Var, Sem\rangle$ *be the network*

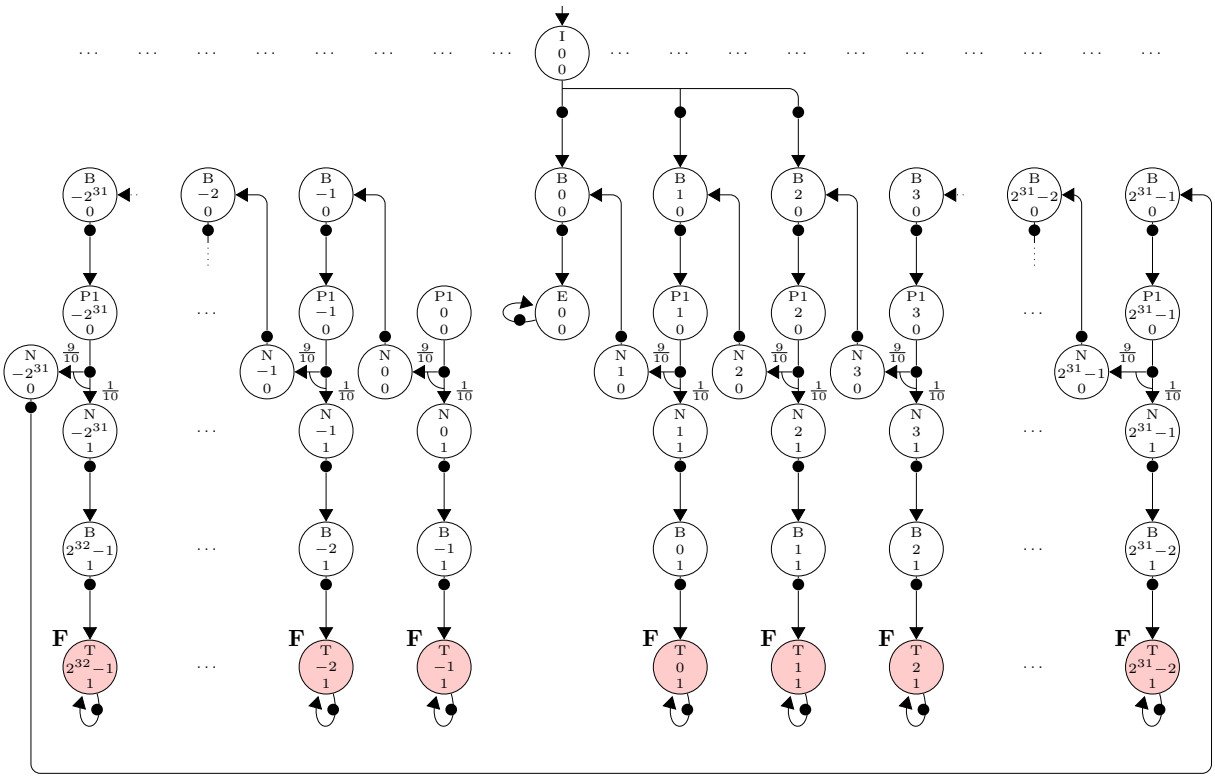


FIGURE 4.8: The MDP-semantics $[\![P]\!]$ of the program depicted in Figure 4.6 and 4.7.

program depicted in Figure 4.6 and 4.7.[1] *We have* $Var = \{\texttt{f}, \texttt{p}\}$ *where* `f` *is a Boolean variable indicating failure and* `p` *is a 32-bit signed integer* `p`*, i.e. an integer in the interval* $[-2^{31}, 2^{31}[$*, storing the number of packets to send. The program first calls* `num_pkts()` *and assigns the resulting value to* `p`*. It will then try to send* `p` *packets of data. To do this, while no failure has occurred and* `p!=0`*, the program calls a function* `send_to()`*, assigns* `send_to()`*'s return value to* `f` *and decrements* `p`*. For the purposes of model checking, the library functions* `num_pkts()` *and* `send_to()` *have been replaced with stubs.*

The annotated control-flow graph of P *is depicted in Fig. 4.7 and the MDP* $[\![P]\!]$ *is depicted in Fig. 4.8. A state* $\langle \ell, u \rangle$ *of* $[\![P]\!]$ *is depicted with three labels: "*ℓ*" (top), "*u(`p`)*" (middle) and "*u(`f`)*" (bottom). Due to space constraints, we omit some states with location* I *and* E*. From the MDP we see that* $Prob^{-}([\![P]\!])$ *and* $Prob^{+}([\![P]\!])$*, i.e. the minimum and maximum probabilities of failure, are* 0 *and* $\frac{19}{100}$*, respectively.*

[1] This program is adapted from [KKNP09] and is due to David Parker.

4.3 Weakest Preconditions

Now we have defined probabilistic programs, we are in a position define *weakest preconditions* [Dij75]. We first informally discuss weakest preconditions in a non-probabilistic setting. Suppose a program comprises variables, Var, and control-flow edges, $\mathscr{E}$. A *predicate* on Var is a Boolean-valued function, $p : \mathscr{U}_{Var} \to \mathbb{B}$. In [Dij75], weakest preconditions are *predicate transformers*, i.e. functions that map predicates to predicates. Informally, for every predicate, $p : \mathscr{U}_{Var} \to \mathbb{B}$, and every control-flow edge, $e \in \mathscr{E}$, the *weakest precondition* of p under e is another predicate, $\mathrm{WP}(e, p) : \mathscr{U}_{Var} \to \mathbb{B}$, such that for every data state $u \in \mathscr{U}_{var}$ we have that $\mathrm{WP}(e, p)(u)$ is true if and only if there is a transition from e for u and p is guaranteed to hold after we take e.

Note that, due to the presence of probabilistic choice, Dijkstra's definition of weakest precondition is not immediately applicable to probabilistic programs. We remark that there exist quantitative adaptations that *can* deal with probabilistic semantics but these are not defined over Boolean-valued predicates (see, e.g., [MM05]). We will use Dijkstra-style weakest preconditions and apply them only under certain restricted conditions:

Definition 4.6 (Weakest precondition). *Let $p : \mathscr{U}_{Var} \to \mathbb{B}$ be a predicate and let $Sem : \mathscr{U}_{Var} \to \mathbb{PD}\mathscr{U}_{Var}$ be a function that yields, for every data state $u \in \mathscr{U}_{Var}$, either the empty set or a singleton set comprising a point distribution. We define the weakest precondition of p under Sem to be the predicate $WP(Sem, p) : \mathscr{U}_{Var} \to \mathbb{B}$ such that, for every data state $u \in \mathscr{U}_{Var}$, we have*

$$WP(Sem, p)(u) = \begin{cases} \texttt{tt} & \text{if } \exists u' \in \mathscr{U}_{Var} : Sem(u) = \{[u']\} \wedge p(u') \ , \\ \texttt{ff} & \text{otherwise} \ . \end{cases}$$

We define weakest preconditions for a function $Sem : \mathscr{U}_{Var} \to \mathscr{U}_{Var}$ similarly.

The conditions in Definition 4.6 mean that, in practice, we can take the weakest precondition of control-flow edges labelled with deterministic assignments (e.g. *var*=e) and edges from branching locations (e.g. [e]), but not probabilistic assignments (e.g. *var*=`coin(1,2)`) or non-deterministic assignments (e.g. *var*=`*`).

In practice, taking the weakest precondition is a cheap, syntactical operation. For a control-flow edge from a branching location, labelled with `[`e`]`, the weakest precondition of a predicate p is the conjunction of p and e. For a control-flow edge from an assignment location, labelled with a deterministic assignment, var`=`e, the weakest precondition of a predicate p is the predicate p with every instance of var in p substituted with e.

Chapter 5

Abstraction Refinement for Probabilistic Software

5.1 Introduction

In this chapter, we will introduce an approach for computing *quantitative* properties (see Section 3.3.2) of *probabilistic* software (see Chapter 4). A central problem in software verification is that, due to the complex nature of software, we typically cannot analyse programs directly via their low-level semantics. In our setting this means that, despite the fact there are efficient algorithms for computing quantitative properties of MDPs (e.g. [CY90, CY98, BdA95, Put94, dA99]), we cannot normally compute quantitative properties of a probabilistic program, P, via the MDP, $[\![P]\!]$.

The inability to verify programs directly via their low-level semantics is not an issue that is specific to *probabilistic* programs. A common approach in non-probabilistic software verification is to reason about properties of programs via finitary *abstractions*. Abstractions are formal mathematical models that yield an approximation of the property under consideration when model checked. This approximation concerns the validity of a qualitative property or, in our quantitative setting, the value of a quantitative property. For a given program, there are usually many abstractions of varying precisions that we can consider. The main challenge in using abstractions is to find an abstraction of the program that is both precise enough to give a good approximation and that is small enough

to model check efficiently. Finding such abstractions *manually* can be very cumbersome. A widely recognised methodology to *automatically* find suitable abstractions is what we call the "*abstraction-refinement paradigm*".

In the abstraction-refinement paradigm, one considers increasingly precise abstract models in an *abstraction-refinement loop*. Each abstract model is an abstraction of the program *by construction*. We typically start the loop with a very coarse, imprecise abstraction. Generally, imprecise abstractions yield coarse approximations of the property at hand but are cheaper to model check than abstractions that are more precise. The loop terminates only once an abstract model yields an approximation that is satisfactory. The key step in the abstraction-refinement loop is the automatic identification of new, more precise abstractions when the current approximation is not satisfactory — an automated *refinement step*. A good refinement procedure ensures that, in the long run, the approximation of the property under consideration improves while keeping the cost of model checking as low as possible by ensuring that no unnecessary information is added to the abstractions.

A particularly successful instantiation of the abstraction-refinement paradigm is that of *counter-example guided abstraction refinement* (CEGAR) [Kur94, CGJ+00]. It is at the heart of prominent abstraction-based model checkers for (non-probabilistic) software such as SLAM [BR01], BLAST [HJMS03] MAGIC [CCG+04] and SATABS [CKSY05]. In CEGAR, the validity of a certain class of qualitative properties, called *safety* properties,[1] is approximated via *existential abstractions* [CGL94]. Existential abstractions are abstractions that over-approximate the possible behaviours of the program. In practice this means that, in CEGAR, safety properties can be *verified* but not *refuted* via abstractions. Fortunately, in a non-probabilistic setting, the falsity of safety properties is witnessed by a single finite path of the model — a *counter-example*. Refutation of qualitative safety properties is achieved by trying to map counter-examples of abstract models to counter-examples of the program [CGJ+00]. Counter-examples for which this fails are called *spurious* counter-examples. The principal characteristic of CEGAR is that refinement is achieved by eliminating spurious abstract counter-examples. This elimination can

[1] The property $Reach^+$ defined in Section 3.3.1 is a prime example of a safety property.

be done via *weakest preconditions* [CGJ+00] or via *interpolation* [HJMM04].

A direct *quantitative* adaptation of CEGAR for a guarded command language is presented in [HWZ08]. Here, strong probabilistic simulations of [SL94] are used to obtain one bound on the quantitative property under consideration. The remaining bound can be obtained by analysing probabilistic counter-examples [HK07, AL09]. These counter-examples are also used for refinement in [HWZ08]. We do not discard the possibility that a direct adaptation of CEGAR could be used to verify probabilistic software. However, in this chapter, we opt to use an abstraction-refinement method for which we do not need to analyse probabilistic counter-examples.

We therefore look at abstraction-refinement methodologies that use *three-valued* abstractions such as [SG07, GC06]. The abstractions used in this setting are typically *modal* or *mixed* abstractions [LT88, DGG97]. The main benefit of using these abstractions is that both verification and refutation of qualitative properties can be achieved directly via abstractions (i.e. there is no need to consider counter-examples for the purposes of refutation). Moreover, as opposed to realising refinement via the elimination of spurious abstract counter-examples, the refinement step for three-valued abstractions comprises the elimination of indefinite results from the abstraction [SG07]. Our justification for using three-valued abstraction frameworks is that they avoid the need to analyse counter-examples and, in a quantitative setting, counter-examples are substantially more complex than their non-probabilistic counterparts [HK07, AL09].

A quantitative analogue of the three-valued abstraction-refinement approach uses abstractions that provide specifically tailored lower and upper bounds on quantitative properties under consideration. To this end, the abstractions we will use in this chapter are the *stochastic two-player game abstractions* (or "games") introduced in [KNP06] — we discussed game abstractions in Section 3.4.1. Our justification of using game abstractions instead of, say, a probabilistic adaptation of modal or mixed abstractions is two-fold. Firstly, stochastic two-player games are themselves a well-understood formalism and are very similar in nature to MDPs. In practice, this means we easily adapt existing machinery for computing properties of MDPs to analyse game abstractions. Secondly, the nature of game abstractions allows us to compute abstractions in much the same way that exis-

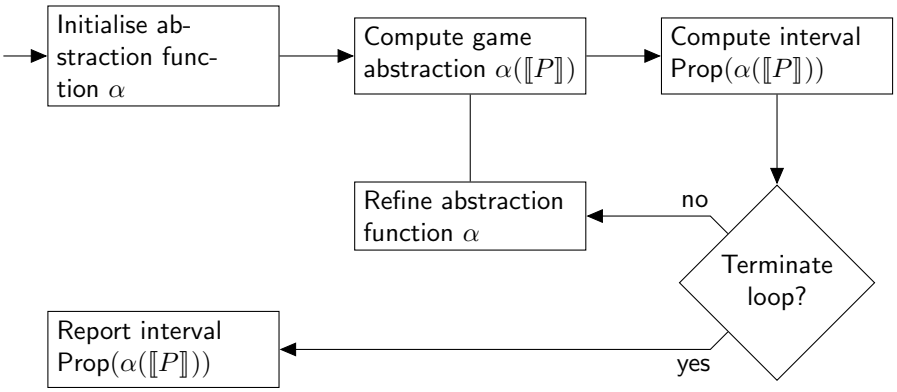


FIGURE 5.1: High-level overview of how to automatically approximate $Prop([\![P]\!])$ of a probabilistic program P via a *quantitative abstraction-refinement loop* using the game abstractions of Section 3.4.

tential abstractions are computed. This makes it easier to adapt a CEGAR-based model checker to compute game abstractions instead.

In this chapter, we describe an instantiation of the abstraction-refinement paradigm for probabilistic software using game abstractions. The focus of our presentation is on the procedures, heuristics and optimisations that are needed to make this abstraction-refinement loop work for real probabilistic software. For the underlying theory, we will directly employ the theoretical framework of game abstractions described in Section 3.4.

Before we go into the technical details of our abstraction-refinement loop, however, we give a high-level overview of this chapter.

Overview of chapter The focus in this chapter is on computing, for a given probabilistic program $P \in \text{PROG}$, and a given quantitative property $Prop \in \{Prob^-, Prob^+, Cost^-, Cost^+\}$, the *value* $Prop([\![P]\!]) \in \mathbb{R}$. We will compute $Prop([\![P]\!])$ via an abstraction-refinement loop, which we depict in Figure 5.1. Similarly to qualitative abstraction-refinement loops, our loop comprises three important phases: the *abstraction* phase, *model checking* phase and *refinement* phase. We first discuss the abstraction phase.

Akin to most abstraction-based software model checkers, we will focus on *predicate abstractions* [GS97]. In our case predicate abstractions are games that abstract the data space of programs via a finite set of predicates but that leave the control-flow structure of the programs intact. We will introduce game-based predicate abstractions of probabilistic programs in Section 5.3.1. For our overview here it is sufficient to know that predicates

induce an abstraction function α and, following Section 3.4, each such abstraction function corresponds to a game abstraction $\alpha([\![P]\!])$ of $[\![P]\!]$. Even if $\alpha([\![P]\!])$ itself is relatively small, we face the problem that, for realistic programs, we cannot construct this abstraction directly from the low-level MDP semantics $[\![P]\!]$. A common strategy to obtain existential abstractions of non-probabilistic programs in CEGAR is to let SAT solvers compute the abstract transition functions [LBC03, CKSY04]. We will show in Section 5.3.2 how to adapt these methods to compute game abstractions of probabilistic programs.

Once we have constructed the game abstraction $\alpha([\![P]\!])$, we evaluate the property *Prop* on $\alpha([\![P]\!])$ with a straightforward extension of standard verification methods for MDPs (see, e.g., [Par02, HKNP06]). By Theorem 3.30, we have:

$$Prop(\alpha([\![P]\!])) \leq \langle Prop([\![P]\!]), Prop([\![P]\!])\rangle \ .$$

Here, the order $\leq$ is the one described in Definition 3.8. That is, $Prop(\alpha([\![P]\!]))$ is a tuple $\langle l, u\rangle$ such that $Prop([\![P]\!]) \in [l, u]$.

If the difference between the upper bound, u, and the lower bound, l, is small enough then we terminate the abstraction-refinement loop. If this is not the case then our abstraction is not precise enough and we proceed with a *refinement step*. As our abstraction functions are induced by predicates, the refinement step in our framework augments the existing abstraction function with new predicates. In Section 5.4 we show how we can analyse the transition function of $\alpha([\![P]\!])$ to automatically obtain new predicates.

Finally, in Section 5.5 we discuss and evaluate an implementation of the abstraction-refinement loop described in Figure 5.1 for probabilistic ANSI-C software. We experimentally validate our implementation and various heuristics on a range of probabilistic programs. We also describe and evaluate two extensions of our basic approach.

5.2 Assumptions

Our abstraction and refinement procedures make assumptions on the behaviour of probabilistic programs which we will formalise here. Recall that the semantics of control-flow

edges from probabilistic locations are mappings from data states to arbitrary (singleton sets of) *distributions* on data states of the program. Analogously, the semantics of control-flow edges from assignment locations are mappings from data states to arbitrary *sets* (of point distributions) on data states of the program. In practice, such a liberal definition of semantics makes it difficult to automatically construct or refine abstractions. We therefore make the following assumption on probabilistic programs:

Assumption 5.1. *Let* $P = \langle\langle \mathscr{L}, \mathscr{E}\rangle, \{\mathscr{L}_{\text{N}}, \mathscr{L}_{\text{P}}, \mathscr{L}_{\text{B}}\}, \ell_i, \mathscr{L}_{\text{T}}, \mathscr{L}_{\text{C}}, \mathit{Var}, \mathit{Sem}\rangle$ *be a probabilistic program. For every* $\ell \in \mathscr{L}_{\text{N}} \cup \mathscr{L}_{\text{P}}$ *there are* finite *families of functions and reals*

$$\mathit{Sem}^1_\ell, \ldots, \mathit{Sem}^k_\ell : \mathscr{U}_{\mathit{Var}} \to \mathscr{U}_{\mathit{Var}} \quad \text{and} \quad a^1, \ldots, a^k \in [0,1]$$

such that for all $u \in \mathscr{U}_{\mathit{Var}}$ *we have*

$$\begin{aligned}
&\mathit{Sem}(\langle \ell, \textsc{Succ}_{\mathscr{E}}(\ell)\rangle)(u) = \{[\mathit{Sem}^l_\ell(u)] \mid l \in [1,k]\} \quad \text{if } \ell \in \mathscr{L}_{\text{N}} \text{ and}\\
&\mathit{Sem}(\langle \ell, \textsc{Succ}_{\mathscr{E}}(\ell)\rangle)(u) = \{a^1 \cdot [\mathit{Sem}^1_\ell(u)] + \ldots + a^k \cdot [\mathit{Sem}^k_\ell(u)]\} \quad \text{if } \ell \in \mathscr{L}_{\text{P}}\,.
\end{aligned}$$

Essentially we require that the sets available from assignment locations and the distributions available from probabilistic locations can be constructed from a finite number of components for every data state. An indirect consequence of our assumption is that, if we abstract $[\![P]\!]$ using an abstraction function $\alpha : \mathscr{L} \times \mathscr{U}_{\mathit{Var}} \to \hat{S}$ and $|\hat{S}|$ is finite, then $\alpha([\![P]\!])$'s transition function is also finite. That is, there are only finitely many sets of point distributions and singleton sets of distributions on $\hat{S}$ that are of the structure enforced by our assumption.

All (deterministic) assignments *var*=e_1 trivially satisfy our assumption. Of the nondeterministic and probabilistic functions shown in Figure 4.3, the functions *var*=`ndet(`e_1`)`, *var*=`coin(`e_1`,`e_2`)` and *var*=`uniform(`e_1`)` satisfy our assumption if the expressions e_1, e_2 are constants. The assignment *var*=`*` only satisfies the assumption if $\textsc{Type}(\mathit{var})$ is finite. In practice, even if $\textsc{Type}(\mathit{var})$ is infinite, we can still deal with this assignment as long as it does not influence the property under consideration.

5.3 Constructing Game Abstractions

We now discuss various topics related to the abstraction of probabilistic programs. In Section 5.3.1, we explain how *predicates* over a program's data space induce an abstraction relation. Then, in Section 5.3.2, we discuss how we construct such predicate abstractions for real programs in practice.

5.3.1 Predicate Abstractions

In this section, we will explain the abstraction function, α, that we consider for probabilistic programs in practice. Recall that, for a program

$$P = \langle\langle \mathscr{L}, \mathscr{E}\rangle, \{\mathscr{L}_{\text{N}}, \mathscr{L}_{\text{P}}, \mathscr{L}_{\text{B}}\}, \ell_i, \mathscr{L}_{\text{T}}, \mathscr{L}_{\text{C}}, \mathit{Var}, \mathit{Sem}\rangle \ ,$$

the state space of $[\![P]\!]$ is $\mathscr{L} \times \mathscr{U}_{\mathit{Var}}$ and comprises tuples of *program locations* $\ell \in \mathscr{L}$ and *data states* $u \in \mathscr{U}_{\mathit{Var}}$. We consider *predicate abstractions* [GS97] — i.e. games that preserve P's control-flow and that abstract the state space $\mathscr{U}_{\mathit{Var}}$ of P via a finite set of predicates $\mathit{Pred} = \{p_1, \ldots, p_n\}$ on Var.

Formally, these predicates are Boolean functions $p_i : \mathscr{U}_{\mathit{Var}} \to \mathbb{B}$ over the data space of P. The abstract data space under Pred is $\mathscr{L} \times \mathbb{B}^n$. That is, the abstract state space is *finite* and comprises P's control-flow locations and n-tuples of Booleans. The n-tuples are abstract representations of P's data space which do not preserve the value of each variable but just the validity of each predicate in $\{p_1, \ldots, p_n\}$. Under these predicates, an abstraction function, α, has the type $\alpha : (\mathscr{L} \times \mathscr{U}_{\mathit{Var}}) \to (\mathscr{L} \times \mathbb{B}^n)$.

Before we define α, we observe that, in practice, CEGAR model checkers do not keep track of the validity of *every* predicate in *every* control-flow location — often we only actually need to track predicates for a small section of the control-flow and not exploiting this is likely to affect the efficiency and scalability of the model checker. We therefore introduce a *localisation mapping* $\textsc{Map} : \mathscr{L} \times \mathit{Pred} \to \mathbb{B}$ which allows us to define a scope for every predicate. The meaning of $\textsc{Map}(\ell, p_i)$ is that predicate p_i is enabled at control-flow location $\ell \in \mathscr{L}$. This localisation mapping gives rise to localised abstraction

functions $\alpha_\ell : \mathscr{U}_{Var} \to \mathbb{B}^n$ that dictate how the data space $\mathscr{U}_{Var}$ should be abstracted at each control-flow location $\ell \in \mathscr{L}$. For each $u \in \mathscr{U}_{Var}$ this function $\alpha_\ell(u)$ yields the unique n-tuple $\langle b_1, \ldots, b_n \rangle \in \mathbb{B}^n$ that satisfies:

(i) $\forall i \in [1, n] : \quad \textsc{Map}(\ell, p_i) \Rightarrow (b_i \Leftrightarrow p_i(u))$ and

(ii) $\forall i \in [1, n] : \neg\textsc{Map}(\ell, p_i) \Rightarrow \neg b_i$.

The first condition, (i), ensures that, when the predicate $p_i \in \mathit{Pred}$ is enabled at location $\ell \in \mathscr{L}$ then, for all $u \in \mathscr{U}_{Var}$, the value of b_i in $\alpha_\ell(u)$ must match the value of $p_i(u)$. Condition (ii) concerns predicates that are not relevant and simply ensures that the value of b_i in ℓ is false whenever p_i is not enabled in ℓ. Note that α_ℓ is always injective but generally not surjective — i.e. not every valuation of predicates is feasible.

We finally define $\alpha : (\mathscr{L} \times \mathscr{U}_{Var}) \to (\mathscr{L} \times \mathbb{B}^n)$ as follows: for every $\ell \in \mathscr{L}$ and $u \in \mathscr{U}_{Var}$ we take $\alpha(\ell, u) = \langle \ell, \alpha_\ell(u) \rangle$. The abstraction function α induces a game abstraction $\alpha([\![P]\!])$ of $[\![P]\!]$ (see Definition 3.28, page 48).

We will conclude our discussion on predicate abstraction with an example:

Example 5.2. *Recall the simple network program*

$$P = \langle \langle \mathscr{L}, \mathscr{E} \rangle, \{\mathscr{L}_{\mathrm{N}}, \mathscr{L}_{\mathrm{P}}, \mathscr{L}_{\mathrm{B}}\}, \ell_i, \mathscr{L}_{\mathrm{T}}, \mathscr{L}_{\mathrm{C}}, \mathit{Var}, \mathit{Sem} \rangle$$

introduced in Example 4.5 (page 59). Let $\mathit{Pred} = \{p_1\}$ *and let* p_1 *be the predicate* (`fail`). *Moreover, let the predicate localisation mapping be such that* $\textsc{Map}(\ell, p_1)$ *holds only for* $\ell \in \{B, N\}$. *We have, e.g., that* $\alpha_B, \alpha_{P1} \in \mathscr{U}_{Var} \to \mathbb{B}^1$ *are functions such that* $\alpha_B(u) = u(\texttt{fail})$ *and* $\alpha_{P1}(u) = \texttt{ff}$ *for each* $u \in \mathscr{U}_{Var}$. *Now suppose* α *is the abstraction function as defined for Pred and* MAP *in this section. The game abstraction* $\alpha([\![P]\!])$ *is depicted in Figure 5.2 where a state* $\langle \ell, \langle b_1 \rangle \rangle$ *is depicted with a top label "*ℓ*" and a bottom label which, if* p_1 *is enabled according to* MAP, *is "*b_1*" and is "*$*$*", otherwise. Analogously, the game abstraction in Figure 5.3 is the abstraction constructed for an empty set of predicates. We have that* $\mathit{Prob}^+(\alpha([\![P]\!])) = \langle 0, 1 \rangle$ *for both abstractions.*

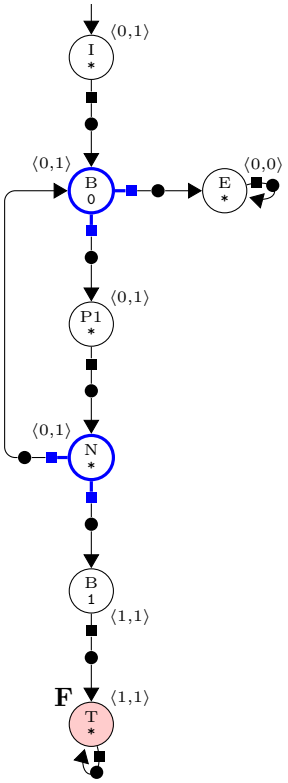


FIGURE 5.2: Game abstraction of Figure 4.7 with predicates $p_1 = (\texttt{f})$ such that MAP(ℓ, p_1) holds only for $\ell = B$.

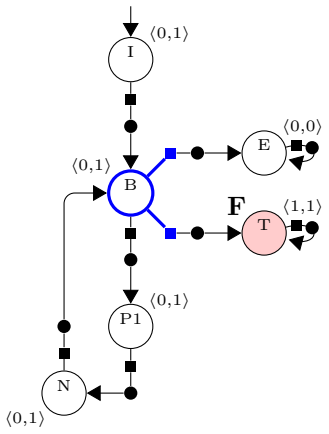


FIGURE 5.3: Game abstraction of Figure 4.7 with the set of predicates, *Pred*, being the empty set.

5.3.2 Enumeration of Transitions with ALL-SAT

Consider again a probabilistic program

$$P = \langle\langle \mathscr{L}, \mathscr{E}\rangle, \{\mathscr{L}_{\mathrm{N}}, \mathscr{L}_{\mathrm{P}}, \mathscr{L}_{\mathrm{B}}\}, \ell_i, \mathscr{L}_{\mathrm{T}}, \mathscr{L}_{\mathrm{C}}, \mathit{Var}, \mathit{Sem}\rangle \ .$$

In the previous section, we have shown how a set of predicates, *Pred*, and a localisation mapping, MAP, together induce an abstraction function α on $[\![P]\!]$'s state space. In turn, in Section 3.4, it is shown how such abstraction functions induce a game abstraction $\alpha([\![P]\!])$. However, even if $\alpha([\![P]\!])$ itself is relatively small, we are typically unable to obtain it directly from $[\![P]\!]$, which is usually too large to construct. This is where our program-level abstraction differs from, say, [KNP06, KKNP10], where we assume that the

MDP to abstract, e.g. $[\![P]\!]$, is readily available.

The key difficulty is in computing $\alpha([\![P]\!])$'s transition function, $\hat{T}$. Computing the state space, $\hat{S}$, and initial states, $\hat{I}$, can be done with standard methods and the functions $\hat{L}$ and $\hat{R}$ are constructed easily due to our restricted use of propositional labelling and costs in our program semantics.[2] Observe that for every $\langle \ell, \mathbf{b} \rangle \in \mathscr{L} \times \mathbb{B}^n$ we have that

$$\hat{T}(\langle \ell, \mathbf{b} \rangle) = \left\{ \{ \alpha(\textsc{Join}([\ell'], \lambda)) \mid \langle \ell, \ell' \rangle \in \mathscr{E}, \lambda \in \mathit{Sem}(\langle \ell, \ell' \rangle)(u) \} \mid u \in \alpha_\ell^{-1}(\mathbf{b}) \right\} .$$

That is, any data state $u \in \mathscr{U}_{Var}$ of P abstracted by $\mathbf{b}$ induces a player C state in $\hat{T}(\langle \ell, \mathbf{b} \rangle)$. This player C state comprises an abstracted version of every transition from $\langle \ell, u \rangle$.

Instead of directly constructing $\hat{T}$ from $[\![P]\!]$, the approach we follow in this section is to enumerate transitions in $\hat{T}$ using SAT solvers. That is, for each control-flow location we will construct a SAT formula such that a satisfiable instance of this formula corresponds to a player A transition in $\hat{T}$ from this location and we will use an ALL-SAT procedure to enumerate all such transitions — akin to [LBC03, CKSY04].

For $\langle \ell, \mathbf{b} \rangle \in \mathscr{L} \times \alpha_\ell(\mathscr{U}_{Var})$, we will show how to construct $\hat{T}(\langle \ell, \mathbf{b} \rangle)$ depending on the location type of ℓ. We first discuss the case when ℓ is an assignment locations in detail and then consider other location types.

Assignment locations Suppose $\ell \in \mathscr{L}_{\mathrm{N}}$ and $\ell' = \textsc{Succ}_{\mathscr{E}}(\ell)$. By Assumption 5.1 and using the definitions of α, we are able to rewrite $\hat{T}(\langle \ell, \mathbf{b} \rangle)$ as follows:

$$\begin{aligned} \hat{T}(\langle \ell, \mathbf{b} \rangle) &= \left\{ \{ \textsc{Join}([\ell'], \alpha_{\ell'}([\mathit{Sem}_\ell^i(u)])) \mid i \in [1,k] \} \mid u \in \alpha_\ell^{-1}(\mathbf{b}) \right\} \\ &= \left\{ \{ [\langle \ell', \alpha_{\ell'}(\mathit{Sem}_\ell^i(u)) \rangle] \mid i \in [1,k] \} \mid u \in \alpha_\ell^{-1}(\mathbf{b}) \right\} . \end{aligned}$$

We can use this definition to observe the following: every player C state in $\hat{T}(\langle \ell, \mathbf{b} \rangle)$ is of the form $\{[\langle \ell', \mathbf{b}^i \rangle] \mid i \in [1,k]\}$ for some $\mathbf{b}^1, \ldots, \mathbf{b}^k \in \mathbb{B}^n$ and, more importantly, we have $\{[\langle \ell', \mathbf{b}^i \rangle] \mid i \in [1,k]\} \in \hat{T}(\langle \ell, \mathbf{b} \rangle)$ if and only if

$$(\mathbf{b} = \alpha_\ell(u)) \wedge (\mathbf{b}^1 = \alpha_{\ell'}(\mathit{Sem}_\ell^1(u))) \wedge \ldots \wedge (\mathbf{b}^k = \alpha_{\ell'}(\mathit{Sem}_\ell^k(u))) \tag{5.1}$$

[2] In practice, we do not compute $\hat{S}$ directly. We construct the *reachable* states of $\hat{S}$ from $\hat{T}$ and $\hat{I}$.

is satisfied for some $u \in \mathscr{U}_{Var}$. Next, we encode (5.1) in propositional logic. To do this, we represent u, $Sem^1_\ell(u)$, ..., $Sem^k_\ell(u)$ and $\mathbf{b}$, $\mathbf{b}^1$, ... ,$\mathbf{b}^k$ with finite sets of Boolean variables and we encode applications of α_ℓ, $\alpha_{\ell'}$ and $Sem^1_\ell, \ldots, Sem^k_\ell$ as constraints over these variables. The precise details of how this SAT encoding is done is beyond the scope of this thesis — we use existing methods for this encoding and refer to [CKSY04, KS08] for a comprehensive discussion.

To find *all* player A transitions in ℓ, we then employ an ALL-SAT procedure to find *all* values of $\mathbf{b}, \mathbf{b}^1, \ldots, \mathbf{b}^k$ that satisfy (5.1) for some $u \in \mathscr{U}_{Var}$. That is, we use a SAT solver to find a satisfiable instance of (5.1) and we extract the values of $\mathbf{b}, \mathbf{b}^1, \ldots, \mathbf{b}^k$ from this instance and add $\{[\langle \ell', \mathbf{b}^l \rangle] \mid l \in [1,k]\}$ to $\hat{T}(\langle \ell, \mathbf{b} \rangle)$. We then augment (5.1) with a clause that prevents the same values of $\mathbf{b}, \mathbf{b}^1, \ldots, \mathbf{b}^k$ from being found again and run the SAT solver on this augmented formula. We repeat this until (5.1) is found to be unsatisfiable.

We illustrate the abstraction of assignment locations by means of an example:

Example 5.3. *Consider again the program shown in Figure 4.7 and the control-flow edge $\langle N, B \rangle \in \mathscr{E}$ labelled with the assignment* `p=p-1`*. Now suppose we have predicates $p_1 =$ (`p==0`), $p_2 =$ (`p==1`), $p_3 =$ (`p==2`) and all are enabled in both N and B. To satisfy Assumption 5.1, we represent $Sem(\langle N, B \rangle)$ with a single function $Sem^1_N : \mathscr{U}_{Var} \to \mathscr{U}_{Var}$ that yields $Sem^1_N(u) = u_{[\mathtt{p} \mapsto u(\mathtt{p})-1]}$ for all $u \in \mathscr{U}_{Var}$. For this example, expanding α's definition in (5.1) yields the following formula over $\mathbf{b} = \langle b_1, b_2, b_3 \rangle, \mathbf{b}^1 = \langle b^1_1, b^1_2, b^1_3 \rangle$:*

$$\begin{aligned} &b_1 \Leftrightarrow (u(\mathtt{p}) = 0) && \wedge\ b_2 \Leftrightarrow (u(\mathtt{p}) = 1) && \wedge\ b_3 \Leftrightarrow (u(\mathtt{p}) = 2) \\ &b^1_1 \Leftrightarrow (Sem^1_N(u)(\mathtt{p}) = 0) && \wedge\ b^1_2 \Leftrightarrow (Sem^1_N(u)(\mathtt{p}) = 1) && \wedge\ b^1_3 \Leftrightarrow (Sem^1_N(u)(\mathtt{p}) = 2)\ . \end{aligned}$$

From the definition of Sem^1_N we can easily see that we have $Sem^1_N(u)(\mathtt{p}) = n$ if and only if $u(\mathtt{p}) - 1 = n$.[3] *We encode this formula in propositional logic and employ a SAT solver to find the following satisfiable valuations of $\mathbf{b}$ and $\mathbf{b}^1$ (we also show corresponding values*

[3] At least this is so for $n \in \{0, 1, 2\}$ where the range of `p` is not an issue.

of $u \in \mathscr{U}_{Var}$):

$$\begin{aligned}
&\mathbf{b} = \langle 100\rangle,\ \mathbf{b}^1 = \langle 000\rangle, && (u(\mathtt{p}) = 0)\\
&\mathbf{b} = \langle 010\rangle,\ \mathbf{b}^1 = \langle 100\rangle, && (u(\mathtt{p}) = 1)\\
&\mathbf{b} = \langle 001\rangle,\ \mathbf{b}^1 = \langle 010\rangle, && (u(\mathtt{p}) = 2)\\
&\mathbf{b} = \langle 000\rangle,\ \mathbf{b}^1 = \langle 001\rangle, && (u(\mathtt{p}) = 3)\\
&\mathbf{b} = \langle 000\rangle,\ \mathbf{b}^1 = \langle 000\rangle, && (u(\mathtt{p}) = -2,850)\ .
\end{aligned}$$

From these satisfying assignments we can construct $\hat{T}$ for transitions from N. We show an arbitrary value of p *that induces each satisfying assignment in brackets. From these satisfying assignments we get, e.g., $\hat{T}(\langle N, 100\rangle) = \{\{[\langle B, 000\rangle]\}\}$ and $\hat{T}(\langle N, 000\rangle) = \{\{[\langle B, 001\rangle]\}, \{[\langle B, 000\rangle]\}\}$. Informally, the player* A *non-determinism in $\langle N, 000\rangle$ is related to our inability to decide whether $Sem^1_N(u)(\mathtt{p}) = 2$ when $u(\mathtt{p}) \notin \{0, 1, 2\}$. Note that a similar transition function (taking into account an additional predicate* f*) between B and N can be found in the abstraction depicted in Figure 5.8 (page 88).*

Probabilistic locations Perhaps surprisingly, due to Assumption 5.1, our abstraction method for probabilistic locations is essentially the same as those for assignment locations. That is, we use the same SAT methods and formulas. The only difference is that we extract different transitions from the satisfiable instances. Suppose $\ell \in \mathscr{L}_{\mathrm{P}}$ and $\ell' = \textsc{Succ}_{\mathscr{E}}(\ell)$. We can rewrite $\hat{T}(\langle \ell, \mathbf{b}\rangle)$ as follows:

$$\begin{aligned}
\hat{T}(\langle \ell, \mathbf{b}\rangle) &= \big\{\{\textsc{Join}([\ell'], \alpha_{\ell'}(a^1 \cdot [Sem^l_\ell(u)] + \ldots + a^k \cdot [Sem^k_\ell(u)]))\} \mid u \in \alpha_\ell^{-1}(\mathbf{b})\big\}\\
&= \big\{\{\textsc{Join}([\ell'], (a^1 \cdot [\alpha_{\ell'}(Sem^1_\ell(u))] + \ldots a^k \cdot [\alpha_{\ell'}(Sem^k_\ell(u))]))\} \mid u \in \alpha_\ell^{-1}(\mathbf{b})\big\}\ .
\end{aligned}$$

Clearly, player C states in $\hat{T}(\langle \ell, \mathbf{b}\rangle)$ are of the form $\{\textsc{Join}([\ell'], (a^1 \cdot [\mathbf{b}^1] + \ldots a^k \cdot [\mathbf{b}^k]))\}$ for $\mathbf{b}^1, \ldots, \mathbf{b}^k \in \mathbb{B}^n$. Moreover, we have that a player C state induced by $\mathbf{b}^1, \ldots, \mathbf{b}^k$ is in $\hat{T}(\langle \ell, \mathbf{b}\rangle)$ if and only if (5.1) holds. We use the procedure described for assignment locations to find values of $\mathbf{b}, \mathbf{b}^1, \ldots, \mathbf{b}^k$ that satisfy (5.1) for some $u \in \mathscr{U}_{Var}$.

Branching locations Remaining to consider are branching locations $\ell \in \mathscr{L}_{\text{B}}$. Let us again expand the definition of $\hat{T}(\langle \ell, \mathbf{b} \rangle)$:

$$\begin{aligned}\hat{T}(\langle \ell, \mathbf{b} \rangle) &= \left\{ \{ \textsc{Join}([\ell'], \alpha_{\ell'}([u])) \mid \langle \ell, \ell' \rangle \in \mathscr{E}, \mathit{Sem}(\langle \ell, \ell' \rangle)(u) = \{[u]\} \} \mid u \in \alpha_\ell^{-1}(\mathbf{b}) \right\} \\ &= \left\{ \{ [\langle \ell', \alpha_{\ell'}(u) \rangle] \} \mid u \in \alpha_\ell^{-1}(\mathbf{b}), \langle \ell, \ell' \rangle \in \mathscr{E}, \mathit{Sem}(\langle \ell, \ell' \rangle)(u) = \{[u]\} \right\} .\end{aligned}$$

This equality holds because the conditionals labelling the outgoing edges of a branching location never overlap (there is no player C non-determinism in branching locations). We now know that player C states in $\hat{T}(\langle \ell, \mathbf{b} \rangle)$ are of the form $\langle \ell', \mathbf{b} \rangle$ with $\ell' \in \mathscr{E}(\ell)$ and $\mathbf{b} \in \mathbb{B}^n$. Moreover, we know that $\{[\langle \ell', \mathbf{b}' \rangle]\} \in \hat{T}(\ell, \mathbf{b})$ if and only if:

$$\alpha_\ell(u) = \mathbf{b} \wedge \alpha_{\ell'}(u) = \mathbf{b}' \wedge \mathit{Sem}(\ell, \ell')(u) = \{[u]\} \tag{5.2}$$

for some $u \in \mathscr{U}_{\mathit{Var}}$. Again, all values of $\mathbf{b}$, $\mathbf{b}'$ that satisfy (5.2) for some $u \in \mathscr{U}_{\mathit{Var}}$ can be found with an ALL-SAT procedure. We encode $\mathbf{b}$, $\mathbf{b}'$, u and u' with Boolean variables and encode $\alpha_\ell, \alpha_{\ell'}$ *and* the condition $\mathit{Sem}(\ell, \ell')(u) = \{[u]\}$ as constraints over these variables. The condition $\mathit{Sem}(\ell, \ell')(u) = \{[u]\}$ essentially just corresponds to the conditional expression, $[e_1]$, that labels this control-flow edge.

The price we pay for using SAT to construct game abstractions is that our ALL-SAT procedure may find the same player A transition many times. This is something that does not occur when SAT is used to compute existential abstractions of non-probabilistic programs [LBC03, CKSY04]. We will illustrate the issue with an example:

Example 5.4. *Consider a program with two Boolean variables* `x`, `y` *and a probabilistic assignment* `x=coin(1,2)` *at a location* $\ell \in \mathscr{L}_{\text{P}}$ *with* $\ell' = \textsc{Succ}_{\mathscr{E}}(\ell)$. *Using Assumption 5.1, we model* $\mathit{Sem}(\langle \ell, \ell' \rangle)$ *with two functions* $\mathit{Sem}^1_\ell(u) = u_{[\mathtt{x} \mapsto 0]}$ *and* $\mathit{Sem}^2_\ell(u) = u_{[\mathtt{x} \mapsto 1]}$ *and two real values* $a^1 = a^2 = \frac{1}{2}$. *Suppose we have a single predicate* $p_1 = (\mathtt{x} = \mathtt{y})$ *which is enabled at* ℓ'. *This induces the following SAT formula over variables* $\mathbf{b} = \langle b_1 \rangle$, $\mathbf{b}^1 = \langle b_1^1 \rangle$ *and* $\mathbf{b}^2 = \langle b_1^2 \rangle$ *and* $u \in \mathscr{U}_{\mathit{Var}}$*:*

$$\neg b_1 \wedge b_1^1 \Leftrightarrow (\mathit{Sem}^1_\ell(\mathtt{x}) = \mathit{Sem}^1_\ell(\mathtt{y})) \wedge b_1^2 \Leftrightarrow (\mathit{Sem}^2_\ell(\mathtt{x}) = \mathit{Sem}^2_\ell(\mathtt{y})) .$$

We remark that we have $\neg b_1$ *because* p_1 *is not enabled at* ℓ*. Given the definition of* Sem^1_ℓ *and* Sem^2_ℓ *the SAT formula can be rewritten to*

$$\neg b_1 \wedge b_1^1 \Leftrightarrow (0 = u(\mathtt{y})) \wedge b_1^2 \Leftrightarrow (1 = u(\mathtt{y}))\ .$$

Satisfiable assignments to this SAT formula are $\mathbf{b} = \langle 0\rangle$, $\mathbf{b}^1 = \langle 0\rangle$, $\mathbf{b}^2 = \langle 1\rangle$ *and* $\mathbf{b} = \langle 0\rangle$, $\mathbf{b}^1 = \langle 1\rangle$, $\mathbf{b}^2 = \langle 0\rangle$*. The assignments correspond to the same player* C *state. That is, both assignments induce the player* C *state* $\{\textsc{Join}([\ell'], \frac{1}{2}\cdot[\langle \ell', 0\rangle] + \frac{1}{2}\cdot[\langle \ell', 1\rangle])\}$ *in* $\hat{T}(\langle \ell, 0\rangle)$*.*

Through a syntactic analysis that establishes which variables are affected by the assignment under consideration, we are sometimes able to detect when there is no player A non-determinism. In this case, instead of enumerating player A transitions via SAT, we enumerate over player C non-determinism instead. This helps us deal with non-deterministic assignments (like *var*=∗) that do not affect the property under consideration.

5.4 Refining Predicate Abstractions

So far, our discussion on game abstractions of probabilistic programs has been restricted to computing abstractions of programs under fixed sets of predicates and fixed localisation mappings. In this section, we focus on practical methods to *automatically* find good predicates and localisation mappings. That is, we will discuss how we can realise the *refinement step* in Figure 5.1.

In this section, to aid our presentation, we focus on approximating $Prob^+([\![P]\!])$ for some probabilistic program P and, where necessary, we will indicate how the refinement step would change if we were to consider $Prob^-$, $Cost^-$ or $Cost^+$, instead.

Suppose we have a probabilistic program P and we have constructed a predicate abstraction $\alpha([\![P]\!])$ induced by predicates *Pred* and a localisation mapping MAP. We need to refine the abstraction only if $\alpha([\![P]\!])$ is imprecise. That is, if

$$\mathrm{UB}(Prob^+(\alpha([\![P]\!]))) - \mathrm{LB}(Prob^+(\alpha([\![P]\!]))) > 0\ . \tag{5.3}$$

The task of our refinement step is to define a new abstraction function, $\alpha^{\#}$, through new predicates $\mathit{Pred}^{\#}$, and a new localisation mapping $\textsc{Map}^{\#} : \mathscr{L} \times \mathit{Pred}^{\#} \to \mathbb{B}$ on P's control-flow locations. To ensure we obtain a *more* precise abstraction than $\alpha([\![P]\!])$, we require $\mathit{Pred}^{\#} \supseteq \mathit{Pred}$ and that, for every predicate $p_i \in \mathit{Pred}$ and control-flow location $\ell \in \mathscr{L}$ we have that $\textsc{Map}(\ell, p_i) \Rightarrow \textsc{Map}^{\#}(\ell, p_i)$.

The only cause of imprecision in game abstractions is player A non-determinism. Intuitively, the basic principle of our refinement step is therefore to eliminate player A non-determinism.

Our discussion on the refinement step begins in Section 5.4.1 with identifying the player A states in which we want to eliminate player A non-determinism. We then show in Section 5.4.2 how to augment $\mathit{Pred}^{\#}$ and $\textsc{Map}^{\#}$ with predicates that eliminate this choice. Finally, in Section 5.4.3, we discuss a heuristic for propagating predicates.

5.4.1 Refinable States

For presentational clarity, in this section, we will consider arbitrary[4] game abstractions $\hat{G} = \langle \hat{S}, \hat{I}, \hat{T}, \hat{L}, \hat{R} \rangle$ of MDPs $M = \langle S, I, T, L, R \rangle$ as obtained via an abstraction function $\alpha : S \to \hat{S}'$, i.e. $\alpha(M) = \hat{G}$. We still focus on the property Prob^{+} and we will point out where the discussion differs for Prob^{+}, Cost^{-} and Cost^{+}.

We will define which states of $\hat{G}$ are *refinable*. Informally, a state, $\hat{s} \in \hat{S}$, is said to be refinable if restricting the choices made by player A in $\hat{s}$ may influence the bounds $\mathit{Prob}^{-+}(\hat{G}, \hat{s})$ or $\mathit{Prob}^{++}(\hat{G}, \hat{s})$. In refinable states, it may pay to eliminate player A non-determinism. In contrast, in states that are not refinable, both the lower bound, $\mathit{Prob}^{-+}(\hat{G}, \hat{s})$, and the upper bound, $\mathit{Prob}^{++}(\hat{G}, \hat{s})$, are attainable regardless of player A's strategy, so there is no indication that eliminating the choice between these player A transitions in non-refinable states would improve the precision of the abstraction.

To aid the formalisation of refinable states, we define bounds on Prob^{+} for player C

[4] As opposed to just abstractions of probabilistic programs.

states. That is, for every $\hat{\Lambda} \in \overline{\mathbb{P}}\mathbb{D}\hat{S}$ we define $Prob^{-+}(\hat{G}, \hat{\Lambda})$ and $Prob^{++}(\hat{G}, \hat{\Lambda})$ as follows:

$$Prob^{-+}(\hat{G}, \hat{\Lambda}) = \sup_{\hat{\Lambda} \to \hat{\lambda}} \left(\sum_{\hat{s}' \in \hat{S}} \left(\hat{\lambda}(\hat{s}') \cdot Prob^{-+}(\hat{G}, \hat{s}') \right) \right) ,$$

$$Prob^{++}(\hat{G}, \hat{\Lambda}) = \sup_{\hat{\Lambda} \to \hat{\lambda}} \left(\sum_{\hat{s}' \in \hat{S}} \left(\hat{\lambda}(\hat{s}') \cdot Prob^{++}(\hat{G}, \hat{s}') \right) \right) .$$

Intuitively, $Prob^{-+}(\hat{G}, \hat{\Lambda})$ is the best lower bound player A can achieve once he has transitioned to a player C state $\hat{\Lambda}$ and, analogously, $Prob^{++}(\hat{G}, \hat{\Lambda})$ is the best upper bound player A can achieve once he has transitioned to $\hat{\Lambda}$. The definition of these functions is directly derived from the fixpoint characterisation of $Prob^{+}$ in Section 3.26 and is the only point of our discussion that would differ when one considers $Prob^{-}$, $Cost^{-}$ or $Cost^{+}$, instead. We illustrate the definition of $Prob^{-+}(\hat{G}, \hat{\Lambda})$ and $Prob^{++}(\hat{G}, \hat{\Lambda})$ with an example:

Example 5.5. *Consider the game $\hat{G}$ depicted in Figure 5.3 (page 71). For player A states $\{[\ell, \langle\rangle]\}$ where $\ell \in \{B, N, P1, T, E\}$ we have that*

$$Prob^{-+}(\hat{G}, \langle T, \langle\rangle\rangle) = Prob^{++}(\hat{G}, \langle T, \langle\rangle\rangle) = 1 \text{ and}$$
$$Prob^{-+}(\hat{G}, \langle E, \langle\rangle\rangle) = Prob^{++}(\hat{G}, \langle E, \langle\rangle\rangle) = 0 .$$

and for $\ell \in \{P1, N, B\}$ we have $Prob^{-+}(\hat{G}, \langle \ell, \langle\rangle\rangle) = 0$ and $Prob^{++}(\hat{G}, \langle \ell, \langle\rangle\rangle) = 1$.

Moreover, for all player C states $\{[\ell, \langle\rangle]\}$ with $\ell \in \{B, N, P1, T, E\}$ we have

$$Prob^{-+}(\hat{G}, \{[\ell, \langle\rangle]\}) = Prob^{-+}(\hat{G}, \langle \ell, \langle\rangle\rangle) \text{ and}$$
$$Prob^{++}(\hat{G}, \{[\ell, \langle\rangle]\}) = Prob^{++}(\hat{G}, \langle \ell, \langle\rangle\rangle) .$$

We can now define which player C states are *minimal* or *maximal* in $\hat{T}(\hat{s})$ in the sense that, if player A is interested in achieving the lowest or highest bound, respectively, then he *must* transition to such a player C state.[5] For every state $\hat{s} \in \hat{S}$ we define minimal

[5] That is, unless $\hat{s}$ is a target state. We exclude such states from our discussion — target states are *never* refinable when considering probabilistic safety or liveness properties.

player C states $V_{\hat{s}}^{-} \subseteq \hat{T}(\hat{s})$ and maximal player C states $V_{\hat{s}}^{+} \subseteq \hat{T}(\hat{s})$ as follows:

$$V_{\hat{s}}^{-} = \{\hat{\Lambda} \in \hat{T}(\hat{s}) \mid Prob^{-+}(\hat{G}, \hat{\Lambda}) = Prob^{-+}(\hat{G}, \hat{s})\} ,$$
$$V_{\hat{s}}^{+} = \{\hat{\Lambda} \in \hat{T}(\hat{s}) \mid Prob^{++}(\hat{G}, \hat{\Lambda}) = Prob^{++}(\hat{G}, \hat{s})\} .$$

The intuition is that, if player A wants to achieve the lower bound, he *has* to play with a strategy that transitions to player C states in $V_{\hat{s}}^{-}$. Analogously, for the upper bound, player A has to transition to player C states in $V_{\hat{s}}^{+}$.[6] Note that the sets $V_{\hat{s}}^{-}$ and $V_{\hat{s}}^{+}$ are guaranteed to be non-empty for predicate abstractions of probabilistic programs because, by construction, these games are finitely branching for player A. However, the sets $V_{\hat{s}}^{-}$ and $V_{\hat{s}}^{+}$ are generally *not* disjoint. In fact, in practice many states have that $V_{\hat{s}}^{-} = V_{\hat{s}}^{+}$.

We clarify the definitions of $V_{\hat{s}}^{-}$ and $V_{\hat{s}}^{+}$ by means of an example:

Example 5.6. *Consider again the game $\hat{G}$ depicted in Figure 5.3. We have that $V^{-}_{\langle P1,\langle\rangle\rangle} = V^{+}_{\langle P1,\langle\rangle\rangle} = \{[\langle N,\langle\rangle\rangle]\}$. That is, a single player A transition is responsible for achieving both the lower bound, 0, and the upper bound, 1, in $\langle P1,\langle\rangle\rangle$. The actual cause of $\langle P1,\langle\rangle\rangle$'s unsatisfactory bounds is the player A choice in $\langle B,\langle\rangle\rangle$: we have that $V^{-}_{\langle B,\langle\rangle\rangle} = \{\{[\langle P1,\langle\rangle\rangle]\},\{[\langle E,\langle\rangle\rangle]\}\}$ and $V^{+}_{\langle P1,\langle\rangle\rangle} = \{\{[\langle P1,\langle\rangle\rangle]\},\{[\langle T,\langle\rangle\rangle]\}\}$ — i.e. player A has the power to choose between the lower and upper bound by transitioning to player C states $\{[\langle E,\langle\rangle\rangle]\}$ or $\{[\langle T,\langle\rangle\rangle]\}$, respectively.*

This example shows that the fact that a player A state, $\hat{s}$, has unsatisfactory bounds may not be caused by a choice available to player A in $\hat{s}$ — it may instead be caused by player A non-determinism in some player A state reachable from $\hat{s}$. This means that the bounds in $\hat{s}$ may not improve if we eliminate player A non-determinism in $\hat{s}$.

However, suppose player A transitions to a player C state in $\hat{T}(\hat{s}) \setminus V_{\hat{s}}^{-}$. In this case, by definition of $V_{\hat{s}}^{-}$, the lower bound, $Prob^{-+}(\hat{G}, \hat{s})$, can no longer be attained. Because $V_{\hat{s}}^{-}$ is necessarily non-empty this means that the choice made by player A in $\hat{s}$ *is* directly relevant to the bounds in $\hat{s}$. That is, when $\hat{T}(\hat{s}) \setminus V_{\hat{s}}^{-}$ is non-empty, player A has the power to choose between potentially achieving the lower bound (by transitioning to a player C

[6] However, unlike the lower bound, for the upper bound not *every* player A strategy that picks only maximal player C states actually achieves the upper bound.

state in $V_{\hat{s}}^-$) and definitely not achieving the lower bound (by transitioning to a player C state in $\hat{T}(\hat{s}) \setminus V_{\hat{s}}^-$). If $\hat{T}(\hat{s}) \setminus V_{\hat{s}}^-$ *is* empty, however, then player A has no such power and the lower bound can be achieved regardless of player A's choice in $\hat{s}$.

An analogous argument holds for the upper bound: if and only if the set $\hat{T}(\hat{s}) \setminus V_{\hat{s}}^+$ is non-empty, player A can choose between player C states in $\hat{T}(\hat{s}) \setminus V_{\hat{s}}^+$ and player C states in $V_{\hat{s}}^+$ and hence player A can make the upper bound $Prob^{++}(\hat{G}, \hat{s})$ unattainable.

Our definition of a *refinable* state follows naturally from these observations: we call a state $\hat{s}$ *refinable* if and only if either $\hat{T}(\hat{s}) \setminus V_{\hat{s}}^-$ is non-empty *or* $\hat{T}(\hat{s}) \setminus V_{\hat{s}}^+$ is non-empty.

Example 5.7. *In Fig. 5.3 the state* $\langle B, \langle\rangle\rangle$ *is refinable as* $\hat{T}(\langle B, \langle\rangle\rangle) \setminus V^-_{\langle B,\langle\rangle\rangle} = \{\{[\langle T, \langle\rangle\rangle]\}\}$ *and* $\hat{T}(\langle B, \langle\rangle\rangle) \setminus V^+_{\langle B,\langle\rangle\rangle} = \{\{[\langle E, \langle\rangle\rangle]\}\}$. *Another refinable state is* $\langle B, \langle 0\rangle\rangle$ *in Fig. 5.2.*

Observe that a sufficient condition under which a state $\hat{s}$ is refinable is that $V_{\hat{s}}^- \neq V_{\hat{s}}^+$. In [KKNP10] it is shown that a state satisfying this condition always exists in games for which (5.3) holds.[7] Because our propositional and reward labelling differs from [KKNP10], this result is subject to the fact the L and R are exact for every state — this is the case for all predicate abstractions of all probabilistic programs by construction.

Refinable state selection In practice, a given game abstraction, $\alpha([\![P]\!])$, may feature many refinable states and, considering we are only refining one of them, we need to ensure we make an informed choice as to which of these states we will use in our refinement step. We conclude this section by discussing two heuristics for selecting a refinable state.

Our first heuristic selects what we call a *coarsest* refinable state in $\alpha([\![P]\!])$. A coarsest refinable state of $\alpha([\![P]\!])$ is such that the difference in lower and upper bound, i.e. $Prob^{++}(\alpha([\![P]\!]), \langle \ell, \mathbf{b}\rangle) - Prob^{-+}(\alpha([\![P]\!], \langle \ell, \mathbf{b}\rangle)$, is at least as great as the difference in any other refinable state of $\alpha([\![P]\!])$. The idea of refining a coarsest refinable state is that we refine $\alpha([\![P]\!])$ where we are most likely to make an impact. The intuition is that, if a refinable state already has relatively tight bounds, then refining it is less likely to help our overall approximation.

In contrast, our second heuristic selects what we call a *nearest* refinable state in

[7] This result, i.e. [KKNP10, Lemma 6], should be accredited to Gethin Norman and David Parker.

$\alpha([\![P]\!])$. A nearest refinable state $\langle \ell, \mathbf{b} \rangle$ of $\alpha([\![P]\!])$ is such that the shortest play in $\alpha([\![P]\!])$ from an initial state to $\langle \ell, \mathbf{b} \rangle$ is at least as short as the shortest such play for any other refinable state of $\alpha([\![P]\!])$. The idea of refining the nearest refinable state is that we refine $\alpha([\![P]\!])$ as close to the initial states as possible in the hope that a refinement of $\langle \ell, \mathbf{b} \rangle$ has a better chance of affecting the bounds at initial states.

5.4.2 Predicate Discovery

Let $M, \hat{G}$ and α be as defined in the previous section. In this section, we discuss how to implement the refinement step by defining a new abstraction function $\alpha^{\#}$ through a predicate set $\mathit{Pred}^{\#}$ and localisation map $\textsc{Map}^{\#}$. We first look how we could implement the refinement step if we could modify $\alpha^{\#}$ directly.

Following the discussion on refinable states in the previous section a sensible refinement strategy is to define $\alpha^{\#}$ in such a way that every refinable state $\hat{s} \in \hat{S}$ of $\alpha(M)$ is "split" into states $\hat{s}^{min}$, $\hat{s}^{max}$, $\hat{s}^{both}$ and $\hat{s}^{none}$ which concretise to sets of states of the MDP M corresponding to player A transitions from $\hat{s}$ to player C states in $V^{-}_{\hat{s}} \setminus V^{+}_{\hat{s}}$, $V^{-}_{\hat{s}} \setminus V^{+}_{\hat{s}}$, $V^{-}_{\hat{s}} \cap V^{+}_{\hat{s}}$ and $\hat{T}(\hat{s}) \setminus (V^{-}_{\hat{s}} \cup V^{+}_{\hat{s}})$, respectively. This is, in fact, the value-based refinement strategy proposed in [KKNP10].[8]

Unfortunately, there are two problems in realising an analogous refinement step for predicate abstractions of probabilistic programs. Firstly, adjustments to $\mathit{Pred}^{\#}$ and $\textsc{Map}^{\#}$ are not local to a refinable state — every state with the same control-flow location is affected. We found that it is not practical to refine *every* refinable state because too many predicates are introduced too quickly. This problem is easily solved by refining only one refinable state, $\langle \ell, \mathbf{b} \rangle$, of an abstracted program, $\alpha([\![P]\!])$, in every refinement step.

Our second issue is that the proposed definition of $\alpha^{\#}$ is difficult to realise through adaptations of $\mathit{Pred}^{\#}$ and $\textsc{Map}^{\#}$. More specifically, we have not found a good way to find predicates that correspond with *sets* of player C states such as $V^{-}_{\langle \ell, \mathbf{b} \rangle}$ or $V^{+}_{\langle \ell, \mathbf{b} \rangle}$. However, we *can* find predicates that eliminate the choice between any two player A transitions. We therefore adopt a scheme in which we first select a refinable state, $\langle \ell, \mathbf{b} \rangle$, and then pick

[8] This refinement strategy should be accredited to David Parker and Gethin Norman.

two transitions, $\langle \ell, \mathbf{b} \rangle \to \hat{\Lambda}^-$ and $\langle \ell, \mathbf{b} \rangle \to \hat{\Lambda}^+$. We ensure that these transitions satisfy at least one of the following conditions:

$$\hat{\Lambda}^- \in V^-_{\langle \ell, \mathbf{b} \rangle} \text{ and } \hat{\Lambda}^+ \in (\hat{T}(\langle \ell, \mathbf{b} \rangle) \setminus V^-_{\langle \ell, \mathbf{b} \rangle}) \text{ or}$$
$$\hat{\Lambda}^+ \in V^+_{\langle \ell, \mathbf{b} \rangle} \text{ and } \hat{\Lambda}^- \in (\hat{T}(\langle \ell, \mathbf{b} \rangle) \setminus V^+_{\langle \ell, \mathbf{b} \rangle}) .$$

Here, $\hat{T}$ is $\alpha([\![P]\!])$'s transition function. We give preference to refinable states and player A transitions that satisfy both conditions. We will eliminate the choice between the two transitions from $\langle \ell, \mathbf{b} \rangle$. The hope is that eliminating the choice between a particular player A transition in $V^-_{\langle \ell, \mathbf{b} \rangle}$ and a particular player A transition in $\hat{T}(\langle \ell, \mathbf{b} \rangle) \setminus V^-_{\langle \ell, \mathbf{b} \rangle}$, say, actually eliminates the choice between all player A transitions between these sets of player C states. We note that, by the definition of refinable states, it is always possible to choose $\langle \ell, \mathbf{b} \rangle \to \hat{\Lambda}^-$ and $\langle \ell, \mathbf{b} \rangle \to \hat{\Lambda}^+$ as described.

In the remainder of this section we explain how to discover new predicates given a refinable state $\langle \ell, \mathbf{b} \rangle$ and a player A choice $\langle \ell, \mathbf{b} \rangle \to \hat{\Lambda}^-$ and $\langle \ell, \mathbf{b} \rangle \to \hat{\Lambda}^+$ such that $\hat{\Lambda}^- \neq \hat{\Lambda}^+$. To eliminate this choice, we will construct an improved set of predicates $\mathit{Pred}^\#$ and a new localisation mapping $\textsc{Map}^\#$. For presentational purposes, we define $\mathit{Pred}^\#$ and $\textsc{Map}^\#$ incrementally through a procedure $\textsc{Add}$. That is, we let $\mathit{Pred}^\#$ and $\textsc{Map}^\#$ be Pred and $\textsc{Map}$ initially and, whenever we see fit to enable a predicate $p : \mathscr{U}_{Var}$ in a location $\ell \in \mathscr{L}$, we will call $\textsc{Add}(\ell, p)$ to adapt $\mathit{Pred}^\#$ and $\textsc{Map}^\#$ accordingly. A call to $\textsc{Add}(\ell, p)$ adds predicate p to $\mathit{Pred}^\#$ and sets $\textsc{Map}^\#(\ell, p)$ to true if this is necessary.[9]

We discuss our predicate discovery mechanism separately for each location type. We will explain our refinement method by refining the abstraction in Figure 5.3, and hence, as the only refinable state in this game has a branching location, this is where we start.

[9] In reality we extract the Boolean atoms from p's propositional structure and add non-trivial such atoms to $\mathit{Pred}^\#$ and $\textsc{Map}^\#$.

Branching locations Suppose $\ell \in \mathscr{L}_{\mathrm{B}}$ is a branching location. In Section 5.3.2, we learnt that $\hat{\Lambda}^-$ and $\hat{\Lambda}^+$ must be of the form

$$\hat{\Lambda}^- = \{[\langle \ell^-, \mathbf{b}^- \rangle]\} \text{ and}$$
$$\hat{\Lambda}^+ = \{[\langle \ell^+, \mathbf{b}^+ \rangle]\}$$

for some target locations $\ell^-, \ell^+ \in \mathscr{E}(\ell)$ and some $\mathbf{b}^-, \mathbf{b}^+ \in \mathbb{B}^n$, respectively. Because of the way we pick $\hat{\Lambda}^-$ and $\hat{\Lambda}^+$, we must have that either $\ell^- \neq \ell^+$ or $\mathbf{b}^- \neq \mathbf{b}^+$. If $\ell^- \neq \ell^-$, then the player A non-determinism between $\hat{\Lambda}^-$ and $\hat{\Lambda}^+$ is caused in part by our inability to establish in $\langle \ell, \mathbf{b} \rangle$ which conditional branch is satisfied. Informally, in this case, the refinement step is simply to add one of the conditionals labelling the control-flow edge $\langle \ell, \ell^- \rangle$ or $\langle \ell, \ell^+ \rangle$. Formally, we take the weakest precondition of one of the branches, e.g. $\mathrm{WP}(\mathit{Sem}(\langle \ell, \ell^- \rangle), \mathtt{tt})$, and call $\textsc{Add}(\ell, \mathrm{WP}(\mathit{Sem}(\langle \ell, \ell^- \rangle), \mathtt{tt}))$ to update $\alpha^\#$ and $\textsc{Map}^\#$.

We illustrate this with an example:

Example 5.8. *Consider the abstraction $\alpha([\![P]\!])$ depicted in Figure 5.3 (page 71) of the program depicted in Figure 4.7 (page 59). In Example 5.7, we have established that $\langle B, \langle\rangle\rangle$ is a refinable state of this game with*

$$V^-_{\langle B, \langle\rangle\rangle} = \{\{[\langle E, \langle\rangle\rangle]\}, \{[\langle N, \langle\rangle\rangle]\}\} \text{ and}$$
$$V^+_{\langle B, \langle\rangle\rangle} = \{\{[\langle T, \langle\rangle\rangle]\}, \{[\langle N, \langle\rangle\rangle]\}\} .$$

We pick $\hat{\Lambda}^-$ to be $\{[\langle T, \langle\rangle\rangle]\}$ and $\hat{\Lambda}^+$ to be $\{[\langle E, \langle\rangle\rangle]\}$ and observe that the control-flow locations in $\hat{\Lambda}^-$ and $\hat{\Lambda}^+$ differ. The control-flow edge $\langle B, T \rangle$ is labelled with a conditional, $\mathtt{[f]}$, and, as $\mathit{WP}(\mathit{Sem}(\langle B, T \rangle), \mathtt{tt}) = \mathtt{f}$, we call $\textsc{Add}(B, \mathtt{f})$ to update $\mathit{Pred}^\#$ and $\textsc{Map}^\#$. The refined abstraction $\alpha^\#([\![P]\!])$ is depicted in Figure 5.4.

The refinement we described was conditional on the fact that $\ell^- \neq \ell^+$. We employ a different approach to refinement when $\ell^- = \ell^+$. If this is the case, since $\hat{\Lambda}^- \neq \hat{\Lambda}^+$, it must be that $\mathbf{b}^- \neq \mathbf{b}^+$. This implies that for some predicate, $p_i \in \mathit{Pred}$, the values b_i^- and b_i^+ differ. In this instance, the player A choice between $\hat{\Lambda}^-$ and $\hat{\Lambda}^+$ is caused in part by our

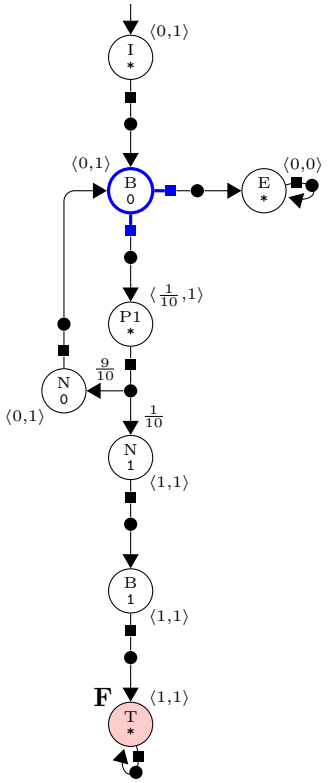


FIGURE 5.4: Predicate abstraction of Figure 4.7 with a predicate $p_1 = (\texttt{f})$ where $\textsc{Map}(\ell, p_1)$ holds only when $\ell \in \{B, N\}$.

inability to establish whether p_i holds in $\langle \ell, \mathbf{b} \rangle$ — we therefore simply call $\textsc{Add}(\ell, p_i)$.

This kind of refinement is needed in the refinable state of Figure 5.7 (page 87).

Assignment locations Suppose $\ell \in \mathscr{L}_{\text{N}}$ is an assignment location with $\ell' = \textsc{Succ}_{\mathscr{E}}(\ell)$ and suppose that Assumption 5.1 is satisfied through functions $Sem^1_\ell, \ldots, Sem^k_\ell$. In Section 5.3.2 we learnt that

$$\hat{\Lambda}^- = \{[\ell', \mathbf{b}^{-,j}] \mid j \in [1,k]\} \text{ and}$$
$$\hat{\Lambda}^+ = \{[\ell', \mathbf{b}^{+,j}] \mid j \in [1,k]\}$$

for some $\mathbf{b}^{-,1}, \ldots, \mathbf{b}^{-,k}, \mathbf{b}^{+,1}, \ldots, \mathbf{b}^{+,k} \in \mathbb{B}^n$. Now, since $\hat{\Lambda}^- \neq \hat{\Lambda}^+$, there must be some predicate $p_i \in \mathit{Pred}$ and some resolution of non-deterministic choice $j \in [1,k]$ such that

$b_i^{-,j} \neq b_i^{+,j}$.[10] We therefore add the weakest precondition of p_i under Sem_ℓ^j to ℓ — i.e. we call $\textsc{Add}(\ell, \mathrm{WP}(Sem_\ell^j, p_i))$. As long as we can define Sem_ℓ^j syntactically then computing this weakest precondition is a cheap, syntactical operation. This is the case for all non-deterministic (and probabilistic) assignments discussed in this thesis.

Example 5.9. *Consider the abstraction $\alpha([\![P]\!])$ depicted in Figure 5.4 of the program P described in Figure 4.7 (page 59). In Example 5.7, we established that $\langle N, \langle * \rangle\rangle$ is a refinable state of this game and that $V^-_{\langle N,\langle * \rangle\rangle} = \{\{\langle B, \langle 0 \rangle\rangle\}\}$ and $V^-_{\langle N,\langle * \rangle\rangle} = \{\{\langle B, \langle 1 \rangle\rangle\}\}$. Hence, we can only pick $\hat{\Lambda}^-$ to be $\{[\langle B, \langle 0 \rangle\rangle]\}$ and $\hat{\Lambda}^+$ to be $\{[\langle B, \langle 1 \rangle\rangle]\}$. Let Sem_N^1 be the function satisfying Assumption 5.1 for N. Evidently, the abstract states disagree on the predicate* `f`*. The weakest precondition $WP(Sem_N^1, \texttt{f})$ is* `f` *as the variable* `f` *is not changed in N. We call $\textsc{Add}(N, \texttt{f})$ to update $Pred^\#$ and $\textsc{Map}^\#$. The refined abstraction $\alpha^\#([\![P]\!])$ is depicted in Figure 5.2.*

Probabilistic locations Similarly to our discussion on abstraction in Section 5.3.2, the refinement procedure for probabilistic locations $\ell \in \mathscr{L}_\mathrm{P}$ is analogous to the refinement procedure for assignment locations. That is, we know $\hat{\Lambda}^-$ and $\hat{\Lambda}^+$ are

$$\hat{\Lambda}^- = \{\textsc{Join}([\ell'], (a^1 \cdot [\mathbf{b}^{-,1}] + \ldots + a^j \cdot [\mathbf{b}^{-,j}])\} \text{ and}$$
$$\hat{\Lambda}^+ = \{\textsc{Join}([\ell'], (a^1 \cdot [\mathbf{b}^{+,1}] + \ldots + a^j \cdot [\mathbf{b}^{+,j}])\}$$

where ℓ' is $\textsc{Succ}_{\mathscr{E}}(\ell)$ and $\mathbf{b}^{-,1}, \ldots, \mathbf{b}^{-,k}, \mathbf{b}^{+,1}, \ldots, \mathbf{b}^{+,k} \in \mathbb{B}^n$. Like for assignment locations we refine based on a predicate $p_i \in Pred$ for which some resolution of probabilistic choice $j \in [1, k]$ is such that $b_i^{-,j} \neq b_i^{+,j}$ via the weakest precondition of p_i under Sem_ℓ^j.

5.4.3 Predicate Propagation

The refinement step, as explained in Section 5.4.2, adds a single predicate to a single control-flow location. In practice, this means that the number of abstraction-refinement steps that is required, and hence the cost of our overall approach, is exorbitantly high.

[10] We can find this p_i because we do not actually reduce $\hat{\Lambda}^-$ and $\hat{\Lambda}^+$ to sets — we remember which branch abstracts to which abstract state.

```
procedure TraceAdd(⟨ℓ, b⟩, p)
begin
  let π̂ be a shortest play from
    an initial state to ⟨ℓ, b⟩
  call TraceAddRec(⟨ℓ, b⟩, p, π̂)
end

procedure TraceAddRec(ℓ, p, π̂)
begin
  call Add(ℓ, p)
  if |π̂| > 0
    let π̂′ be a prefix of π̂ of length |π̂| − 1
      that ends in a player A state
    let ⟨ℓ′, b′⟩ be the last state of π̂′
    let p′ be WP(Sem(⟨ℓ′, ℓ⟩), p)
    call TraceAddRec(ℓ′, p′, π̂′)
  endif
end
```

FIGURE 5.5: TRACEADD: A procedure for predicate propagation via a play of $\alpha([\![P]\!])$.

```
procedure PrecAdd(⟨ℓ, b⟩, p)
begin
  let D be ∅
  call PrecAddRec(⟨ℓ, b⟩, p, D)
end

procedure PrecAddRec(ℓ, p, D)
begin
  if ℓ ∉ D
    call Add(ℓ, p)
    let D′ be D ∪ {ℓ}
    for each ℓ′ ∈ ℰ^{-1}(ℓ)
      let p′ be WP(Sem(⟨ℓ′, ℓ⟩), p)
      call PrecAddRec(ℓ′, p′, D′)
    endfor
  endif
end
```

FIGURE 5.6: PRECADD: A procedure for predicate propagation via the control-flow graph.

However, there is scope to improve our refinement step. Consider again the program depicted in Figure 4.7 (page 59) and its game-based predicate abstraction in Figure 5.3 (page 71). In Example 5.8 we discussed how our refinement step adds a predicate `f` to control-flow location B resulting in the game abstraction depicted in Figure 5.4 (page 84). Although this refined game is a little more precise, effectively we have pushed the player A non-determinism that used to be in $\langle B, \langle\rangle\rangle$ in Figure 5.3 to $\langle N, \langle *\rangle\rangle$ in Figure 5.2. Indeed, Example 5.9 shows how we end up propagating the predicate `f` to N.

We could have anticipated that the player A choice eliminated in $\langle B, \langle\rangle\rangle$ would be pushed back to $\langle N, \langle *\rangle\rangle$; after all, we have no means of knowing whether `f` holds in transitions from $\langle N, \langle *\rangle\rangle$. In this section we will propose two methods, TRACEADD and PRECADD, for automatically propagating predicates that are found by the basic refinement step. These procedures serve as a wrapper around ADD. That is, when we have some refinable state $\langle \ell, \mathbf{b} \rangle \in \mathscr{L} \times \mathscr{U}_{Var}$ and some predicate $p : \mathscr{U}_{Var} \to \mathbb{B}$ for which

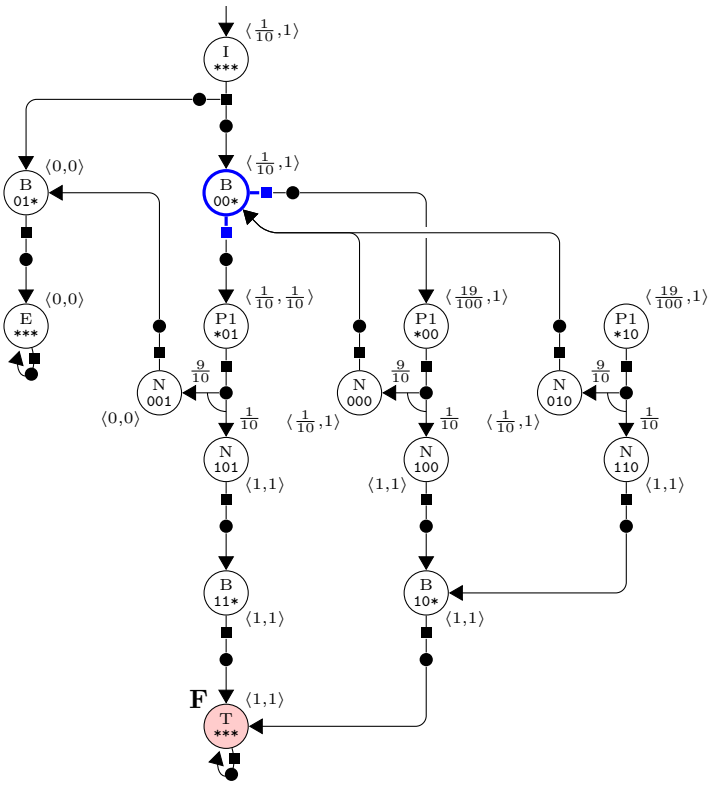


FIGURE 5.7: A predicate abstraction of the program in Figure 4.7 with predicates $p_1 = (\texttt{f})$, $p_2 = (\texttt{p==0})$, $p_3 = (\texttt{p==1})$ such that $\textsc{Map}(\ell, p_i)$ holds only if $i = 1$ and $\ell \in \{B, N\}$, $i = 2$ and $\ell \in \{B, P1, N\}$ or $i = 3$ and $\ell \in \{P1, N\}$.

we call $\textsc{Add}(\ell, p,)$ in Section 5.4.2 then, in this section, we call $\textsc{TraceAdd}(\langle \ell, \mathbf{b} \rangle, p)$ or $\textsc{PrecAdd}(\langle \ell, \mathbf{b} \rangle, p)$ instead. Our first procedure, TRACEADD, is depicted in Figure 5.5. Intuitively, this procedure attempts to ensure that the player A non-determinism is eliminated completely by propagating predicates back to an initial state. It does this by first selecting an arbitrary shortest play $\hat{\pi}$ from an initial state in $\alpha([\![P]\!])$ to $\langle \ell, \mathbf{b} \rangle$ and by traversing this play from the end to the beginning. Our second procedure, PRECADD, depicted in Figure 5.6, simply propagates predicates backwards over the control-flow graph. It only refines each location once to ensure that the call to PRECADD terminates.

We note that the weakest preconditions we take in these procedures are normal weakest preconditions in Dijkstra's sense (see Section 4.3). However, unlike in our basic refinement steps in Section 5.4.2, in TRACEADD and PRECADD we take the weakest pre-

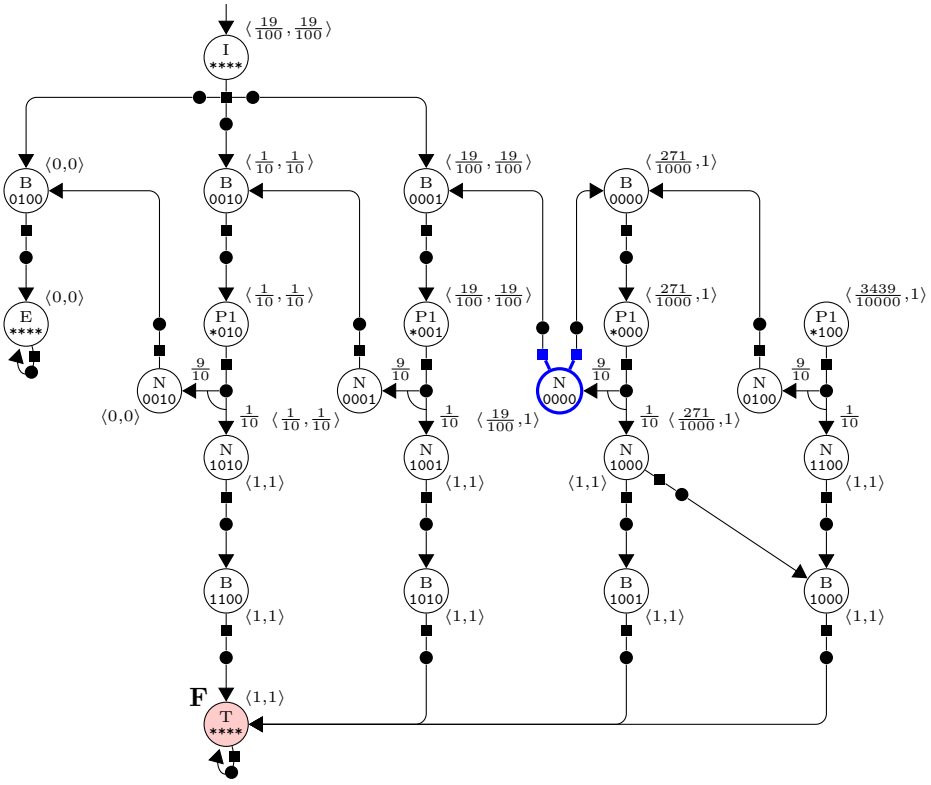


FIGURE 5.8: A predicate abstraction $\hat{G}$ of the program in Figure 4.7 with $Prob^{+}(\hat{G}) = \langle \frac{19}{100}, \frac{19}{100} \rangle$ and with α being induced by predicates $p_1 = (\texttt{f})$, $p_2 = (\texttt{p==0})$, $p_3 = (\texttt{p==1})$ and $p_4 = (\texttt{p==2})$ such that $\text{MAP}(\ell, p_i)$ holds only if $i = 1$ and $\ell \in \{B, N\}$ or if $i \in [2, 4]$ and $\ell \in \{B, P1, N\}$.

condition of arbitrary program statements, potentially including assignments that have non-deterministic or probabilistic semantics. For such assignments we are not always able to compute exact weakest precondition. This is because a weakest precondition of, say, a non-deterministic assignment is existentially quantified and, unfortunately, our method for computing abstractions requires predicates to be free of quantifiers. Moreover, there is no known practical procedure that can eliminate these quantifiers when modelling the semantics of programs in a bit-vector logic [Kro10a]. Therefore, as do non-probabilistic software model checkers,[11] we over-approximate the existentially quantified predicate by removing the existential quantifier as well as any Boolean subformulas that depend on it.

For example, suppose we are interested in computing the weakest precondition of a

[11] Of course, many software model checkers use interpolant-based refinement instead, avoiding the need to take weakest preconditions altogether.

predicate $p = (\texttt{var==5} \wedge \texttt{y} < 100)$ under an assignment $\texttt{var=*}$. Formally, this weakest precondition is

$$\exists \texttt{var} \in \text{TYPE}(\texttt{var}) : (\texttt{var==5} \wedge \texttt{y} < 100) \tag{5.4}$$

and, with the quantifier (i.e. $\exists \texttt{var} \in \text{TYPE}(\texttt{var})$) and the dependent subformula (i.e. $\texttt{var==5}$) removed, the quantifier-free over-approximation of (5.4) is $\texttt{y} < 100$.

An analogous treatment is necessary for assignments with probabilistic semantics. Formally, instead of an existential quantifier, however, the proper mathematical weakest precondition of these assignments involves a summation [MM05]. In practice, though, the over-approximation is done in exactly the same fashion.

We conclude our discussion on refinement with another example.

Example 5.10. *Consider again the refinement step performed in Example 5.8 on the abstraction in Figure 5.3 (page 71) of the program in Figure 4.7 (page 59). We will demonstrate what happens if we call* $\text{TRACEADD}(\langle B, \langle\rangle\rangle, \texttt{f})$ *or* $\text{PRECADD}(\langle B, \langle\rangle\rangle, \texttt{f})$ *instead of* $\text{ADD}(B, \texttt{f})$. *The only shortest play from* $\langle I, \langle\rangle\rangle$ *to* $\langle B, \langle\rangle\rangle$ *is*

$$\hat{\pi} = \langle I, \langle\rangle\rangle \to \{[\langle B, \langle\rangle\rangle]\} \to [\langle B, \langle\rangle\rangle] \to \langle B, \langle\rangle\rangle \ .$$

In the first call of TRACEADDREC *we end up calling* $\text{ADD}(B, \texttt{f})$. *However, taking the weakest precondition of* $\texttt{f}$ *over the control-flow edge* $\langle I, B\rangle$ *results in a trivial predicate due to the assignment* $\texttt{f=0}$. *If we were to call* $\text{PRECADD}(\langle B, \langle\rangle\rangle, \texttt{f})$ *instead, then the same refinement would be made in* B *but we would also propagate the predicate back over the program's loop and call* $\text{ADD}(N, \texttt{f})$. *However, for the control-flow edge* $\langle P1, N\rangle$ *labelled with an assignment* $f = \mathit{coin}(1, 10)$, *as* $\mathit{WP}(\texttt{f=}\theta, \texttt{f})$ *is* θ, *the predicate* $\texttt{f}$ *does not get propagated. For this example,* PRECADD *has performed both the refinement steps of Example 5.8 and 5.9, resulting in the abstraction depicted in Figure 5.2.*

If we now refine the abstraction in Figure 5.2 via the refinable state $\langle B, \langle 0\rangle\rangle$ *and* PRECADD, *then we would call*

$$\text{ADD}(B, \neg\texttt{f} \wedge (\texttt{p==0})),\ \text{ADD}(N, \neg\texttt{f} \wedge (\texttt{p==1}))\ \textit{and}\ \text{ADD}(P1, (\texttt{p==1}))$$

which, in practice, results in enabling predicate `p==0` *in* B *and predicate* `p==1` *in* N *and* $P1$*. The resulting abstraction is depicted in Figure 5.7. Note that due to this refinement step the approximation of* $Prob^{+}([\![P]\!])$ *has improved from an approximation* $\langle 0, 1\rangle$ *to the approximation* $\langle \frac{1}{10}, 1\rangle$*. Through one further refinement step with* PRECADD*, we get the abstraction depicted in Figure 5.8 which yields a precise approximation* $\langle \frac{19}{100}, \frac{19}{100}\rangle$*.*

5.5 Experimental Results & Extensions

To validate the approach described in this chapter we have implemented a model checker for ANSI-C programs. This model checker, called QPROVER, is described in Appendix B.1. We also discuss extensions of our framework, including extensions that we call "*predicate initialisation*" and "*reachable state restriction*".

5.5.1 Experiments & Analysis

We have evaluated our implementation against a wide range of case studies and properties (see Appendix C). Most relevant are the network admin utility (PING), a file transfer protocol client (TFTP) and a network time protocol client (NTP). These programs are not manually crafted models — they are actual code from utilities employed in many Linux distributions. Each program is approximately 1,000 lines of ANSI-C code featuring functions, arrays, pointers, bit-level arithmetic, etc. We have manually adapted these programs with assumptions about the failure rate of kernel calls to IO functions and assumptions on user input (we assume, say, that the user provides a valid hostname). We also adapted parts of the source code that our tool found problematic, but only on parts of the code that do not affect the properties we consider.[12] We also consider an implementation of Freivald's algorithm (FREI) and a sequentialised model of Herman's self-stabilisation protocol (HERM) from APEX [LMOW08]. Finally, we consider various smaller programs from the probabilistic verification literature, including pGCL case studies martingale (MGALE), amplification (AMP) and faulty factorial (FAC) from [MM05],

[12] Most notably we removed calls to `memset`. When inlined, these calls introduce loops that initialise data structures. To prove these loops terminate is a significant burden for our method.

and sequentialised, ANSI-C versions of the bounded retransmission protocol (BRP), IPv4 ZeroConf protocol (ZERO) and the consensus protocol (CONS) from PRISM [HKNP06].

All experiments were run on an Intel Core Duo 2 7200 with 2GB RAM with Fedora 8. We use a timeout setting of 600 seconds for every experiment and we terminate the abstraction-refinement loop whenever absolute error is less than or equal to 10^{-4}. All timings are reported in seconds. In Figure 5.9, we present performance results of the available refinement settings. Available settings are the refinable state selection method (COARSEST/NEAREST) and the predicate propagation method (TRACEADD/PRECADD). In Figure 5.12, for each property, we give detailed statistics for the best configuration of each property.[13] We show more details for two specific runs PING D (3) in Figure 5.10 and NTP A (6) in Figure 5.11.

We remark that the model checker we use to verify game abstractions is an adaptation of the symbolic verification back-end for MDPs in PRISM [Par02, HKNP06]. Due to the verification algorithms used by this model checker the bounds shown in Figure 5.9 are numerical approximations of the actual bounds.

Applicability Firstly, we find it very promising that we can successfully compute quantitative properties of many probabilistic programs with our approach. In particular, we know of no verification tool that is capable of analysing probabilistic properties of programs like PING, TFTP and NTP. All prominent probabilistic verification tools, including PRISM [HKNP06], MRMC [KZH+09], RAPTURE [JDL02], PASS [HWZ08] and APEX [LMOW08], target models in simpler modelling languages and therefore cannot handle the software-specific constructs that appear in these programs. In addition to this, the state space of these programs is far beyond the capabilities of state-of-the-art probabilistic model checkers for finite-state systems such as PRISM [HKNP06] or MRMC [KZH+09].

Overall performance From Figure 5.12 we see that most time is spent computing abstractions or model checking. It is not surprising that model checking is relatively expensive in our setting. We have to employ probabilistic model checking algorithms

[13] The runs we show are COARSEST/TRACEADD for PING A, B, NTP B,C, FREI and MGALE B, COARSEST/PRECADD for MGALE A, AMP A, BRP B and CONS B and NEAREST/PRECADD for the remaining runs.

			TraceAdd								PrecAdd								
			Coarsest				Nearest				Coarsest				Nearest				
			Itr	Pre	Avg	Time	Itr	Pre	Avg	Time	Itr	Pre	Avg	Time	Itr	Pre	Avg	Time	Result
PING	A	1	20	32	5.9	26.3	24	31	5.3	34.0	–	–	–	>600	–	–	–	>600	[0.0792, 0.0792]
	B	1	1	10	2.3	0.82	1	10	2.3	0.83	1	10	2.3	0.82	1	10	2.3	0.81	[0, 0]
	D	1	22	45	9.8	351	24	32	4.9	24.2	12	27	5.7	6.75	13	32	6.2	6.89	[0.9108, 0.9108]
		2	28	43	10.9	520	–	–	–	>600	–	–	–	>600	18	38	8.3	22.2	[0.837936, 0.837936]
		3	–	–	–	>600	–	–	–	>600	–	–	–	>600	22	49	11.6	460	[0.770901, 0.770901]
	C	0	25	30	7.3	81.5	23	22	4.2	23.9	11	15	2.8	3.95	11	15	2.8	3.82	[1.07609, 1.07609]
TFTP	A		–	–	–	>600	–	–	–	>600	25	37	7.4	67.5	25	37	7.4	67.1	[0.993953, 0.993953]
	B		–	–	–	>600	–	–	–	>600	26	53	9.2	147	27	53	9.2	87.3	[0.987943, 0.987943]
	C		–	–	–	>600	–	–	–	>600	23	38	6.9	59.4	22	37	6.9	53.4	[1.77777, 1.77777]
NTP	A	1	–	–	–	>600	–	–	–	>600	–	–	–	>600	16	27	2.9	71.4	[0.08, 0.08]
		2	–	–	–	>600	–	–	–	>600	–	–	–	>600	22	51	4.7	149	[0.1536, 0.1536]
		4	–	–	–	>600	–	–	–	>600	–	–	–	>600	24	55	6.6	260	[0.283607, 0.283607]
		6	–	–	–	>600	–	–	–	>600	–	–	–	>600	26	59	8.5	421	[0.393645, 0.393645]
	B	1	14	34	0.6	10.5	15	34	0.6	12.0	20	48	4.7	77.4	18	31	3.0	13.3	[0.92, 0.92]
		2	17	37	0.8	22.3	19	37	0.8	26.2	21	50	5.6	131	20	49	4.7	53.1	[0.9936, 0.9936]
		4	18	39	0.9	29.0	19	39	0.9	31.2	22	52	6.6	207	22	53	6.6	106	[0.999959,1]
		6	18	39	0.9	30.2	19	39	0.9	30.9	22	52	6.6	205	23	55	7.5	148	[0.999959,1]
	C	1	17	34	0.6	21.1	17	34	0.6	21.0	14	24	2.9	14.0	14	24	2.9	14.1	[1, 1]
		2	20	37	0.8	36.5	21	37	0.8	39.4	19	47	4.7	61.7	19	47	4.7	61.4	[1.08 , 1.08]
		4	21	39	0.9	44.5	21	39	0.9	45.2	21	51	6.6	117	21	51	6.6	117	[1.08691, 1.086955]
		6	21	39	0.9	45.7	21	39	0.9	44.6	23	55	8.5	215	23	55	8.5	215	[1.08691, 1.086955]
PREI	A	1	3	20	3.0	2.92	3	20	3.0	2.82	3	20	3.0	3.70	3	20	3.0	2.73	[0.5, 0.5]
		2	3	20	3.0	5.41	4	21	3.1	7.83	3	20	3.0	5.29	4	21	3.1	7.29	[0.5, 0.5]
		3	3	20	3.0	559	–	–	–	>600	–	–	–	>600	–	–	–	>600	[0.25, 0.25]
	B	1	3	20	3.0	4.22	3	20	3.0	2.78	3	20	3.0	2.74	3	20	3.0	2.70	[0.25, 0.25]
		2	3	20	3.0	5.48	5	23	3.1	9.04	3	20	3.0	5.32	5	23	3.1	8.95	[0.25, 0.25]
		3	3	20	3.0	568	–	–	–	>600	–	–	–	>600	–	–	–	>600	[0.25, 0.25]
HERM	A	3	11	28	8.6	6.38	16	26	7.5	9.18	9	17	6.3	1.64	8	16	5.7	1.38	[1, 1]
		5	17	49	16.2	75.2	35	37	13.0	172	14	23	8.8	9.75	12	22	8.2	8.56	[1, 1]
		7	–	–	–	>600	–	–	–	>600	20	29	11.7	76.6	18	28	11.2	74.1	[1, 1]
	C	3	1	0	0.0	0.12	1	0	0.0	0.04	1	0	0.0	0.04	1	0	0.0	0.04	[0, 0]
		5	1	0	0.0	0.12	1	0	0.0	0.04	1	0	0.0	0.06	1	0	0.0	0.05	[0, 0]
		7	1	0	0.0	0.04	1	0	0.0	0.08	1	0	0.0	0.14	1	0	0.0	0.04	[0, 0]
MGALE	A	10	8	19	4.0	30.2	8	19	4.0	29.2	6	17	3.3	1.95	6	17	3.3	2.15	[0.125, 0.125]
		100	11	28	5.8	108	11	28	5.8	108	9	26	5.2	10.7	9	26	5.2	12.2	[0.015625, 0.015625]
		1,000	–	–	–	>600	–	–	–	>600	12	35	7.1	186	12	35	7.1	187	[0.00195312, 0.00195312]
	B	10	3	3	0.4	0.07	3	3	0.4	0.07	5	13	2.5	0.36	5	13	2.5	0.56	[0.999998, 1]
		100	3	3	0.4	0.07	3	3	0.4	0.07	8	22	4.4	7.79	8	22	4.4	6.71	[0.999998, 1]
		1,000	3	3	0.4	0.07	3	3	0.4	0.07	11	31	6.2	221	11	31	6.2	223	[0.999998, 1]
AMP	A	20	42	24	13.9	7.31	44	24	13.9	7.07	24	25	14.2	2.02	24	25	14.2	2.16	[0.996829, 0.996829]
		40	82	44	26.4	39.7	84	44	26.4	37.7	44	45	26.7	9.18	44	45	26.7	9.50	[0.99999, 0.99999]
		60	122	64	38.9	143	124	64	38.9	143	64	65	39.2	26.2	64	65	39.2	27.7	[1, 1]
	C	20	22	24	12.7	1.98	24	24	12.6	1.90	24	25	14.2	1.82	24	25	14.2	1.81	[0.00317121, 0.00317121]
		40	34	36	19.4	6.99	36	36	19.4	5.69	36	37	21.7	5.30	36	37	21.7	5.17	[0, 7.53393e-05]
		60	34	36	19.4	5.56	36	36	19.4	5.73	36	37	21.7	6.32	36	37	21.7	5.25	[0, 7.53393e-05]
FAC	A	20	43	22	4.5	5.40	43	22	4.5	4.90	23	23	4.5	2.98	23	23	4.5	2.82	[1, 1]
		40	83	42	8.8	33.4	83	42	8.8	35.1	43	43	8.8	19.4	43	43	8.8	18.4	[1, 1]
		60	123	62	13.1	121	123	62	13.1	133	63	63	13.1	76.2	63	63	13.1	73.9	[1, 1]
	B	20	43	22	5.1	13.1	43	22	5.1	14.5	23	23	5.1	9.38	23	23	5.1	7.84	[20.202, 20.202]
		40	83	42	9.9	107	83	42	9.9	119	43	43	9.9	64.7	43	43	9.9	63.9	[40.404, 40.404]
		60	123	62	14.7	463	123	62	14.7	481	63	63	14.8	274	63	63	14.8	257	[60.606, 60.606]
BRP	A	16	5	9	1.9	0.30	6	8	1.7	0.34	6	10	4.1	0.52	5	8	3.3	0.38	[0, 8e-06]
		32	5	9	1.9	0.30	6	8	1.7	0.34	6	10	4.1	0.52	5	8	3.3	0.40	[0, 8e-06]
		64	5	9	1.9	0.30	6	8	1.7	0.35	6	10	4.1	0.51	5	8	3.3	0.38	[0, 8e-06]
	B	16	–	–	–	>600	–	–	–	>600	9	9	3.7	0.90	12	14	6.4	1.64	[0, 2.64636e-05]
		32	–	–	–	>600	–	–	–	>600	9	9	3.7	0.88	12	14	6.4	1.66	[0, 2.64636e-05]
		64	–	–	–	>600	–	–	–	>600	9	9	3.7	0.87	12	14	6.4	1.66	[0, 2.64636e-05]
ZERO	A	10	–	–	–	>600	–	–	–	>600	15	16	7.2	7.52	15	16	7.2	7.26	[0.908688, 0.908688]
		30	–	–	–	>600	–	–	–	>600	35	36	16.5	66.7	35	36	16.5	64.7	[0.92405, 0.92405]
		50	–	–	–	>600	–	–	–	>600	55	56	25.9	245	55	56	25.9	238	[0.937004, 0.937004]
	B	10	47	15	6.4	19.8	48	15	6.4	29.7	17	16	7.1	6.09	17	16	7.1	5.11	[10.5099, 10.5099]
		30	107	35	15.6	339	109	35	15.6	572	37	36	16.6	88.8	37	36	16.6	84.5	[34.5093, 34.5093]
		50	–	–	–	>600	–	–	–	>600	57	56	26.2	444	57	56	26.2	443	[62.9048, 62.9048]
CONS	A	2	32	38	13.4	292	36	31	10.2	74.4	26	42	12.6	39.6	26	39	11.7	43.9	[0.999996, 0.999998]
		3	32	39	13.5	134	41	33	11.9	149	30	44	14.6	78.3	28	41	13.5	89.8	[0.999991, 0.999999]
		4	39	39	15.2	541	47	35	13.4	303	33	47	16.5	165	29	43	15.2	125	[0.999984, 0.999995]
	B	2	40	38	12.9	255	39	32	11.6	131	25	31	10.8	44.7	25	33	10.8	50.7	[0.499999, 0.499999]
		3	42	38	13.9	348	41	34	13.1	251	27	33	12.6	114	27	35	12.6	128	[0.499999, 0.499999]
		4	–	–	–	>600	–	–	–	>600	29	35	14.4	216	29	37	14.4	257	[0.5, 0.5]

FIGURE 5.9: We give, for each combination of COARSEST/NEAREST and TRACEADD/PRECADD, the number of abstraction-refinement iterations (ITR), the total number of predicates (PRE), the average number of predicates per control-flow location (AVG) and the total time required for verification (TIME). A time of ">600" indicates a timeout. We also show the final bounds (RESULT). Because each configuration may yield different bounds we depict the bounds of the leftmost configuration that terminates.

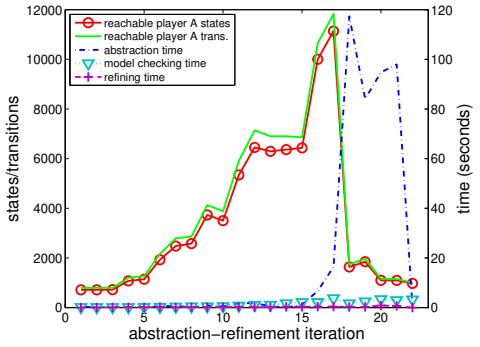


FIGURE 5.10: Statistics of the PING D (3) experiment using NEAREST and PRECADD.

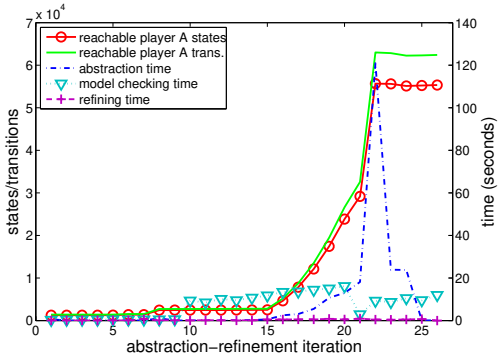


FIGURE 5.11: Statistics of the NTP A (6) experiment using NEAREST and PRECADD.

twice per iteration — once for each bound. However, in practice, we do observe that the model checking time is relatively consistent between different abstraction-refinement iterations, whereas the abstraction time is quite unpredictable (see Figure 5.10 and 5.11). This is because we compute abstractions incrementally — if the localisation mapping is not updated for some parts of the control-flow then we do not recompute the abstraction here.

Properties for which abstraction is very expensive include PING D, FREI and MGALE. For PING D we will see (in Section 5.5.3) that a lot of time is wasted computing the transition function for abstract states that are unreachable. For FREI, abstraction is expensive because there is a non-deterministic choice for which the number of functions required to satisfy Assumption 5.1 is exponential in the parameter value. MGALE is expensive to abstract because it contains non-linear arithmetic. Although the same is true for FAC, say, the problem for MGALE is that the non-linear arithmetic is also present in the predicates, making the SAT instances more challenging to solve.

Game abstractions An observation we make from Figure 5.12 is that the number of player A transitions is usually not much higher than the number of player A states in the final abstraction. This means that the final game abstractions do not contain much player A non-determinism and, as such, are not far removed from being probabilistically bisimular to the program. In fact, the amount of player A non-determinism is typically quite limited throughout the abstraction-refinement loop (see Figure 5.10 and 5.11).

			ITR	PRE	AVG	STA	TRA	ABS	CHK	REF	TIME
PING	A	1	20	32	5.9	588	657	23%	61%	14%	26.3
	B	1	1	10	2.3	902	1039	46%	11%	0%	0.81
	D	1	13	32	6.2	359	375	32%	46%	16%	6.89
		2	18	38	8.3	524	536	50%	36%	11%	22.2
		3	22	49	11.6	974	1003	92%	6%	1%	460
	C	0	11	15	2.8	256	262	11%	68%	14%	3.82
TFTP	A		25	37	7.4	15984	16328	28%	63%	7%	67.1
	B		27	53	9.2	18646	19048	37%	52%	9%	87.3
	C		22	37	6.9	5884	6093	15%	76%	7%	53.4
NTP	A	1	16	27	2.9	2501	2769	7%	89%	2%	71.4
		2	22	51	4.7	10551	11877	34%	62%	3%	149
		4	24	55	6.6	28875	32552	44%	52%	2%	260
		6	26	59	8.5	55343	62415	53%	44%	1%	421
	B	1	14	34	0.6	533	563	19%	65%	12%	10.5
		2	17	37	0.8	1448	1528	16%	68%	13%	22.3
		4	18	39	0.9	2481	2606	20%	67%	11%	29.0
		6	18	39	0.9	2481	2606	22%	65%	10%	30.2
	C	1	17	34	0.6	537	567	12%	76%	10%	21.1
		2	20	37	0.8	1472	1552	10%	79%	9%	36.5
		4	21	39	0.9	2537	2662	14%	76%	8%	44.5
		6	21	39	0.9	2537	2662	16%	73%	8%	45.7
FREI	A	1	3	20	3.0	1658	1673	64%	17%	3%	2.92
		2	3	20	3.0	1724	1748	81%	8%	2%	5.41
		3	3	20	3.0	1724	1748	99%	0%	0%	559
	B	1	3	20	3.0	1658	1673	51%	30%	7%	4.22
		2	3	20	3.0	1724	1748	81%	8%	2%	5.48
		3	3	20	3.0	1724	1748	99%	0%	0%	568
HERM	A	3	16	16	5.7	388	402	27%	67%	4%	1.38
		5	24	22	8.2	2506	2626	16%	82%	1%	8.56
		7	18	28	11.2	11832	12152	13%	86%	0%	74.1
	C	3	1	0	0.0	40	48	44%	13%	0%	0.04
		5	1	0	0.0	40	48	40%	10%	0%	0.05
		7	1	0	0.0	40	48	45%	14%	0%	0.04
MGALE	A	10	6	17	3.3	55	55	91%	4%	2%	1.95
		100	9	26	5.2	85	85	95%	3%	1%	10.7
		1,000	12	35	7.1	115	115	98%	0%	0%	186
	B	10	5	13	2.5	147	168	64%	16%	12%	0.36
		100	8	22	4.4	1155	1344	94%	3%	1%	7.79
		1,000	11	31	6.2	9219	10752	99%	0%	0%	221
AMP	A	20	24	25	14.2	528	548	17%	72%	8%	2.02
		40	44	45	26.7	1028	1068	14%	80%	4%	9.18
		60	64	65	39.2	1528	1588	11%	85%	3%	26.2
	C	20	24	25	14.2	528	548	19%	69%	9%	1.81
		40	36	37	21.7	877	914	15%	78%	5%	5.17
		60	36	37	21.7	877	914	15%	78%	5%	5.25
FAC	A	20	23	23	4.5	162	162	22%	72%	3%	2.82
		40	43	43	8.8	302	302	16%	81%	1%	18.4
		60	63	63	13.1	442	442	14%	84%	0%	73.9
	B	20	23	23	5.1	182	182	9%	88%	1%	7.84
		40	43	43	9.9	342	342	5%	93%	0%	63.9
		60	63	63	14.8	502	502	4%	95%	0%	257
BRP	A	16	5	8	3.3	369	450	32%	38%	12%	0.38
		32	5	8	3.3	369	450	31%	37%	11%	0.40
		64	5	8	3.3	369	450	32%	38%	11%	0.38
	B	16	9	9	3.7	386	456	21%	55%	17%	0.90
		32	9	9	3.7	386	456	21%	54%	16%	0.88
		64	9	9	3.7	386	456	22%	55%	16%	0.87
ZERO	A	10	15	16	7.2	453	453	3%	95%	1%	7.26
		30	35	36	16.5	1273	1273	3%	95%	0%	64.7
		50	55	56	25.9	2093	2093	3%	96%	0%	238
	B	10	17	16	7.1	482	482	6%	89%	3%	5.11
		30	37	36	16.6	1382	1382	3%	95%	0%	84.5
		50	57	56	26.2	2282	2282	2%	97%	0%	443
CONS	A	2	26	39	11.7	467	474	7%	87%	5%	43.9
		3	28	41	13.5	711	721	4%	91%	3%	89.8
		4	29	43	15.2	958	970	4%	92%	2%	125
	B	2	25	31	10.8	406	408	5%	89%	4%	44.7
		3	27	33	12.6	608	610	2%	93%	3%	114
		5	29	35	14.4	810	812	2%	95%	2%	216

FIGURE 5.12: For the best refinement configuration, we show (ITR), (PRE) and (AVG) as before. We also show the number of states (STA) and player A transitions (TRA) in the final abstraction and the percentage of the total time (TIME) spent abstracting (ABS), model checking (CHK) and refining (REF), respectively.

Previously, we explained that in the abstraction-refinement loop we consider increasingly precise games. In general, at least when measured in terms of player A states and transitions in this game, this is indeed the case. An exception to this is PING. In Figure 5.10 we see that the games considered half-way through the abstraction-refinement loop have much larger state spaces than the final or initial abstraction. This is because, in our implementation, we only consider *reachable* states of games and, as the predicates make the abstractions more precise, we are able to discard many abstract states as being unreachable. The steady increase in the size of the state space we see in Figure 5.11 is more representative of the remainder of the case studies.

Divergence In most cases, the timeouts in Figure 5.9 are due to scalability issues that occur during abstraction and model checking. For properties PING A, B & C, however, for some configurations, our refinement method *diverges*. It is well-known that refinement methods based on weakest preconditions may diverge [JM06].

In fact, for PING we presented better results in [KKNP09]. These experiments were performed with a different version of our tool and we were fortunate that, in this version, we chose refinable states in such a way that divergence did not occur as often. We deliberately use the current results in order to show that, for some programs, our refinement methodology can be relatively fragile — i.e. small unrelated changes to the abstraction-refinement process can make the difference between a terminating run and divergence. We have not observed divergence for other case studies.

Loops Except for FREI, all probabilistic programs we have considered contain control-flow loops and, with the further exception of HERM and MGALE, the parameter value we give in Fig. 5.9 and 5.12 corresponds directly to the number of loop iterations of the main program loop.[14] We observe that e.g. NTP A, AMP A, ZERO A & B, the verification time is roughly exponential in the number of loop iterations. This exponential behaviour occurs whenever each loop iteration has to be included in the abstract model. That the computational cost of abstraction rises exponentially in this instance is not atypical of predicate abstraction — computing precise transition relations for non-probabilistic

[14] For MGALE the number of loop iterations is logarithmic in the parameter.

abstractions in CEGAR is already known to be exponential in the number of predicates in practice [KS06]. This problem is accentuated for larger programs such as, say, NTP, because more predicates are required per loop iteration, increasing the cost of abstraction and model checking more quickly.

Fortunately, there are many programs and properties for which we do not need to consider all loop iterations of the concrete model. This is because we are willing to settle for approximations — i.e. intervals with an absolute error of 10^{-4} or less — and often such an approximation can be established by analysing just a few loop iterations. We observe this behaviour in AMP C, BRP A & B. For these models the probability of satisfying the property is close to 0. The abstractions we find for these models contain only the last few loop iterations and, independently of what happens before we get to these last iterations, the abstraction of the last few iterations enables us to establish that the probability of reaching the target location is below 10^{-4}.

For NTP B, we observe dual behaviour. The probability of this property is very close to 1 for large parameters. For this case, the abstraction we find contains the *first* few loop iterations and is able to show that at least $1 - 10^{-4}$ of the total probability mass always reaches the target state. This case happens less frequently because of the nature of our refinements. We always add predicates by going backwards from the source of the error. In practice this means that usually the predicates describe the last few iterations of the loop instead of the first few. However, in the case of NTP the predicates are general enough to also describe the first few iterations. We observe similar behaviour for a cost property in NTP C.

In MGALE B, where we are interested in the probability with which the program terminates, a good abstraction only needs to consider *one* loop iteration. In this program a "gambler" continuously gambles some money and, with some probabilistic choice, he either wins or loses the bet. The program terminates as soon as the gambler wins or when the gambler runs out of money. With TRACEADD we find predicates with which we can establish that, even if the gambler has enough funds to bet infinitely many times, the gambler will eventually win (and the program will eventually terminate) with probability 1. That is, even though the program always terminates in the qualitative sense (there are

no non-terminating paths) we show the program almost surely terminates (the probability of termination is 1) with a very compact abstraction. With PrecAdd, we do not find this abstraction and unwind the loop instead.

For AMP C; BRP A, B; NTP B, C and MGALE B, after a certain parameter value, the verification cost no longer increases and, in principle, we can verify these properties with arbitrarily large parameters. For example, we can run NTP C on parameter $10,000$ without any added cost. Of course, if we change the termination criterion of our abstraction-refinement loop then this will affect the number of loop iterations we need to take into consideration.

Refinable state selection Generally, both Nearest and Coarsest perform similarly. We often see that the choice between Nearest and Coarsest has more impact when using TraceAdd (see PING C and D, HERM A, B and, D). This is because often the set of available refinable states share a common control-flow location. Whereas, for TraceAdd, a minute difference in the data component of a refinable state can mean we propagate predicates over a completely different play, for PrecAdd, the predicate propagation is independent of the data component. Hence, if most refinable states share a common control-flow location then the selection of such a refinable state has more impact on TraceAdd than PrecAdd. When refinable states from various control-flow locations are available, as in NTP B, FREI and BRP, we see that the choice between Coarsest and Nearest also has significant impact for PrecAdd.

Predicate propagation For FREI, we see that the choice between TraceAdd and PrecAdd has little effect on the performance of the abstraction-refinement loop. This is because FREI has no program loops and, as a result, the two propagation methods behave equivalently. For TFTP, HERM A, MGALE A, AMP A, FAC A & B, BRP B, ZERO A & B, CONS A & B we observe that TraceAdd is slower than PrecAdd. The underlying cause of this is that TraceAdd generally adds fewer predicates in each refinement step and — as long as the predicates found by PrecAdd are necessary and will be found at a later stage by TraceAdd — TraceAdd generally needs more abstraction-refinement iterations to obtain the same abstraction (this is exemplified by Example 5.10).

			SATABS						QPROVER						TOTAL	SPEED
			ITR	PRE	AVG	ABS	CHK	REF	ITR	PRE	AVG	ABS	CHK	REF	TIME	UP
PING	A	1	3	21	7.8	8%	35%	2%	3	37	7.8	16%	22%	1%	7.27	3.62
		2	5	35	10.5	9%	43%	5%	1	43	10.5	21%	10%	0%	14.1	>42.58
		3	5	39	11.9	3%	16%	1%	2	48	12.9	54%	17%	1%	45.7	>13.13
	B	1	6	23	11.6	5%	34%	3%	1	42	11.6	40%	6%	0%	16.1	0.05
	D	1	3	27	8.3	7%	36%	4%	2	38	9.0	21%	14%	2%	8.81	0.78
		2	5	38	12.6	3%	18%	3%	3	48	13.3	38%	28%	1%	31.0	0.71
		3	5	34	12.6	0%	3%	0%	8	51	15.1	32%	60%	0%	167	2.76
	C	0	3	9	3.8	4%	22%	0%	7	24	4.5	11%	46%	5%	7.20	0.53
TFTP	A		3	24	5.9	3%	8%	0%	13	40	9.3	24%	45%	5%	56.9	1.18
	B		5	39	9.9	3%	10%	4%	12	57	12.5	10%	58%	2%	151	0.58
	C		3	16	2.0	1%	6%	0%	17	40	8.1	14%	63%	5%	67.4	0.79
FREI	A	1	1	6	0.0	9%	14%	0%	3	20	3.0	47%	10%	2%	4.08	0.72
		2	1	6	0.0	4%	1%	0%	3	20	3.0	76%	8%	1%	5.79	0.93
	B	1	1	6	0.0	7%	2%	0%	3	20	3.0	60%	12%	3%	3.22	1.31
		2	1	6	0.0	4%	1%	0%	3	20	3.0	76%	7%	2%	5.74	0.95
HERM	A	3	11	19	9.2	4%	40%	19%	2	24	9.2	13%	13%	0%	1.75	0.79
		5	24	31	14.0	1%	35%	20%	2	36	14.0	12%	27%	0%	13.4	0.64
		7	27	43	18.9	0%	7%	6%	1	48	18.9	16%	68%	0%	84.4	0.88
MGALE	A	10	16	20	5.3	0%	0%	0%	1	26	5.3	99%	0%	0%	563	0.00
		100	–	–	–	–	–	–	–	–	–	–	–	–	>600	<0.02
		1,000	–	–	–	–	–	–	–	–	–	–	–	–	>600	<0.31
AMP	A	20	3	5	2.3	0%	2%	0%	20	25	14.2	9%	82%	4%	3.94	0.51
		40	3	5	2.3	0%	0%	0%	40	45	26.7	6%	90%	2%	20.9	0.44
		60	3	5	2.3	0%	0%	0%	60	65	39.2	4%	93%	1%	68.6	0.38
	C	20	21	23	13.6	5%	27%	55%	2	25	14.2	1%	5%	0%	6.52	0.28
		40	41	43	26.1	3%	20%	73%	1	44	26.1	0%	1%	0%	38.2	0.14
		60	61	63	38.6	2%	26%	69%	1	64	38.6	0%	0%	0%	140	0.04
FAC	A	20	39	21	4.3	4%	15%	73%	2	23	4.5	0%	2%	0%	17.3	0.16
		40	79	41	8.6	2%	9%	85%	2	43	8.8	0%	1%	0%	128	0.14
		60	100	51	10.8	1%	8%	75%	12	63	13.1	1%	10%	0%	305	0.24
	B	20	39	21	4.9	4%	15%	64%	2	23	5.1	1%	11%	0%	19.2	0.41
		40	79	41	9.7	2%	9%	78%	2	43	9.9	0%	7%	0%	141	0.45
		60	100	51	12.1	1%	6%	57%	12	63	14.8	1%	33%	0%	415	0.62
ZERO	A	10	12	14	6.8	1%	5%	10%	2	16	7.2	0%	80%	0%	22.3	0.33
		30	32	34	16.1	0%	3%	29%	2	36	16.6	0%	65%	0%	201	0.32
		50	–	–	–	–	–	–	–	–	–	–	–	–	>600	<0.40
	B	10	13	13	6.2	2%	12%	21%	3	16	7.1	1%	59%	0%	9.22	0.55
		30	33	33	15.7	1%	4%	39%	3	36	16.7	0%	53%	0%	194	0.44
		50	–	–	–	–	–	–	–	–	–	–	–	–	>600	<0.74
CONS	A	2	21	30	17.1	2%	42%	15%	1	47	17.1	1%	35%	0%	32.6	1.34
		3	29	33	19.8	1%	28%	15%	1	50	19.8	0%	51%	0%	76.5	1.17
		4	31	37	23.3	1%	17%	12%	1	54	23.3	0%	68%	0%	149	0.84
	B	2	18	29	16.2	2%	36%	16%	1	45	16.2	1%	39%	0%	23.7	1.89
		3	20	31	18.0	0%	15%	6%	2	48	18.9	1%	73%	0%	71.0	1.60
		5	21	33	19.8	0%	6%	3%	3	51	21.5	1%	87%	0%	195	1.11

FIGURE 5.13: Results for predicate initialisation. We show the number of abstraction-refinement iterations (ITR), predicates (PRE) and the average number of predicates enabled per control-flow location (AVG) after both the qualitative (SATABS) and quantitative (QPROVER) abstraction-refinement loop. We also show the time spent abstracting (ABS), model checking (CHK) and refining (REF) in both tools as percentages of the total time (TOTAL TIME). Finally we show the speed-up factor (SPEED UP) relative to the best run in Figure 5.9.

For NTP B, C, we see that PRECADD is much slower than TRACEADD. This is because PRECADD propagates predicates too greedily. In particular, it propagates predicates backwards through some irrelevant part of the program (it adds predicates to locations that are not reachable from the initial states without going through the target location).

5.5.2 Predicate Initialisation

In this section, we describe and experimentally evaluate an extension of our approach called *predicate initialisation*. The idea of this approach is to "initialise" the predicate set when we verify a property of a probabilistic program P by first establishing, with a non-probabilistic CEGAR implementation, whether $Reach^{+}([\![P]\!])$ holds. That is, whether there is a path in $[\![P]\!]$ from the initial location to a target location. Of course, we first replace any probabilistic choice in P with a non-deterministic choice.

For most properties we check, these counter-examples exist. This means there is very little we can guarantee about the predicates used by CEGAR. However, the general idea is that hopefully many of the predicates found by CEGAR are needed to verify our probabilistic properties. We also hope that these predicates are found more efficiently with CEGAR than with our quantitative approach. The main idea is therefore to run CEGAR first, to extract the predicates and localisation mapping from the final CEGAR abstraction, and to use these predicates to initialise our abstraction-refinement loop.

Experimental results We have implemented the predicate initialisation method with the CEGAR model checker SATABS [CKSY05]. We ran this implementation on the runs of Figure 5.9. We do not include HERM C, MGALE B and BRP as these properties can already be verified instantaneously. We also do not include NTP because SATABS was not able to verify this program. The results are presented in Figure 5.13.

Through predicate initialisation, we are now able to verify PING A for larger parameter values that previously failed due to divergence. Moreover, in PING D, predicate initialisation improves overall performance by a factor of 2.76. In general, however, even though SATABS is typically quite fast, applying predicate initialisation does not help.

We observe that the ratio of abstraction-refinement loops performed by SATABS and QPROVER, respectively, is extremely sensitive to the property that is being verified. For AMP A, a target location can be reached via many paths — including many short ones — and SATABS quickly finds a short counter-example. In contrast, for AMP B, the only path to the target is one that goes through all loop iterations of the program. We see that SATABS is much faster for AMP A than for AMP B, but, importantly, it does not

actually discover many predicates in doing so — in contrast to AMP B, using predicate initialisation still leaves a lot of work for QPROVER.

We see a general trend that, as the parameter values increase, a proportionally smaller amount of time is spent running SATABS. However, we find that the average number of predicates increases through predicate initialisation which in turn adversely affects the performance of QPROVER. Unfortunately, the added cost for QPROVER seems to overshadow the time savings we get from running SATABS, meaning that in general it does not pay to use predicate initialisation.

5.5.3 Reachable State Restriction

We finally describe an optimisation of the abstraction process called "*reachable state restriction*". This optimisation is especially important because it can be applied to almost any abstraction-refinement implementation.

Suppose we are abstracting a probabilistic program P via an abstraction function $\alpha : \mathscr{L} \times \mathscr{U}_{Var} \to \mathscr{L} \times \mathbb{B}^n$ defined though predicates, *Pred*, and a localisation mapping, MAP. Because of the nature of probabilistic model checking methods, before we evaluate a property on $\alpha([\![P]\!])$, we first perform a reachability analysis on $\alpha([\![P]\!])$'s state space. That is, we compute the set of reachable predicate valuations $\mathit{Reach}_\ell \subseteq \mathbb{B}^n$ for each control-flow location $\ell \in \mathscr{L}$. Formally, we have that $\mathbf{b} \in \mathit{Reach}_\ell$ if and only if there is a play in $\alpha([\![P]\!])$ from an initial state to $\langle \ell, \mathbf{b} \rangle$.

The key idea is to feed this reachability information back to our abstraction procedure such that, after the refinement step, when employing SAT to construct the abstraction under a refined abstraction function $\alpha^\#$, we avoid enumerating some of the transitions that are not reachable in $\alpha^\#([\![P]\!])$. We realise this by augmenting the SAT formulas in Section 5.3.2 with a restriction on source states — i.e. when we abstract a location $\ell \in \mathscr{L}$ we add the constraint that $\alpha_\ell(u) \in \mathit{Reach}_\ell$.[15] We may still enumerate some transitions that are unreachable in $\alpha^\#([\![P]\!])$ as the reachability information has come from a previous, less precise abstraction, $\alpha([\![P]\!])$.

[15] Note that we use α_ℓ and *not* $\alpha^\#_\ell$.

			NORMAL				OPTIMISED				SPEED
			ABS	CHK	REF	TIME	ABS	CHK	REF	TIME	UP
PING	A	1	23%	61%	14%	26.3	14%	63%	21%	18.0	1.46
		2	–	–	–	>600	14%	82%	3%	444	>1.35
	B	1	46%	11%	0%	0.81	43%	14%	0%	0.85	0.96
	D	1	32%	46%	16%	6.89	24%	55%	16%	8.10	0.85
		2	50%	36%	11%	22.2	30%	50%	16%	14.2	1.56
		3	92%	6%	1%	460	45%	44%	9%	45.8	10.04
		4	–	–	–	>600	52%	39%	7%	106	>5.67
		6	–	–	–	>600	60%	36%	2%	459	>1.31
	C	0	11%	68%	14%	3.82	11%	71%	12%	4.37	0.87
TFTP	A		28%	63%	7%	67.1	18%	72%	8%	61.1	1.10
	B		37%	52%	9%	87.3	32%	57%	9%	85.5	1.02
	C		15%	76%	7%	53.4	16%	75%	7%	54.5	0.98
NTP	A	1	7%	89%	2%	71.4	6%	89%	2%	73.3	0.97
		2	34%	62%	3%	149	26%	69%	3%	136	1.10
		4	44%	52%	2%	260	33%	63%	2%	217	1.20
		6	53%	44%	1%	421	41%	56%	2%	330	1.28
	B	1	19%	65%	12%	10.5	24%	51%	20%	7.37	1.42
		2	16%	68%	13%	22.3	20%	57%	19%	15.1	1.48
		4	20%	67%	11%	29.0	21%	58%	18%	18.4	1.58
		6	22%	65%	10%	30.2	20%	59%	18%	18.6	1.62
	C	1	12%	76%	10%	21.1	13%	71%	14%	16.4	1.28
		2	10%	79%	9%	36.5	11%	75%	11%	28.7	1.27
		4	14%	76%	8%	44.5	12%	74%	12%	33.6	1.33
		6	16%	73%	8%	45.7	12%	75%	10%	34.9	1.31
FREI	A	1	64%	17%	3%	2.92	64%	16%	3%	2.84	1.03
		2	81%	8%	2%	5.41	80%	9%	2%	5.31	1.02
		3	99%	0%	0%	559	99%	0%	0%	559	1.00
	B	1	51%	30%	7%	4.22	65%	16%	3%	2.89	1.46
		2	81%	8%	2%	5.48	80%	9%	2%	5.24	1.04
		3	99%	0%	0%	568	99%	0%	0%	557	1.02
HERM	A	3	27%	67%	4%	1.38	26%	67%	4%	1.36	1.01
		5	16%	82%	1%	8.56	18%	80%	1%	8.58	1.00
		7	13%	86%	0%	74.1	14%	85%	0%	117	0.63
MGALE	A	10	91%	4%	2%	1.95	59%	23%	10%	0.46	4.26
		100	95%	3%	1%	10.7	73%	15%	6%	1.79	5.97
		1,000	98%	0%	0%	186	83%	11%	3%	5.64	32.99
AMP	A	20	17%	72%	8%	2.02	23%	68%	6%	2.30	0.88
		40	14%	80%	4%	9.18	22%	73%	3%	10.9	0.85
		60	11%	85%	3%	26.2	21%	75%	2%	29.5	0.89
	C	20	19%	69%	9%	1.81	21%	69%	7%	2.10	0.86
		40	15%	78%	5%	5.17	19%	75%	4%	7.33	0.71
		60	15%	78%	5%	5.25	19%	75%	5%	5.82	0.90
FAC	A	20	22%	72%	3%	2.82	21%	73%	4%	2.71	1.04
		40	16%	81%	1%	18.4	16%	81%	1%	19.4	0.95
		60	14%	84%	0%	73.9	14%	84%	0%	73.7	1.00
	B	20	9%	88%	1%	7.84	8%	89%	1%	7.89	0.99
		40	5%	93%	0%	63.9	5%	94%	0%	64.8	0.99
		60	4%	95%	0%	257	4%	95%	0%	259	0.99
ZERO	A	10	3%	95%	1%	7.26	7%	91%	1%	7.98	0.91
		30	3%	95%	0%	64.7	6%	92%	0%	67.4	0.96
		50	3%	96%	0%	238	6%	93%	0%	243	0.98
	B	10	6%	89%	3%	5.11	10%	85%	2%	5.72	0.89
		30	3%	95%	0%	84.5	6%	92%	0%	93.6	0.90
		50	2%	97%	0%	443	4%	95%	0%	449	0.99
CONS	A	2	7%	87%	5%	43.9	10%	84%	4%	39.2	1.12
		3	4%	91%	3%	89.8	6%	90%	2%	87.5	1.03
		4	4%	92%	2%	125	6%	90%	2%	115	1.08
	B	2	5%	89%	4%	44.7	9%	85%	5%	31.6	1.41
		3	2%	93%	3%	114	5%	91%	3%	76.9	1.48
		5	2%	95%	2%	216	3%	94%	2%	165	1.31

FIGURE 5.14: Results for reachable state restriction. We show the num. of abstraction-refinement iterations (ITR), number of predicates (PRE), average number of predicates enabled per control-flow location (AVG) and the total time (TIME) for both the best run in Figure 5.9 (NORMAL) and the same run with reachable state restriction (OPTIMISED). We also show the speed-up factor (SPEED UP).

Experimental results We have evaluated the reachable state restriction optimisation on each run in Figure 5.9 that is not already instantaneous and present the results in Figure 5.14. We observe our optimisation has a small overhead caused by the extraction of the reachability information and processing it to a form we can encode it SAT. For abstractions where only a small number of abstract states are unreachable, such as HERM A, AMP and FAC, this means that employing our optimisation introduces a slight overhead.

For larger programs, such as PING, TFTP and NTP, we observe both modest speed-ups and a ten-fold speed-up for PING D. What is promising is that we managed to check PING A and D for much larger parameters with this optimisation enabled. That the abstractions we consider for PING indeed contain many unreachable states is exemplified by Figure 5.10. This figure shows that lots of unreachable states are pruned late in the abstraction-refinement loop. We remark that the spike in abstraction times shown in Figure 5.10 is almost entirely due to the abstraction of unreachable states.

Another model where our optimisation works well is MGALE. We observe a speed-up factor of nearly 33 for this model. This large speed-up is possible because, for this model, the abstraction process is very expensive due to non-linear arithmetic found in the predicates. We believe our experimental results justify enabling this optimisation by default — at typically very little cost, we have the potential for great time savings.

5.6 Conclusions

In this chapter, we have presented an approach for computing quantitative properties of probabilistic software based on automated abstraction refinement. Our approach employs the game abstractions introduced in [KNP06] as abstractions of probabilistic programs. We described how to construct predicate abstractions of probabilistic programs via SAT-based methods, akin to [LBC03, CKSY04]. We also discussed a refinement procedure for game abstractions induced by predicates, a number of heuristics for this refinement procedure and two extensions of our technique.

We evaluated our approach on a wide range of case studies. Notably, we demon-

strated that our approach is capable of computing quantitative properties of *real* network clients comprising 1,000 lines of complex ANSI-C code. The state space of these network programs is far beyond what current finite-state probabilistic model checkers can handle. Moreover, these programs feature too many software-specific constructs to be verified by quantitative verification techniques such as [DJJL01, JDL02, HWZ08, WZ10, LMOW08].

We observed that the optimisation called "reachable state restriction" can yield very significant time savings at very little overhead. We remark that this optimisation is not specific to our approach and can be employed in any abstraction-refinement loop. However, the additional advantage we have in our implementation is that reachability information is readily available at no extra cost.

We conclude this chapter by suggesting directions for future research.

Model checking Firstly, we note that the model checking phase of our abstraction-refinement loop is often quite slow. There are many ways in which we could try and improve upon this. Firstly, as we see in Figure 5.12, the game abstractions we verify are often quite small. We therefore argue that an explicit-state solver for games may perform better than our symbolic model checker. Another argument for using an explicit-state method is that, for symbolic data structures to be effective, there needs to be some regularity in the models that are being verified. In practice, however, there is very little structure in the abstract transition functions generated by SAT.

Further time savings could be made by taking into account that probabilistic and non-deterministic choices typically occur very sparsely in game abstractions. To improve the efficiency of the model checker we think it could be beneficial to shorten long plays of deterministic transitions prior to model checking, as is done in [DJJL02].

Abstraction The range of non-deterministic and probabilistic choices we can handle in practice is limited both by Assumption 5.1 and the practical limitations of SAT solvers. That is, because of Assumption 5.1, we can only handle probabilistic choices for which transition probabilities are independent of the data state. Moreover, the SAT formulas we construct in Section 5.3.2 are linear in the size[16] of the assignment under consideration,

[16] I.e. the number of functions, $Sem_\ell^1, \ldots, Sem_\ell^k$, required to satisfy Assumption 5.1.

which adversely affects the performance of our abstraction procedure for larger choices.

An important direction of future research is therefore to develop alternative abstraction methods. One possibility is to use symbolic data structures instead of SAT (see [LBC03]). A symbolic abstraction method for probabilistic systems is discussed in [DJJL01]. We believe this approach can easily be extended to generate game abstractions. The benefit of this abstraction method is that, unlike SAT, the symbolic data structures used in [DJJL01] are better able to model and abstract probability distributions. On the other hand, symbolic data structures tend to be inefficient at representing the arithmetic expressions that occur in software [Bry91]. Whether a symbolic abstraction procedure could benefit our method requires further investigation.

Secondly, a common approach in non-probabilistic software model checking is to approximate the transition functions of an abstract model incrementally [DD01, BMR01, CKSY05, JM05, KS06]. Employing this idea in our method may help alleviate the restrictions on non-deterministic and probabilistic behaviour and may improve the scalability of our approach. That being said, it is is not immediately clear what constitutes an approximation of a game abstraction. This is something we will address in Chapter 6.

Our final remark with regards to abstraction is that programs often contain a lot of mundane loops at the beginning of the control-flow. These loops often parse program arguments or initialise data structures. When using an abstraction framework that both under and over-approximates all possible behaviours, like game abstractions, we need to show that the loops terminate in order to obtain a non-trivial lower bound. Proving termination of these loops via predicate abstractions is often a significant strain on the abstraction-refinement loop. To address this problem we suggest that it may pay to augment our approach with termination analyses like, e.g., [CPR05, CKRW10].

Refinement To improve the robustness of our approach we think it is important to address the issue of divergence. We often observe that we are very consistent in our choice of refinable states. That is, because we employ the same refinement heuristic in every iteration of the abstraction-refinement loop, we often refine very similar states in successive refinement steps. This often happens even when other refinable states exist that may lead

to a non-divergent refinement sequence. This suggests it may be beneficial to employ a refinement heuristic that selects refinable states randomly. Under such a heuristic, if good refinable states exist, we will eventually refine such a state with probability 1. Whether such a heuristic would work well in practice requires further investigation.

Our concluding remark concerns the quality of the predicates that we discover with our refinement procedure. Refinement procedures based on weakest preconditions are well-known to be insufficiently powerful to prove the correctness of certain programs [JM06]. Many state-of-the-art non-probabilistic software model checkers employ more sophisticated refinement methods based on interpolating decision procedures (see, e.g., [HJMM04, JM05, McM06]). Due to the probabilistic nature of our programs, however, we are not currently able to use these decision procedures for refinement purposes. An important direction of future research is therefore to develop techniques through which interpolation-based refinement methods can be employed in a probabilistic setting.

Chapter 6

An Abstraction Preorder for Games

6.1 Introduction

In the previous chapter we introduced an approach for computing *quantitative properties* of *probabilistic programs* through *game abstractions*. We demonstrated that this approach works well on a large number of probabilistic programs. However, to get these results, we made certain assumptions about the behaviour of probabilistic programs, restricting the range of non-deterministic and probabilistic behaviours we can deal with in practice. Moreover, from our experimental results we see that the scalability of our approach is in part dictated by the cost of computing game abstractions. In this chapter, we set out to enable further improvements in the applicability and scalability of our approach by augmenting the theory underlying game abstractions.

Our work in this chapter is motivated by the state-of-the-art in non-probabilistic software model checking. Many software model checkers improve the applicability and scalability of their abstraction methods by computing *approximate* transition functions for abstract models. More specifically, in practice, tools such as SLAM [BR01], BLAST [HJMS03], and SATABS [CKSY05] employ a *nested* abstraction-refinement loop, comprising an "outer loop" which considers increasingly precise sets of predicates and an "inner loop" which considers increasingly precise abstract transition functions for a fixed set of predicates.[1] Usually, for a given set of predicates, these tools first construct a coarse

[1] For simplicity we omit the localisation mappings used in Chapter 5 here.

approximation of the abstract transition function that is relatively cheap to construct — e.g. a *Cartesian abstraction* [BPR03] — and consider more precise transition functions only if we cannot justify adding new predicates. This incremental construction of the abstract transition function is discussed in, e.g., [DD01, BMR01, CKSY05, JM05, KS06].

We believe that employing a nested abstraction-refinement loop with game abstractions could improve the scalability and applicability of the abstraction-refinement approach of Chapter 5. However, unlike for non-probabilistic existential abstractions, it is not clear how to approximate the game abstractions of [KNP06]. That is, although a necessary ingredient of a nested abstraction-refinement loop is the ability to consider a range of abstractions under a fixed abstraction function, for game abstractions we do not currently have this ability. For a fixed abstraction function (induced by predicates) and a fixed MDP (induced by a probabilistic program) there is only one game that is defined to be an abstraction (see Section 3.4). We can currently only consider different game abstractions by changing the abstraction function.

In this chapter, we will improve upon this situation by formalising a conditional notion of abstraction for games. That is, we will provide general conditions under which a game G abstracts an MDP M. These conditions are general enough to ensure that, for a fixed abstraction function, we can consider *many* different game abstractions. We will formalise our abstraction through a notion of *simulation* over games which we will call *strong probabilistic game simulation*. Our simulation-based definition of abstraction is inspired by, e.g., [LT88, DGG97, JL91, SL94, AHKV98, Mil99, CGL94], where abstraction is defined through simulation for a variety of abstraction formalisms. In addition to characterising when a game abstracts an MDP, our simulation also defines when one game is more abstract than another game. Formally, our simulation induces an *abstraction relation* on two-player stochastic games.

Our abstraction relation generalises [KNP06] in various ways. Firstly, we alleviate the restriction that the player C states that player A can choose correspond directly to the non-deterministic choice in some state of the concrete model. Instead, player A is given the ability to over or under-approximate the available choices. This improvement helps eliminate the correlation between the number of player C states in game abstractions

and the number of states in the concrete model. Secondly, inspired by [SL94], we define *combined transitions* for games. That is, we will allow players of the abstraction to simulate behaviours via randomised strategies. This helps us over-approximate probabilistic behaviour with non-determinism. Finally, we also no longer require abstractions to be induced by *abstraction functions*, but consider arbitrary *abstraction relations*, instead.

Although it is our eventual aim to use the improved game-based abstraction framework that we develop in this chapter in the abstraction-refinement loop of Chapter 5, the contribution of this chapter is currently purely theoretical. However, in addition to the approach in Chapter 5, the work in this chapter is also directly applicable to other applications of game abstractions, such as [KKNP08, KNP09, WZ10, KNP10].

The need for a more fine-grained notion of abstraction for games was also recognised in [WZ10]. However, instead of simulation, the approach in [WZ10] is based on the theory of *abstract interpretation* [CC77]. Due to the nature of abstract interpretation, the approach in [WZ10] is more general than ours. In particular, approximations of game abstractions need not be game abstractions themselves. However, the conditions under which an abstraction approximates a game abstraction are not necessarily easy to check. We will focus on defining approximations through simple conditions on, e.g., the transition functions of games. We will use the methods in [WZ10] to show that these conditions are sound. An interesting observation made in [WZ10] is that game abstractions (as constructed in [KNP06]) are the most precise abstractions that one can construct for a given MDP and abstraction function.

The remainder of the chapter is organised as follows. We first give a high-level overview of our abstraction framework. In Section 6.2, we recall the key ideas of game abstractions as they are defined in [KNP06]. Then, Section 6.3 introduces *combined transitions* for games, which are used to define abstraction in a general fashion. Finally, in Section 6.4, we define our simulation-based abstraction relation for games and show that this relation is sound with respect to probabilistic reachability properties.

Overview of chapter In this chapter, unlike in Chapter 5, we will focus on abstraction on the level of MDPs. That is, we focus on computing the quantitative properties

$$Prob^- : \text{MDP} \to [0,1] \quad \text{and} \quad Prob^+ : \text{MDP} \to [0,1]$$

as defined in Section 3.3.2. As abstractions we will use games, $\hat{G} \in \text{GAME}$, as defined in Section 3.4.1. We let

$$Prob^- : \text{GAME} \to [0,1] \times [0,1] \quad \text{and} \quad Prob^+ : \text{GAME} \to [0,1] \times [0,1]$$

be properties on games as defined in Section 3.4.2. To approximate $Prop(M)$ for some property, $Prop \in \{Prob^-, Prob^+\}$, and MDP, M, our goal is to find a game, $\hat{G}$, such that

$$Prop(\hat{G}) \leq \langle Prop(M), Prop(M) \rangle \,. \tag{6.1}$$

This is because if (6.1) holds, by the definition of $\leq$ on $[0,1] \times [0,1]$ as specified in Definition 3.8, then $Prop(\hat{G}) = [l,u]$ is an approximation of $Prop(M)$ — i.e. we have that $Prop(M) \in [l,u]$. In Chapter 5, we obtained such a game by constructing the game specified by Definition 3.28 for some abstraction function. The constructed game satisfies (6.1) by construction (see Theorem 3.30).

In this chapter we take a different approach. Our abstraction framework comprises two components. Our first component is an *embedding function*

$$\rho : \text{MDP} \to \text{GAME} \,,$$

which yields a precise representation of an MDP in the form of a game abstraction. Our second component is an abstraction relation

$$\sqsubseteq \subseteq \text{GAME} \times \text{GAME} \,,$$

which defines a notion of precision on games. If two games $\hat{G}, G \in \text{GAME}$ are such that $\hat{G} \sqsubseteq G$, then this means that the game $\hat{G}$ *abstracts* G or, equivalently, that G is *more*

precise than $\hat{G}$. Through the embedding function ρ, we can define when a game abstracts an MDP: $\hat{G} \in \text{GAME}$ abstracts $M \in \text{MDP}$ if and only if $\hat{G} \sqsubseteq \rho(M)$.

To reason about properties of MDPs with our framework, it is necessary to ensure that the components satisfy certain properties. In particular, we will require that for all properties $\mathit{Prop} \in \{\mathit{Prob}^-, \mathit{Prob}^+\}$ the following requirements are met:

$$\mathit{Prop}(\rho(M)) = \langle \mathit{Prop}(M), \mathit{Prop}(M) \rangle \qquad (\forall M \in \text{MDP}) \tag{6.2}$$

$$\mathit{Prop}(\hat{G}) \leq \mathit{Prop}(G)\ . \qquad (\forall \hat{G}, G \in \text{GAME such that } \hat{G} \sqsubseteq G) \tag{6.3}$$

The first requirement, (6.2), is a consistency requirement that ensures the abstract semantics of embedded MDPs mesh with their concrete semantics. The requirement in (6.3) is a soundness requirement which ensures that, when one game $\hat{G}$ is *more abstract* than another G game, then a property Prop yields a *less precise* interval when it is evaluated on $\hat{G}$ than when it is evaluated on G.

To obtain a game that satisfies (6.1) for a given $M \in \text{MDP}$ we simply need to find a game $G \in \text{GAME}$ that abstracts M. That is, when $\hat{G} \sqsubseteq \rho(M)$, we get (6.1) directly from (6.2) and (6.3).

6.2 The Roles of Player A & C

We now recall how game abstraction was defined Section 3.4. Let $M = \langle S, I, T, L, R \rangle$ be an MDP as defined in Section 3.3.1 and let $\hat{G} = \langle \hat{S}, \hat{I}, \hat{T}, \hat{L}, \hat{R} \rangle$ be a game as defined in Section 3.4.1. Suppose $\hat{G}$ is the game abstraction $\alpha(M)$ of the MDP M under some (surjective) abstraction function $\alpha : S \to \hat{S}$. By Definition 3.27, the transition function $\hat{T}$ of $\hat{G}$ is defined for every $\hat{s} \in \hat{S}$ as:

$$\hat{T}(\hat{s}) = \left\{ \{\alpha(\lambda) \mid \lambda \in T(s)\} \mid s \in \alpha^{-1}(\hat{s}) \right\}\ .$$

Recall that the key idea of the game abstraction is to separate the non-determinism that arises from the abstraction from the non-determinism that occurs in the concrete model. This is done by attributing these different types of non-deterministic choice to player A

and player C in $\hat{G}$, respectively. That is, the role of player A in a player A state $\hat{s} \in \hat{S}$ is to pick a player C state $\hat{\Lambda} \in \hat{T}(\hat{s})$. By definition of $\hat{T}$ this player C state corresponds to a set of abstracted non-deterministic choices, $\{\alpha(\lambda) \mid \lambda \in T(s)\}$, for some state $s \in S$ of M that $\hat{s}$ abstracts. The role of player C is to resolve M's non-determinism. Given a player C state $\{\alpha(\lambda) \mid \lambda \in T(s)\}$ in $\hat{T}(\hat{s})$ corresponding to some state $s \in S$ of M, player C picks a distribution $\alpha(\lambda) \in \mathbb{D}\hat{S}$ for some non-deterministic choice $\lambda \in T(s)$ in M.

Intuitively, due to player A's role, game abstractions have the natural ability to provide a lower bound and an upper bound on the value of the properties we consider by taking the infimum or supremum value over all available player A strategies, respectively. When we compute these extremum values, due to player C's role, we quantify over player C strategies as we would over M's strategies. For example, to obtain a lower bound on a probabilistic safety property, we take the infimum over player A strategies and the supremum over player C strategies (see Definition 3.23). This is quite different to how over and under-approximation are realised in, say, *modal* or *mixed* abstractions [LT88, DGG97], where a separate transition function is used to realise over and under-approximation.

In the remainder of this chapter, we will generalise the work of [KNP06]. This generalisation no longer directly enforce the correspondence between player C states $\hat{\Lambda} \in \hat{T}(\hat{s})$ in $\hat{G}$ and concrete states of M that $\hat{s}$ abstracts. However, our generalisation will preserve the main spirit of game abstractions: the roles played by player A and C.

6.2.1 MDP Embeddings

With the roles of player A and C clarified, we are in a position to define the embedding function ρ : MDP $\to$ GAME. Evidently, with *two* types of non-deterministic choice, two-player games are more expressive than MDPs. Considering the role player C plays in games, all non-determinism is attributed to player C in embeddings:

Definition 6.1 (Embeddings). *Let $M = \langle S, I, T, L, R\rangle$ be an MDP. The* embedding *$\rho(M)$ of M in* GAME *is the game $\langle S, I, T_\rho, L_\rho, R_\rho\rangle$ where*

- *$T_\rho(s) = \{T(s)\}$ for all $s \in S$,*
- *$L_\rho(s, a) = \langle L(s, a), L(s, a)\rangle$ for all $s \in S$ and $a \in AP$ and*

```
void main()
{
  uchar n=*;

  while (n>0 &&
    coin(n,256))
  { n--; }

  if (n==0)
    target();
}
```

Figure 6.1: A probabilistic program P for which $Prob^+([\![P]\!]) = 1$.

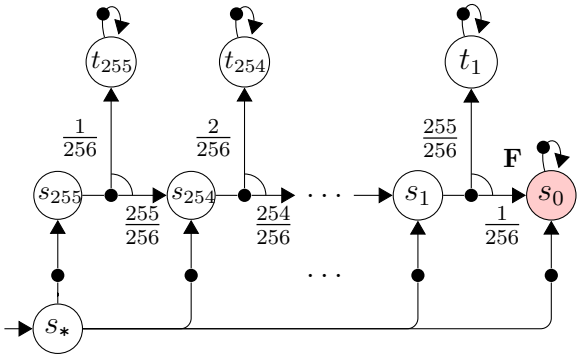


Figure 6.2: An MDP that is describing the semantics $[\![P]\!]$ of the probabilistic program depicted in Figure 6.1.

– $R_\rho(s,a) = \langle R(s), R(s)\rangle$ *for all* $s \in S$.

Embedded MDPs are games in which propositional labels are Boolean and player A has no power. Definition 6.1 allows us to treat MDPs as a special type of game and, more importantly, it enables us to define abstraction as a relation between games. Note that embedded MDPs are always finitely branching for player A.

We will illustrate Definition 6.1 with an example:

Example 6.2. *Consider the program P depicted in Figure 6.1. The data space $\mathscr{U}_{\{\mathtt{n}\}}$ of P is induced by a single variable* n *which is an integer ranging from 0 to 255. The program first assigns* n *non-deterministically. Then, the program enters a loop. In each iteration of this loop, it decrements* n *with probability $\frac{\mathtt{n}}{256}$ and exits the loop with probability $\frac{256-\mathtt{n}}{256}$. The loop terminates as soon as* n *is 0. We are interested in the probability that* n *is 0 at the end of this program. We sketch the MDP $[\![P]\!]$ in Figure 6.2 (we have omitted some initial states). From Figure 6.2 we see that:*

$$Prob^-(M) = \tfrac{255}{256} \cdot \tfrac{254}{256} \cdot \ldots \cdot \tfrac{1}{256} \quad \text{and} \quad Prob^+(M) = 1\,.$$

The game embedding $\rho(M)$ of M is shown in Figure 6.3.

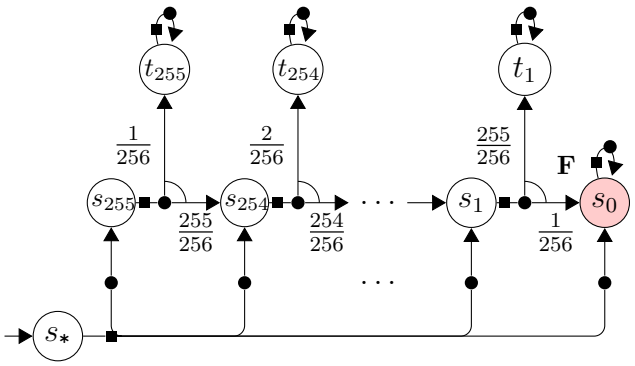


FIGURE 6.3: The game embedding $\rho(M)$ of the MDP M depicted in Figure 6.2.

6.2.2 Consistency Requirement

With the embedding function $\rho : \text{MDP} \to \text{GAME}$ in place we have everything we need to formalise the consistency requirement (6.3) stated in the introduction:

Lemma 6.3. *For all* $M \in \text{MDP}$ *and* $Prop \in \{Prob^-, Prob^+\}$ *we have:*

$$Prop(\rho(M)) = \langle Prop(M), Prop(M) \rangle.$$

Proof. Follows directly from the definition of properties (Section 3.3.2 and 3.4.2) and the definition of embeddings (Definition 6.1). □

The only difference between the evaluation of the upper and lower bound for properties of games is the quantification over player A strategies. However, in embedded games there is precisely one player A strategy and the bounds are trivially equivalent.

6.3 Combined Transitions

In our games, players are permitted to make *probabilistic* choices — that is, they can play with *randomised* strategies. In [SL94], it was recognised that if we provision for abstractions to "simulate" behaviours of embeddings via *randomised* choices, then we can obtain more general notions of abstraction.

Example 6.4. *Consider the game* $\hat{G}$ *and the embedding* $\rho(M)$ *depicted in Figure 6.4.*

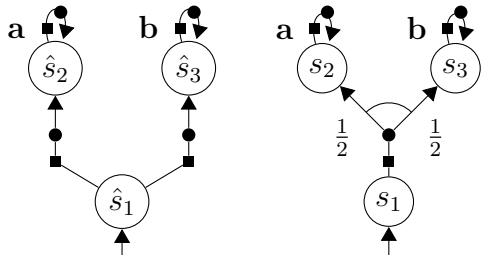


FIGURE 6.4: A game $\hat{G}$ (left) that abstracts the embedding $\rho(M)$ (right) only with combined player A transitions.

Player C is powerless in both games. Consider the player A transition $s_1 \to \{\frac{1}{2}[s_2] + \frac{1}{2}[s_3]\}$ *in* $\rho(M)$. *Neither player A transition* $\hat{s}_1 \to \{[\hat{s}_2]\}$ *or* $\hat{s}_1 \to \{[\hat{s}_3]\}$ *leads to a player C state that matches* $\{\frac{1}{2}[s_2] + \frac{1}{2}[s_3]\}$ *but, if player A of* $\hat{G}$ *chooses both transitions with equal probability, the behaviour of* $\hat{G}$ *effectively matches that of* $\rho(M)$.

In [SL94], the ability to reason about probabilistic choices is formalised through fictitious transitions called *combined transitions*. These transitions are essentially weighted combinations of normal transitions. However, as [SL94] concerns an abstraction relation over MDPs, it only considers combined transitions for one level of choice. In this section, we extend this to combined transitions for a two-player game.

6.3.1 Combined Player C Transitions

We first explain combined player C transitions, which are similar to the combined transitions in [SL94], via a geometric interpretation of player A and player C states. As explained in, say, [MM05, Chapter 6], probability distributions in $\mathbb{D}S$ can be interpreted geometrically as points on the unit plane in $|S|$-dimensional Euclidean space.[2] With the same analogy, we can interpret *sets* of probability distributions, i.e. elements of $\overline{\mathbb{P}}\mathbb{D}S$, as *sets* of points. This geometric interpretation helps us give an intuition as to the choices available to player A and player C in states of the game. We illustrate this with an example:

Example 6.5. *Consider the game* $G = \langle S, I, T, L, R\rangle$ *depicted in Figure 6.5 with* $T(s_1) =$

[2] Unlike [MM05], our probability distributions always sum to 1.

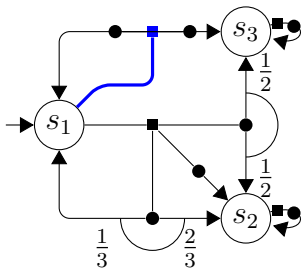


FIGURE 6.5: A game with highlighted player A choice $\{[s_1], [s_3]\} \in T(s_1)\}$.

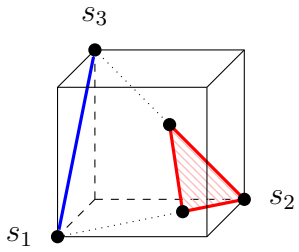


FIGURE 6.6: Geometric interpretation of transitions in $T(s_1)$.

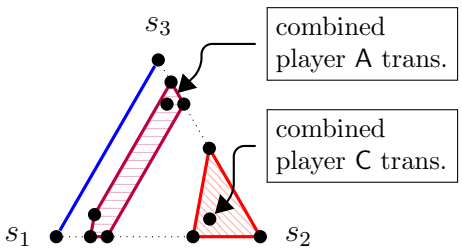


FIGURE 6.7: Geometric interpretation of a *combined* player A transition and a *combined* player C transitions.

$\{\{[s_1], [s_3]\}, \{[s_2], \frac{1}{2}[s_2]+\frac{1}{2}[s_3], \frac{1}{3}[s_1]+\frac{2}{3}[s_2]\}\}$. *The geometric interpretation of* $T(s_1)$ *is depicted in Figure 6.6. The distributions* $[s_1]$, $[s_2]$, $[s_3]$, $\frac{1}{2}[s_2] + \frac{1}{2}[s_3]$ *and* $\frac{1}{3}[s_1] + \frac{2}{3}[s_2]$ *are points on the unit plane. The line between* $[s_1]$ *and* $[s_3]$ *represents the set* $\{[s_1], [s_3]\}$ *and the triangular shape is the set* $\{[s_2], \frac{1}{2}[s_2] + \frac{1}{2}[s_3], \frac{1}{3}[s_1] + \frac{2}{3}[s_2]\}$.

In Example 6.5, in the player C state $\{[s_2], \frac{1}{2}[s_2] + \frac{1}{2}[s_3], \frac{1}{3}[s_1] + \frac{2}{3}[s_2]\}$, player C can make a *probabilistic* choice over probabilistic states. Suppose player C employs a random strategy that picks $[s_2]$ with probability $\frac{1}{5}$, $\frac{1}{2}[s_2]+\frac{1}{2}[s_3]$ with probability $\frac{1}{5}$ and $\frac{1}{3}[s_1]+\frac{2}{3}[s_2]$ with probability $\frac{3}{5}$. The behavioural effect of such a player C strategy is identical to choosing a probabilistic state $\frac{1}{5}[s_1]+\frac{7}{10}[s_2]+\frac{1}{10}[s_3]$ with probability 1. It is this observation that allows us to use fictitious (combined) transitions to probabilistic states to reason about probabilistic combinations of transitions to probabilistic states.

Definition 6.6 (Combined player C transition). *Let* $\Lambda \in \mathbb{PD}S$ *be a player C state in some game* G. *We say there is a* combined player C transition *from* Λ *to* λ, *denoted* $\Lambda \xrightarrow{Cmb} \lambda$, *if and only if, for some countable index set* I *and some family* $\{\lambda_i\}_{i\in I}$ *of distributions, such that* $\Lambda \to \lambda_i$ *for each* $i \in I$, *and some family* $\{w_i\}_{i\in I}$ *of weights, we have* $\lambda = \sum_{i\in I} w_i \cdot \lambda_i$.

Recall that weights always sum to 1. By definition, every normal player C transition $\hat{\Lambda} \to \lambda$ is also a combined player C transition $\hat{\Lambda} \xrightarrow{\text{Cmb}} \lambda$.

Example 6.7. *Consider again the game* G *in Figure 6.5 and the player C state with* $\{[s_2], \frac{1}{2}[s_2]+\frac{1}{2}[s_3], \frac{1}{3}[s_1]+\frac{2}{3}[s_2]\}$ *in* $T(s_1)$. *Consider an index set* $I = \{0, 1, 2\}$ *and the family*

of weights $w_0 = w_1 = \frac{1}{5}$ *and* $w_2 = \frac{3}{5}$ *and probabilistic states* $\lambda_0 = [s_2]$, $\lambda_1 = \frac{1}{2}[s_2] + \frac{1}{2}[s_3]$ *and* $\lambda_2 = \frac{1}{3}[s_1] + \frac{2}{3}[s_2]$. *We have that*

$$w_0 \cdot \lambda_0 + w_1 \cdot \lambda_1 + w_2 \cdot \lambda_2 \;=\; \tfrac{1}{5}[s_1] + \tfrac{7}{10}[s_2] + \tfrac{1}{10}[s_3]$$

and hence

$$\{[s_2], \tfrac{1}{2}[s_2] + \tfrac{1}{2}[s_3], \tfrac{1}{3}[s_1] + \tfrac{2}{3}[s_2]\} \xrightarrow{Cmb} \tfrac{1}{5}[s_1] + \tfrac{7}{10}[s_2] + \tfrac{1}{10}[s_3]$$

is a valid combined player C *transition. This transition is also depicted in Figure 6.7.*

Given the definition of combined player C transitions we can think of player C states as a set of points defining the hull of a convex shape from which player C can draw *any* probabilistic state.

6.3.2 Combined Player A Transitions

The main elegance of combined player C transitions is that they are of the same type as ordinary player C transitions. This is possible because we can represent a distribution over probabilistic states (i.e. an element of $\mathbb{D}\mathbb{D}S$) with a probabilistic state (i.e. an element of $\mathbb{D}S$) itself. We will show that an analogous treatment of combined transitions for player A is also possible. Recall player A chooses between player C states (i.e. *sets* of distributions). A weighted combination of player A transitions is therefore an element of $\mathbb{D}\overline{\mathbb{P}}\mathbb{D}S$ but, crucially, we will represent this weighted combination with a fictitious player C state (i.e. an element of $\overline{\mathbb{P}}\mathbb{D}S$) itself. This allows us to represent a family of weighted player C transitions with a single combined player C transition from a fictitious player C state.

Consider again the MDP depicted in Figure 6.5 and the player C states

$$\Lambda_0 = \{[s_1], [s_3]\} \quad \text{and} \quad \Lambda_1 = \{[s_2], \tfrac{1}{3}[s_1] + \tfrac{2}{3}[s_2], \tfrac{1}{2}[s_2] + \tfrac{1}{2}[s_3]\}$$

available in s_1 as depicted in Figure 6.6. Suppose player A chooses Λ_0 with weight $\frac{3}{4}$ and

Λ_1 with weight $\frac{1}{4}$. Then, regardless of what player C does in Λ_0 or Λ_1, the effective result is a distribution in the convex shape depicted in Figure 6.7 (filled with horizontal lines). As is the case for player C, we could get the same result by letting player A transition to a (fictitious) player C state corresponding to this shape.

Definition 6.8 (Combined player A transition). *Let $s \in S$ be a player A state in some game G and let $\Lambda \in \overline{\mathbb{P}}\mathbb{D}S$ be a set of distributions. We say there is a* combined player A transition *from s to Λ, denoted $s \xrightarrow{Cmb} \Lambda$, if and only if, for some* finite *index set I and for some* finite *family $\{\Lambda_i\}_{i\in I}$ of sets of distributions such that $s \to \Lambda_i$ is a player A transition for each $i \in I$, and some family $\{w_i\}_{i\in I}$ of weights, we have*

$$\Lambda = \left\{ \textstyle\sum_{i\in I} w_i \cdot \lambda_i \mid \{\lambda_i\}_{i\in I} \text{ is a family of distributions s.t. } \forall i \in I : \lambda_i \in \Lambda_i \right\} . \tag{6.4}$$

Note that a *family* of distributions $\{\lambda_i\}_{i\in I}$ introduces a distribution in Λ. The reason we restrict to finite combinations of player A transitions is that there may be uncountably many families when I is countable. This issue does not arise for combined player C transitions.

Analogous to player C transitions, every normal player A transition is a combined player A transition. To ease notation, for the remainder of this chapter, given families $\{\Lambda_i\}_{i\in I}$ and $\{w_i\}_{i\in I}$ we will write $\sum_{i\in I} w_i \cdot \Lambda_i$ to mean the set of distributions in (6.4). Mathematically, combined player A transitions are weighted *Minkowski* additions — operations used to combine convex shapes (see, e.g., [Sch93, Chapter 3]).

Example 6.9. *Consider again the MDP depicted in Figure 6.5. The combined player A transition induced by the index set $I = \{0, 1\}$, weights $w_0 = \frac{3}{4}$, $w_1 = \frac{1}{4}$ and player C states*

$$\Lambda_0 = \{[s_1], [s_3]\} \quad \text{and} \quad \Lambda_1 = \{[s_2], \tfrac{1}{3}[s_1] + \tfrac{2}{3}[s_2], \tfrac{1}{2}[s_2] + \tfrac{1}{2}[s_3]\}$$

is the transition

$$\begin{aligned} s_1 \xrightarrow{Cmb} \{&\tfrac{3}{4}[s_1] + \tfrac{1}{4}[s_2], \tfrac{5}{6}[s_1] + \tfrac{1}{6}[s_2], \tfrac{3}{4}[s_1] + \tfrac{1}{8}[s_2] + \tfrac{1}{8}[s_3], \\ &\tfrac{3}{4}[s_3] + \tfrac{1}{4}[s_2], \tfrac{3}{4}[s_3] + \tfrac{2}{12}[s_2] + \tfrac{1}{12}[s_1], \tfrac{7}{8}[s_3] + \tfrac{1}{8}[s_2]\} . \end{aligned}$$

The resulting player C state contains the distribution $\frac{3}{4}\lambda_0 + \frac{1}{4}\lambda_1$ for every $\lambda_0 \in \Lambda_0$ and $\lambda_1 \in \Lambda_1$. This player C state is depicted Figure 6.6 as the shape filled with horizontal lines.

6.4 An Abstraction Relation

In this section, we will define an *abstraction relation* $\sqsubseteq \subseteq \text{GAME} \times \text{GAME}$ over games via a notion of simulation on games. This formulation of abstraction is notably inspired by, e.g., [LT88, DGG97, SL94], where abstraction is defined using simulations on a variety of formalisms. We will introduce our notion of simulation in Section 6.4.1 and prove some properties of this simulation in Section 6.4.2. Finally, in Section 6.4.3, we will see how our abstraction relation compares to the notion of abstraction in [KNP06].

6.4.1 Strong Probabilistic Game Simulation

Having defined combined transitions in the previous section we are now in a position to define the abstraction relation $\sqsubseteq \subseteq \text{GAME} \times \text{GAME}$ through a new notion of simulation over games that we will call *strong probabilistic game simulation*. We use the adjective "strong" to say we make no special provision for simulations via unobservable steps and we call our simulation "probabilistic" to say we use combined transitions.

Definition 6.10 (Strong probabilistic game simulation). *Let $G = \langle S, I, T, L, R\rangle$ be a game and let $\mathscr{R} \subseteq S \times S$ be a relation on S. We call $\mathscr{R}$ a* strong probabilistic game simulation *on G iff for all $\langle \hat{s}, s\rangle \in \mathscr{R}$ all of the following conditions hold:*

(i) $L(\hat{s}, a) \leq L(s, a)$ *for all* $a \in AP$,

(ii) $R(\hat{s}) \leq R(s)$,

(iii) $\forall s \to \Lambda : \exists \hat{s} \xrightarrow{Cmb} \hat{\Lambda} : \forall \Lambda \to \lambda : \exists \hat{\Lambda} \xrightarrow{Cmb} \hat{\lambda} : \langle \hat{\lambda}, \lambda\rangle \in \mathcal{L}(\mathscr{R})$ *and*

(iv) $\forall s \to \Lambda : \exists \hat{s} \xrightarrow{Cmb} \hat{\Lambda} : \forall \hat{\Lambda} \to \hat{\lambda} : \exists \Lambda \xrightarrow{Cmb} \lambda : \langle \hat{\lambda}, \lambda\rangle \in \mathcal{L}(\mathscr{R})$.

Moreover, we let $\sqsubseteq \subseteq \text{GAME} \times \text{GAME}$ be the relation on games such that $\hat{G} \sqsubseteq G$ if and only if there exists a strong probabilistic game simulation $\mathscr{R}$ on the disjoint product $\hat{G} \uplus G$

such that $I \subseteq \mathscr{R}(\hat{I})$ *and* $\hat{I} \subseteq \mathscr{R}^{-1}(I)$.

Let $G = \langle S, I, T, L, R\rangle$ be a game. A strong probabilistic game simulation $\mathscr{R} \subseteq S \times S$ on G defines a notion of abstraction on the states of G. That is, $\mathscr{R}$ is a relation on S where $\langle \hat{s}, s\rangle \in \mathscr{R}$ means that $\hat{s}$ abstracts (or "simulates") s. Conditions (i) and (ii) are relatively standard and ensure that the propositional and reward labelling in the "abstract" state $\hat{s}$ are less precise than in the "concrete" state s in accordance with the order on $[0,1] \times [0,1]$ and $[0,\infty] \times [0,\infty]$ discussed in Definition 3.8.

Condition (iii) places conditions on the transitions available in $\hat{s}$. Ignoring the first two quantifiers over player A transitions, the condition is much like the *strong probabilistic simulation* of [SL94]. That is, for two player C states $\hat{\Lambda}, \Lambda \in \overline{\mathbb{P}}\mathbb{D}S$ to satisfy the innermost quantifier pair of condition (iii), it must be that every player C transition $\Lambda \to \lambda$ can be matched with a combined player C transition $\hat{\Lambda} \xrightarrow{\text{Cmb}} \hat{\lambda}$ such that the two resulting probabilistic states $\hat{\lambda}$ and λ are in $\mathcal{L}(\mathscr{R})$ (see Definition 3.1, page 25). Essentially this means that everything that player C can do in Λ can be matched by player C in $\hat{\Lambda}$. This condition only works in one direction: player C in $\hat{\Lambda}$ may be able to make transitions that player C in Λ cannot simulate. Hence, we say $\hat{\Lambda}$ *over-approximates* Λ. An intuition for over-approximation of player C states is that, when $\mathscr{R}$ is the identity relation, the convex hull defined by $\hat{\Lambda}$ includes the convex hull defined by Λ.

If we include the first two quantifiers of (iii) in our discussion then we see that every player A transition $s \to \Lambda$ from the concrete state s must be matched by a combined player A transition $\hat{s} \xrightarrow{\text{Cmb}} \hat{\Lambda}$ such that $\hat{\Lambda}$ over-approximates Λ. Note that for both player A and player C transitions we allow the abstract state to match the behaviour of the concrete state with combined transitions.

Condition (iv) is very similar to condition (iii) except that the direction of simulation is reversed for player C. That is, for two player C states $\hat{\Lambda}, \Lambda \in \overline{\mathbb{P}}\mathbb{D}S$ to satisfy the innermost quantifier pair of condition (iv) everything player C can do in $\hat{\Lambda}$ must be matched by player C in Λ. We say in this case that $\hat{\Lambda}$ *under-approximates* Λ. The intuition for under-approximation is that with the identity relation the hull defined by $\hat{\Lambda}$ is included in the hull of Λ. The alternation in (iv) is akin to the preservation condition

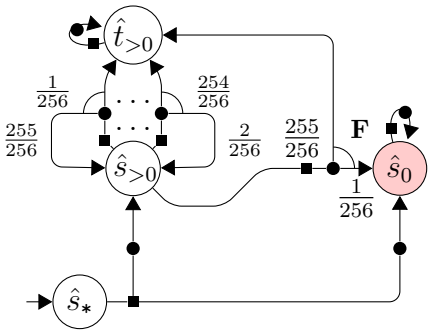


FIGURE 6.8: A game abstraction G_1 of the MDP in Fig. 6.2 with 255 player C states in $\hat{T}(\hat{s}_{>0})$.

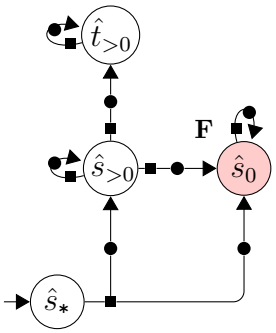


FIGURE 6.9: A game abstraction G_2 of the MDP in Fig. 6.2 with 3 player C states in $\hat{T}(\hat{s}_{>0})$.

of (non-probabilistic) alternating simulations in [AHKV98].

Strong probabilistic game simulations are relations on states of a single game. In the final part of Definition 6.10 we define the abstraction relation $\sqsubseteq \subseteq \text{GAME} \times \text{GAME}$ by considering strong probabilistic game simulations over pairs of games. We require that all initial states of the concrete game are abstracted by some initial state of the abstract game and that all initial states of the abstract game abstract some initial states of the concrete game, respectively.

We illustrate the definition of strong probabilistic game simulation with an example:

Example 6.11. *Reconsider the MDP M depicted in Figure 6.2 (page 113) and its embedding $\rho(M)$ depicted in Figure 6.3 (page 114). Now consider the games $\hat{G}_1$ and $\hat{G}_2$ depicted in Figure 6.8 and 6.9, respectively. Assume the cost labelling is $\langle 0, 0\rangle$ for every state of every game we consider here. The relation*

$$\mathscr{R} = \left\{\langle \hat{s}_*, s_*\rangle, \langle \hat{s}_0, s_0\rangle, \langle \hat{s}_{>0}, s_1\rangle, \ldots, \langle \hat{s}_{>0}, s_{255}\rangle, \langle \hat{t}_{>0}, t_1\rangle, \ldots, \langle \hat{t}_{>0}, t_{255}\rangle\right\}$$

is a strong probabilistic game simulation on both $\hat{G}_1 \uplus \rho(M)$ and $\hat{G}_2 \uplus \rho(M)$ and shows that $\hat{G}_1 \sqsubseteq \rho(M)$ and $\hat{G}_2 \sqsubseteq \rho(M)$.

As this example shows, our conditional notion of abstraction allows us to consider different game abstractions of different precision for a fixed abstraction relation $\mathscr{R}$ and a fixed MDP M. Moreover, our abstraction preorder can also be used to order this

abstraction in terms of precision. That is, it is not hard to see that $\hat{G}_2$ is *more* abstract than $\hat{G}_1$, i.e. $\hat{G}_2 \sqsubseteq \hat{G}_1$, via the relation that relates states of $\hat{G}_2$ and $\hat{G}_1$ with the same label.

The price we pay for $\hat{G}_2$'s precision is the number of player C states in $\hat{s}_{>0}$. Whereas $\hat{G}_2$ has 255 such player C states, the less precise abstraction, $\hat{G}_1$, only has 3.

To see how our abstraction relation is related to the notion of abstraction defined in Section 3.4 we introduce the following lemma:

Lemma 6.12. *Let $M = \langle S, I, T, L, R\rangle$ be an MDP for which there is a bound $r \in \mathbb{R}$ s.t. $R(s) \leq r$ for all $s \in S$. Moreover, let $\hat{S}'$ be an abstract state space and let $\alpha : S \to \hat{S}'$ be an abstraction function. The game $\alpha(M)$ as defined by Definition 3.28 is such that $\alpha(M) \sqsubseteq \rho(M)$.*

Proof. See Section A.2.1 on page 212. □

The proof of Lemma 6.12 reveals that the relation $\mathscr{R} = \{\langle \hat{s}, s\rangle \in \alpha(S) \times S \mid \alpha(s) = \hat{s}\}$ is a strong probabilistic game simulation on $\alpha(M) \uplus \rho(M)$. In our proof we match every player A transition of $\rho(M)$ with a single player A transition in $\alpha(M)$. That is, we do not use different player A transitions of $\alpha(M)$ to satisfy conditions (iii) and (iv) of Definition 6.10. This means that, in $\alpha(M)$, we use the same player C state to realise both over and under-approximation. We also do not use combined transitions.

We now give an intuitive account of how, in our abstraction framework, the ability to under and over-approximation with different player A transitions and the ability to use combined transitions helps us identify other games $\hat{G}$ that, like $\alpha(M)$, are abstractions of $\rho(M)$ under $\mathscr{R}$. Supposing $\alpha(M) = \langle \hat{S}, \hat{I}, \hat{T}, \hat{L}, \hat{R}\rangle$ we show how we can approximate the transition function $\alpha(M)$ with a game $\hat{G} = \langle \hat{S}, \hat{I}, \hat{T}', \hat{L}, \hat{R}\rangle$ — a game that agrees with $\alpha(M)$ on all components but the transition function — such that $\hat{G} \sqsubseteq \alpha(M)$.[3] We will use the geometric interpretation of player A transitions to introduce $\hat{T}$ and $\hat{T}'$ informally (see Section 6.3).

Suppose that $\hat{T}(\hat{s}_1) = \{\{\hat{\lambda}_0\}, \ldots, \{\hat{\lambda}_6\}\}$ in some state, $\hat{s}_1 \in \hat{S}$ (see Figure 6.10). This

[3] We will see later that $\sqsubseteq$ is transitive, meaning that $\hat{G} \sqsubseteq \rho(M)$, also.

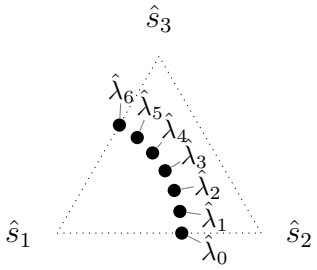


FIGURE 6.10: Geometric interpretation of a number of player C states.

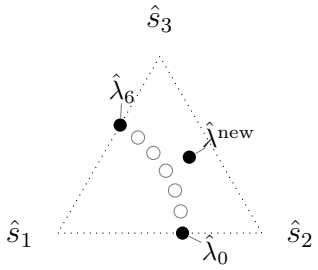


FIGURE 6.11: Representing many player C states via combined player A transitions.

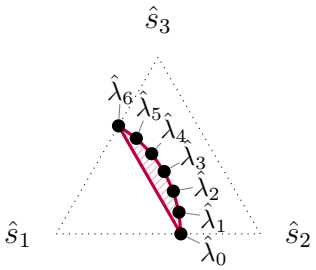


FIGURE 6.12: Geometric interpretation of a single complex player C state.

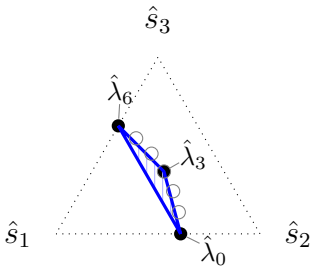


FIGURE 6.13: Under-approximation of the player C state in Figure 6.10.

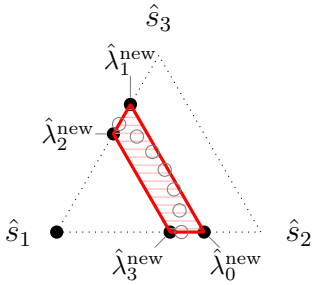


FIGURE 6.14: Over-approximation of the player C state in Figure 6.10.

kind of transition function is typical when $\hat{s}_1$ abstracts many states of M corresponding to probabilistic choices. Due to the ability to match concrete player A transitions with combined player A transitions in abstract models we could consider a less precise transition function with $\hat{T}'(\hat{s}_1) = \{\{\hat{\lambda}_0\}, \{\hat{\lambda}_6\}, \{\hat{\lambda}^{\text{new}}\}\}$ (see Figure 6.11). That is, we can match any player C state available in $\hat{T}$ by taking a weighted combination of the player C states available in $\hat{T}'$. Note that the open circles in Figure 6.11 (and Fig. 6.13 and 6.14) are included for reference only, and are not actual distributions in $\hat{T}'$.

Now suppose that $\hat{T}(\hat{s}_1) = \{\{\hat{\lambda}_0, \ldots, \hat{\lambda}_6\}\}$ (see Figure 6.12). That is $\hat{T}(\hat{s}_1)$ contains a *single* player C state which comprises many distributions. This scenario occurs when $\hat{s}_1$ abstracts a single state of M in which there is a lot of non-determinism. Again we can approximate this player C non-determinism with a more abstract transition relation over the same state space. That is, we can consider transition function where $\hat{T}'(\hat{s}_1)$ yields the set of two player C states, $\{\{\hat{\lambda}_0, \hat{\lambda}_3, \hat{\lambda}_6\}, \{\hat{\lambda}_0^{\text{new}}, \hat{\lambda}_1^{\text{new}}, \hat{\lambda}_2^{\text{new}}, \hat{\lambda}_3^{\text{new}}\}\}$. We depict $\{\hat{\lambda}_0, \hat{\lambda}_3, \hat{\lambda}_6\}$ and $\{\hat{\lambda}_0^{\text{new}}, \hat{\lambda}_1^{\text{new}}, \hat{\lambda}_2^{\text{new}}, \hat{\lambda}_3^{\text{new}}\}$ in Figure 6.13 and 6.14, respectively. In this case we satisfy

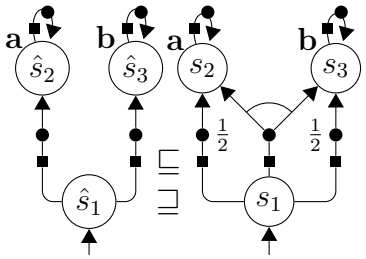


FIGURE 6.15: Two games that are equivalent due to combined player A transitions.

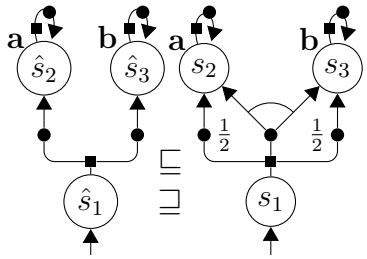


FIGURE 6.16: Two games that are equivalent due to combined player C transitions.

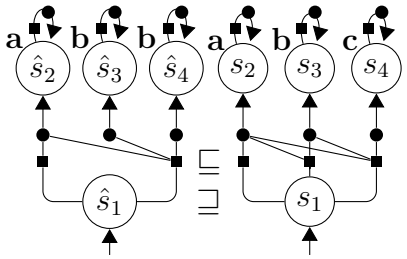


FIGURE 6.17: Two equivalent games due to the ability to under and over approximate non-determinism.

conditions (iii) and (iv) of Definition 6.10 for the player A transition $\hat{s}_1 \to \{\hat{\lambda}_0, \ldots, \hat{\lambda}_6\}$ of $\alpha(M)$ with distinct player A transitions in $\hat{G}$.

In practice the two cases can coincide — there can be many states of M that $\hat{s}_1$ abstracts and each such concrete state may have some complex non-deterministic behaviour.

6.4.2 Properties of Strong Probabilistic Game-Simulation

In this section we prove some properties of our abstraction relation. We start with some observations that are standard for simulations:

Lemma 6.13. *Every game G has a* largest *strong probabilistic game simulation.*

Proof. This follows easily by considering a relation $\mathscr{R} \in S \times S$ such that $\langle \hat{s}, s \rangle \in \mathscr{R}$ if and only if $\langle \hat{s}, s \rangle$ is in some strong probabilistic game simulation on G. That $\mathscr{R}$ is a strong probabilistic game simulation follows from Lemma 3.2, item (iv). □

We will now investigate the nature of the relation $\sqsubseteq$ itself. We will show that $\sqsubseteq$ is a preorder but not a partial order. It is useful for $\sqsubseteq$ to be an order: reflexivity guarantees that a game can abstract itself and transitivity helps us build abstractions incrementally.

Lemma 6.14. *The abstraction relation $\sqsubseteq$ is a* preorder.

Proof. See Section A.2.2 on page 213. □

It turns out that $\sqsubseteq$ is not also a partial order. Examples of $\sqsubseteq$-equivalent games are shown in Figure 6.15, 6.16 and 6.17. For example, the two games in Figure 6.15 are $\sqsubseteq$-

equivalent due to combined player A transitions: the additional player A transition in the right-hand game can be simulated with a combined player A transition in the left-hand game. The two games in Figure 6.16 show an analogous equivalence due to combined player C transitions.

6.4.3 Most and Least Abstract Transition Functions

Recall that the motivation of the work in this chapter is that, in [KNP06], for a fixed MDP M and a fixed abstraction function α, only a single game abstraction can be constructed that abstracts this MDP with this function. In this section, we show that our abstraction preorder is more liberal and, more specifically, that the transition function as defined in [KNP06] is the *most precise* transition function one can define for a given α. Our main interest is in the various ways in which we can define transition function for games. We remark that, orthogonal to this, there are many propositional labellings or cost labellings we can define. We first consider the *most abstract* way in which we can define the transition function of a game abstraction:

Lemma 6.15. *Let $M = \langle S, I, T, L, R\rangle$ be an MDP for which there is a bound $r \in \mathbb{R}$ s.t. $R(s) \leq r$ for all $s \in S$. Moreover, let $\hat{S}'$ be an abstract state space, let $\alpha : S \to \hat{S}'$ be an abstraction function for which $\alpha(S)$ is finite and let $\mathscr{R} \subseteq \alpha(S) \times S$ be the relation defined as $\{\langle \hat{s}, s\rangle \in \alpha(S) \times S \mid \alpha(s) = \hat{s}\}$. A game $\hat{G} = \langle \hat{S}, \hat{I}, \hat{T}, \hat{L}, \hat{R}\rangle$ exists for which $\hat{G} \sqsubseteq \rho(M)$ with $\mathscr{R}$ and*

- *$\hat{T}(\hat{s}) = \{\{[\hat{s}']\} \mid \hat{s}' \in \hat{S}\} \cup \{\{[\hat{s}'] \mid \hat{s}' \in \hat{S}\}\}$ for every $\hat{s} \in \hat{S}$.*

Moreover, for any such $\hat{G}$ we have that any other game $\hat{G}' = \langle \hat{S}, \hat{I}, \hat{T}', \hat{L}, \hat{R}\rangle$ such that $\hat{G}' \sqsubseteq \rho(M)$ with $\mathscr{R}$ and such that $\hat{G}'$ agrees with $\hat{G}$ on $\hat{I}, \hat{L}$ and $\hat{R}$, is abstracted by $\hat{G}$.

Proof. See Section A.2.3 on page 222. □

The number of player C transitions in the game defined in Lemma 6.15 does not depend on the number of concrete states, S, but on the number of abstract states, $\alpha(S)$. It simply defines the most abstract transition function over the state space $\hat{S}$. This is akin to the *total relation* for existential abstractions [CGL94]. Because this transition function is

easy to obtain it may be useful as an initial abstraction in a nested abstraction-refinement loop.

Dual to the *most* abstract transition function is the *least* abstract transition function:

Lemma 6.16. *Let $M = \langle S, I, T, L, R\rangle$ be an MDP for which there is a bound $r \in \mathbb{R}$ s.t. $R(s) \leq r$ for all $s \in S$. Moreover, let $\hat{S}'$ be an abstract state space, let $\alpha : S \to \hat{S}'$ be an abstraction function and let $\mathscr{R} \subseteq \alpha(S) \times S$ be the relation defined as $\{\langle \hat{s}, s\rangle \in \alpha(S) \times S \mid \alpha(s) = \hat{s}\}$. A game $\hat{G} = \langle \alpha(S), \hat{I}, \hat{T}, \hat{L}, \hat{R}\rangle$ exists for which $\hat{G} \sqsubseteq \rho(M)$ with $\mathscr{R}$ and*

- *$\hat{T}(\hat{s}) = \{\{\alpha(\lambda) \mid \lambda \in T(s)\} \mid s \in \alpha^{-1}(\hat{s})\}$ for all $\hat{s} \in \alpha(S)$.*

Moreover, for any such $\hat{G}$ we have that any other game $\hat{G}' = \langle \alpha(S), \hat{I}, \hat{T}', \hat{L}, \hat{R}\rangle$ such that $\hat{G}' \sqsubseteq \rho(M)$ with $\mathscr{R}$ and such that $\hat{G}'$ agrees with $\hat{G}$ on $\hat{I}, \hat{L}$ and $\hat{R}$, abstracts $\hat{G}$.

Proof. See Section A.2.4 on page 224. □

We restrict to a state space $\alpha(S)$ because it is not obvious how to define the transition function on states that do not have any concretisations in a way that is maximal in $\sqsubseteq$.

The least abstract transition function defined in Lemma 6.16 is the transition function used in the framework of [KNP06]. That is, the game abstractions as they are defined in [KNP06] are maximal in $\sqsubseteq$ for a fixed MDP and abstraction function. Observe that, in the worst-case, the number of player C transitions in these games directly corresponds to the number of concrete states in M.

Example 6.17. *Reconsider the MDP M depicted in Figure 6.2 (page 113) and its game abstraction $\hat{G}_1$ depicted in Figure 6.8 (page 121). The game $\hat{G}_1$ is the least abstract game we can define under the relation defined in Example 6.11. The game abstraction $\hat{G}_2$ depicted in Figure 6.9 (page 121) has neither the least nor the most abstract transition function over the relation defined in Example 6.11.*

6.4.4 Soundness Requirement

In this section, we will present the main result of this chapter. We will show that the abstraction relation $\sqsubseteq$ satisfies the soundness requirement informally presented in (6.3).

More specifically, we will show that, for games that are finitely branching for player A, the abstraction preorder $\sqsubseteq$ as defined in Def. 6.10 preserves the properties $Prob^-$ and $Prob^+$.

Theorem 6.18. *Let $\hat{G}$ and G be games such that $\hat{G} \sqsubseteq G$ and $\hat{G}$ and G are finitely branching for player A. We have that*

$$Prob^-(\hat{G}) \leq Prob^-(G) \quad \text{and} \quad Prob^+(\hat{G}) \leq Prob^+(G) \ .$$

Proof. See Section A.2.5 on page 225. □

The soundness requirement (see Theorem 6.18) in combination with the consistency requirement (see Lemma 6.3) enables us to approximate $Prob^-(M)$, $Prob^+(M)$ for MDPs M via games. This restriction to finite branching for player A is not that significant: embedded MDPs are finitely branching for player A by construction and, in the underlying model checker, we can only handle finitary games, anyway. We can extend Theorem 6.18 to include cost properties if the fixpoint characterisations in Lemma 3.26 can be extended to include $Cost^-$ and $Cost^+$.

We illustrate Theorem 6.18 via our running example:

Example 6.19. *Reconsider the MDP M depicted in Figure 6.2 (page 113) and its game abstractions $\hat{G}_1$ and $\hat{G}_2$ depicted in Fig. 6.8 and 6.9 (page 121), respectively. We previously established that $\hat{G}_1 \sqsubseteq \rho(M)$ and $\hat{G}_2 \sqsubseteq \rho(M)$; hence, by Theorem 6.18, we have*

$$Prob^+(\hat{G}_1) \leq Prob^+(\rho(M)) \quad \text{and} \quad Prob^+(\hat{G}_2) \leq Prob^+(\rho(M)) \ .$$

From the figures it is easy to see that $Prob^+(\hat{G}_1) = Prob^+(\hat{G}_2) = \langle 1, 1 \rangle$. Therefore, by definition of $\leq$ and the consistency requirement in Lemma 6.3 we have that $Prob^+(M) = 1$.

Example 6.19 demonstrates that we do not always need to construct the most precise transition function to get good approximations. In our running example we could use either $\hat{G}_1$ or $\hat{G}_2$ to deduce that $Prob^+(M) = 1$.

6.5 Conclusions

In this chapter we generalised the game-based abstraction framework of [KNP06]. Our main motivation for doing this was that, in Chapter 5, we were not able to approximate the transition function of game abstractions. That is, we identified the need for an abstraction framework for game abstractions that is fine-grained enough to allow us to consider multiple game abstractions under a fixed abstraction function.

To achieve this, we developed a conditional notion of abstraction through a simulation on games, called *strong probabilistic game simulation*. With this simulation we no longer require that there is a direct correspondence between player A transitions in game abstractions and concrete states — we allow a separate player A transition to be used for under and over-approximation. We also give game abstractions the ability to simulate concrete behaviours through probabilistic choices. To this end, we generalised an existing notion of *combined transitions* to two-player stochastic games.

We showed that our notion of abstraction generalises that of [KNP06] and that the games constructed by [KNP06] are maximal in our abstraction preorder. In addition to this, we also demonstrated that through strong probabilistic game simulations we are now able to consider many different game abstractions of a program, even under a fixed abstraction function. Finally, we showed that, under mild conditions, our abstraction preorder is sound for probabilistic safety and liveness properties.

We conclude this chapter with suggestions for further work.

Extensions An obvious extension of our results is to extend the soundness result to cost properties. The main step in proving cost properties are preserved by our abstraction preorder is (akin to Lemma 3.26, page 46). We also think it would be interesting to see what other kinds of properties are preserved by our abstraction preorder.

We also think it would be interesting to consider variants of our abstraction preorder. In particular we believe it would be interesting to define a *weak* variant of our *strong* probabilistic game simulation. In the absence of an action labelling, however, such an extension would not be immediately relevant to probabilistic software. We could also

consider separating under-approximating player A transitions from over-approximating player A transitions (akin to [LT88, DGG97]). That is, we could evaluate condition (iv) of Definition 6.10 on the under-approximating transition function and condition (iii) on the over-approximating transition function. We believe that the two levels of non-determinism are only necessary to achieve under-approximation — for the over-approximation a single level of non-determinism may suffice. We may be able to obtain more compact abstractions by exploiting this.

Refinement Another direction of future research is to employ the results of this chapter in the abstraction-refinement method in Chapter 5. The main difficulty in doing so is that the refinement method described in Section 5.4 is not currently able to deal with approximate transition functions. That is, the player A non-determinism in refinable states is no longer necessarily caused by imprecise predicates, but may be caused by an approximate transition function. In the non-probabilistic setting, in CEGAR, there are various approaches for detecting and improving approximate abstract transition functions [DD01, BCDR04, JM05]. However, it requires further investigation to adapt these methods to a probabilistic setting.

Optimality In [WZ10], it was recognised that, with respect to probabilistic safety and liveness properties, for a given abstraction function and a given MDP, the most precise way in which we can evaluate probabilistic safety and liveness properties on the abstract state space is through the game abstractions of [KNP06]. This optimality result is not restricted to game abstractions and includes any kind of abstract model that can be defined on the abstract state space. The optimality in [WZ10] is formulated through abstract interpretation. A similar optimality result is described in [SG06] for non-probabilistic models. Here, an extension of modal abstraction is shown to be the optimal. In [SG06], however, this optimality is defined in terms of the modal μ-calculus properties that can be verified by the abstraction. An interesting direction of research would be to explore the link between the two abstraction formalisms and the two definitions of optimality. More specifically, we think it would be interesting to investigate whether (a probabilistic adaptation of) the modal abstractions in [SG06] is also optimal in the sense of [WZ10].

Complexity Our final remark concerns the computational complexity of $\sqsubseteq$. In an abstraction-refinement implementation, by construction, the game abstractions that are considered are abstractions of the program under consideration. In this setting, we never actually need to check the conditions of Definition 6.10. However, for other applications it may be necessary to actually compute strong probabilistic game simulations. For these cases it is potentially of interest to investigate the computational complexity of $\sqsubseteq$. That is, the complexity of deciding, given games $\hat{G}, G \in \text{GAME}$, whether $\hat{G} \sqsubseteq G$ holds.

Chapter 7

Instrumentation-based Verification of Probabilistic Software

7.1 Introduction

In this chapter, we will propose an alternative method to verify *probabilistic* software. The key idea of this method, named "*instrumentation-based verification*", is to reduce the problem of computing *quantitative* properties of *probabilistic* programs to the problem of verifying a number of *qualitative* properties of *non-probabilistic* programs. We argue that — unlike probabilistic adaptations of non-probabilistic verification techniques — through instrumentation-based verification we can *directly* leverage state-of-the-art tools and techniques in *non-probabilistic* software verification.

We focus exclusively on (probabilistic and non-probabilistic) safety properties in this chapter. There is a large body of research on verifying and refuting *non-probabilistic* safety properties. This includes, notably, classical, manual techniques such as Hoare logic [Flo67, Hoa69, Dij75] and *automated* techniques such as *abstract interpretation* [CC77], *bounded model checking* [BCCY99] and *counter-example guided abstraction refinement* (CEGAR) [Kur94, CGJ+00]. These techniques are known to work well for many practical classes of programs and have been implemented in mature tools such as ASTRÉE [CCF+05], SLAM [BR01, BCLR04], CBMC [CKL04], BLAST [HJMS03], SATABS [CKSY05] and MAGIC [CCG+04]. Many verification techniques for *probabilistic*

systems either *adapt* or *generalise* these non-probabilistic verification techniques. For example, in Chapter 5, inspired by CEGAR, we developed a probabilistic adaptation of non-probabilistic abstraction-refinement techniques. Similar adaptations have been suggested in [HWZ08, DJJL01, DJJL02]. Moreover, probabilistic adaptations of Hoare logic and abstract interpretation have also been considered in [MM05, dHdV02] and [Mon00, DPW01], respectively.

In practice, in terms of automation and scalability, these probabilistic adaptations of non-probabilistic verification techniques do not yet enjoy the same level of success as their non-probabilistic counterparts. We argue this is because the introduction of probabilistic behaviour requires fundamental changes to the very foundations of these verification techniques. In Chapter 5, for example, we had to adapt standard abstraction methods (e.g. [LBC03, CKL04]) to deal with the probabilistic nature of transitions and were only able to do so by making assumptions on how probabilistic behaviour occurs in programs. More importantly, as discussed in Chapter 6, we were not able to use recognised methods to incrementally approximate the transition functions of abstract models (see, e.g., [DD01, BMR01, CKSY05, JM05, KS06]). Moreover, because counter-examples to *probabilistic* safety properties are fundamentally different in nature from counter-examples to *non-probabilistic* safety properties (see, e.g., [HK07, AL09]), we were not able to directly employ state-of-the-art refinement methods such as [CGJ+00, HJMM04].

In this chapter, we develop a new verification technique for probabilistic software that *directly* uses non-probabilistic software verification tools and techniques. That is, we propose to compute *probabilistic* safety properties of *probabilistic* programs by verifying a number of *non-probabilistic* safety properties of *non-probabilistic* programs. Essentially, we will compute (or approximate) probabilistic safety properties by performing a binary search over the unit interval, $[0, 1]$. To do this we need some way to decide whether some bound $\mathbf{p} \in [0, 1]$ is a lower bound or an upper bound for the property under consideration. To accomplish this, we transform (or "*instrument*") the probabilistic program into a non-probabilistic program in such a way that this instrumented program is safe if and only if $\mathbf{p}$ is an upper bound on the probabilistic safety property of the original probabilistic program. This way, we can use non-probabilistic verification tools such as, say, SLAM

[BR01] or BLAST [HJMS03], to obtain bounds on probabilistic safety properties of probabilistic programs. We remark our approach is not limited to the use of model checkers — we can also apply abstract interpreters or, say, Hoare logic, to establish bounds on probabilistic safety properties.

There are many other verification problems that have been solved via a reduction to the verification of non-probabilistic safety properties that can be checked by existing tools. Notably, a reduction method is described in [SB04] which allows one to verify (non-probabilistic) *liveness* properties through safety properties. Analogously, the model checker in [CPR06] verifies safety properties to establish the correctness of a termination argument. In [QW04, LTMP09] a reduction is described which allows one to verify properties of *concurrent* programs via the verification of safety properties of *sequential* programs. Finally, in similar spirit, in [MTLT10], it is shown that certain properties of programs that use the heap can be verified by verifying properties of programs that do not use the heap.

The goals of this chapter are primarily practical in nature — we want to be able to verify real probabilistic software. However, we will also develop a theoretical framework necessary for our approach. In Section 7.2, we formalise the instrumentation process on the level of MDPs. This is applicable to any system with MDP semantics and is not limited to probabilistic software. In Section 7.3 we will discuss in detail how to realise instrumentation on the level of programs. Finally, in Section 7.4, we discuss how we verify non-probabilistic safety properties of instrumented programs in practice. In this section we introduce a tool, called PROBITY, which implements our approach, and present an extensive experimental analysis.

Before we go into the technical details, we discuss the general idea of our approach in a little more detail, using the notation introduced in Chapters 3 and 4.

Overview of chapter In this chapter we will focus on computing probabilistic safety properties $Prob^+$ (see Definition 3.16, page 36). That, is, our aim is to compute or approximate $Prob^+(M)$ for a given MDP M or, more specifically, $Prob^+([\![P]\!])$ for a given probabilistic program P (see Definition 4.1, page 52).

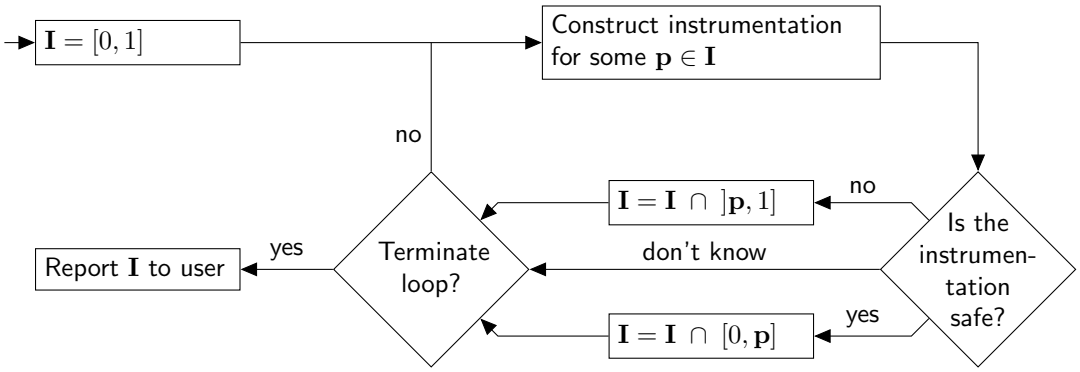


FIGURE 7.1: The overview of an *instrumentation loop* which approximates a probabilistic property $Prob^+([\![P]\!])$ via (non-probabilistic) safety checks on *instrumented* programs.

We depict a high-level overview of our approach in Figure 7.1. We use an interval, $\mathbf{I}$, which is intially $[0, 1]$, to represent our approximation of $Prob^+([\![P]\!])$. In each iteration of the loop in Figure 7.1, we pick a threshold, $\mathbf{p} \in \mathbf{I} \cap \mathbb{Q}$ and decide whether $\mathbf{p}$ is an upper bound on $Prob^+([\![P]\!])$ (i.e. $Prob^+([\![P]\!]) \leq \mathbf{p}$) or a (strict) lower bound (i.e. $Prob^+([\![P]\!]) > \mathbf{p}$). In practice, we pick $\mathbf{p}$ to be in the middle of $\mathbf{I}$. For a given bound $\mathbf{p} \in [0, 1[\cap \mathbb{Q}$ we will resolve this decision problem by verifying or refuting the validity of a *non-probabilistic* safety property $Reach^+ : \text{MDP} \to \mathbb{B}$ for another MDP (as defined by Definition 3.15, page 34). More specifically, we will define a function

$$\langle P, \mathbf{p} \rangle \mapsto P^{\mathbf{p}} \quad : \quad \text{PROG} \times \mathbb{Q} \to \text{PROG} \,, \tag{7.1}$$

which takes a probabilistic program, P, and a bound, $\mathbf{p}$, and produces a non-probabilistic program $P^{\mathbf{p}}$ — called an *instrumentation* — such that

$$Reach^+([\![P^{\mathbf{p}}]\!]) \quad \text{if and only if} \quad Prob^+([\![P]\!]) > \mathbf{p} \,. \tag{7.2}$$

The equivalence in (7.2) enables us to decide whether $Prob^+([\![P]\!])$ exceeds $\mathbf{p}$ by constructing the instrumented program $P^{\mathbf{p}}$ and, say, model checking $Reach^+([\![P^{\mathbf{p}}]\!])$. That is, if $P^{\mathbf{p}}$ is safe for some $\mathbf{p}$, i.e. if $\neg Reach^+([\![P^{\mathbf{p}}]\!])$, then, according to (7.2), $\mathbf{p}$ is an upper bound on $Prob^+([\![P]\!])$ and hence we let $\mathbf{I}$ assume the interval $\mathbf{I} \cap [0, \mathbf{p}]$. Similarly, if $Reach^+([\![P^{\mathbf{p}}]\!])$, then $\mathbf{p}$ is a strict lower bound on $Prob^+([\![P]\!])$ and we let $\mathbf{I}$ be $\mathbf{I} \cap]\mathbf{p}, 1]$.

We terminate the *instrumentation loop* in Figure 7.1 once the difference between the upper and lower bound of $\mathbf{I}$ is within some error threshold. In Section 7.4, we will show that, in practice, we often obtain an interval for which the lower bound and upper bound coincide. We remark that, at least in principle, we can plug in many software verification tools to verify instrumentations.

7.2 Model-level Instrumentation

Although the general focus of this chapter is on verifying probabilistic programs, in this section we formulate the instrumentation process at the level of MDPs. By formulating the instrumentation process on this level our approach is applicable to any system with MDP semantics. Formally, in this section, we introduce an instrumentation function

$$\langle M, \mathbf{p} \rangle \mapsto M^{\mathbf{p}} \quad : \quad \text{MDP} \times \mathbb{Q} \to \text{MDP} \; .$$

This function takes an arbitrary MDP, M, and an arbitrary bound, $\mathbf{p}$, and produces a non-probabilistic MDP, $M^{\mathbf{p}}$. Akin to the discussion in the introduction, in this section, the main soundness criterion this function must satisfy is that $\mathit{Reach}^{+}(M^{\mathbf{p}})$ must hold if and only if $\mathit{Prob}^{+}(M)$ exceeds $\mathbf{p}$. Before we define the instrumentation function we first discuss the nature of the decision problem we are facing in Section 7.2.1 and give an informal description of the instrumentation process in Section 7.2.2 and 7.2.3. We prove the soundness of our instrumentation in Section 7.2.4. Finally, we close this section by discussing alternative methods to instrument MDPs.

7.2.1 When is $\mathit{Prob}^{+}(M) > \mathbf{p}$?

Through the instrumentation process, we want to decide whether, for a given MDP M and bound $\mathbf{p} \in [0, 1[\cap \mathbb{Q}$, is it the case that $\mathit{Prob}^{+}(M)$ is strictly greater than $\mathbf{p}$. The definition of $\mathit{Prob}^{+} : \text{MDP} \to [0, 1]$ (see Definition 3.16, page 36) takes $\mathit{Prob}^{+}(M)$ to be the supremum of a set which contains a probability measure of certain sets of infinite paths of M for every strategy and initial state of M. From this definition it is not easy to

see how it is possible to solve the decision problem via non-probabilistic safety properties. Therefore, in this section, we will simplify this definition.

If $Prob^{+}(M) > \mathbf{p}$ is true then, by the definition of suprema, there is some initial state $s \in I$ and some strategy $\sigma \in Strat_M$ such that the probability of reaching a state satisfying $\mathbf{F}$ from s under σ exceeds $\mathbf{p}$. More formally, we have $Prob^{+}(M) > \mathbf{p}$ if and only if

$$\mathbf{Pr}^{s}_{M,\sigma}(\{\pi \in InfPath^{s}_{M,\sigma} \mid \exists i \in \mathbb{N} : L(\pi^{i}, \mathbf{F})\}) > \mathbf{p} \tag{7.3}$$

for some $s \in I$ and $\sigma \in Strat_M$. We can simplify our decision problem further by observing that taking suprema over *pure* strategies instead of *all* strategies does not affect the value of our property (see Proposition 5.7.1 in [Seg95]). This means that $Prob^{+}(M) > \mathbf{p}$ if and only if (7.3) holds for some initial state $s \in I$ and some *pure strategy* $\sigma \in PureStrat_M$.

Now, using Lemma 3.19 (page 38), we can rewrite the measure in (7.3) as a sum over a possibly infinite set of finite paths, $\mathbf{F}\text{-}FinPath^{s}_{M,\sigma}$. Recall $\mathbf{F}\text{-}FinPath^{s}_{M,\sigma}$ is the set of finite paths of M that start with s, are consistent with σ, and for which the last state satisfies $\mathbf{F}$ and no other state does. We can rewrite (7.3) to

$$\sum_{\pi \in \mathbf{F}\text{-}FinPath^{s}_{M,\sigma}} \textsc{Prob}_{M,\sigma}(\pi) > \mathbf{p}\ . \tag{7.4}$$

That is, the decision problem holds if and only if there is an initial state and a pure strategy under which a countable sum of the probabilities of a certain set of finite paths exceeds $\mathbf{p}$. Our final simplification exploits the fact that the countable sum in (7.4) converges to a value in $[0, 1]$ and is only over non-negative terms. That is, we can rephrase our decision problem as follows: $Prob^{+}(M) > \mathbf{p}$ if and only if for some initial state $s \in I$ and some pure strategy $\sigma \in PureStrat_M$ there is a *finite* subset, $\Pi \subseteq \mathbf{F}\text{-}FinPath^{s}_{M,\sigma}$, such that

$$\sum_{\pi \in \Pi} \textsc{Prob}_{M,\sigma}(\pi) > \mathbf{p}\ .$$

We close this section with an example:

Example 7.1. *Consider the MDP M depicted in Figure 7.2. The states s_7, s_{10}, s_{12} and*

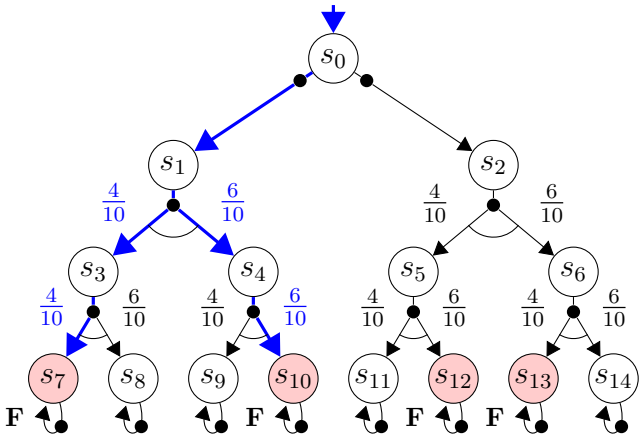


FIGURE 7.2: An MDP M with $Prob^{+}(M) = \frac{13}{25}$ — the highlighted states represent the target and the highlighted paths are some paths of M to this target.

s_{13} satisfy $\mathbf{F}$ *and, hence,* $Prob^{+}(M) = \frac{13}{25}$. *Let us take a pure strategy* $\sigma \in Strat_M$ *with* $\sigma(s_0) = [[s_1]]$ *(recall* $[[s_1]]$ *is the point distribution on the point distribution on* s_1*). We have*

$$\mathbf{F}\text{-}FinPath^{s_0}_{M,\sigma} = \Big\{ s_0 \xrightarrow{[s_1]} s_1 \xrightarrow{\frac{4}{10}[s_3]+\frac{6}{10}[s_4]} s_3 \xrightarrow{\frac{4}{10}[s_7]+\frac{6}{10}[s_8]} s_7 \,,$$
$$s_0 \xrightarrow{[s_1]} s_1 \xrightarrow{\frac{4}{10}[s_3]+\frac{6}{10}[s_4]} s_4 \xrightarrow{\frac{4}{10}[s_9]+\frac{6}{10}[s_{10}]} s_{10} \Big\} \,.$$

These paths are highlighted in Figure 7.2. If we take $\Pi = \mathbf{F}\text{-}FinPath^{s_0}_{M,\sigma}$ *then we see that*

$$\sum_{\pi\in\Pi} \textsc{Prob}_{M,\sigma}(\pi) = 1\cdot 1\cdot 1\cdot \tfrac{4}{10}\cdot 1\cdot \tfrac{4}{10} \;+\; 1\cdot 1\cdot 1\cdot \tfrac{6}{10}\cdot 1\cdot \tfrac{6}{10} = \tfrac{13}{25} \,.$$

That is, there is an initial state, s_0*, a pure strategy,* σ*, and a finite subset of* $\mathbf{F}\text{-}FinPath^{s_0}_{M,\sigma}$ *whose combined probability mass is* $\frac{13}{25}$. *This means that* $Prob^{+}(M) > \mathbf{p}$ *for every* $\mathbf{p} \in [0, \frac{13}{25}[$. *With a dual argument we can deduce that* $Prob^{+}(M) \leq \mathbf{p}$ *for every* $\mathbf{p} \in [\frac{13}{25}, 1[$.

7.2.2 Search-based Instrumentation

We established in Section 7.2.1 that the inequality $Prob^{+}(M) > \mathbf{p}$ holds iff there exists a certain finite *set* of finite paths of M. In contrast, $Reach^{+}(M^{\mathbf{p}})$ holds iff there exists a certain finite path of $M^{\mathbf{p}}$. This means that, intuitively, in order to solve the decision

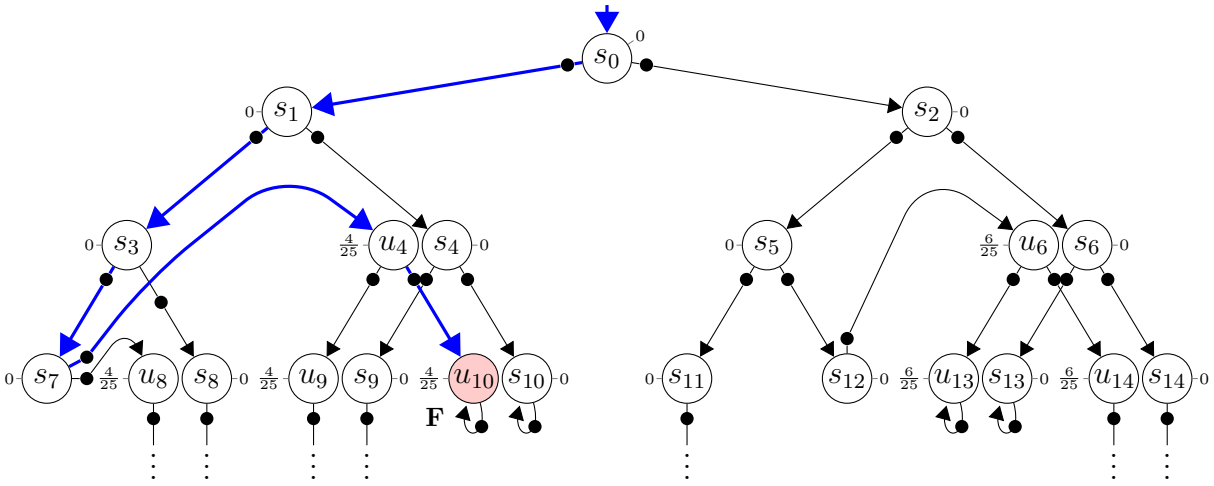


FIGURE 7.3: The instrumentation $M^{\frac{1}{2}}$ of M in Figure 7.2. The instrumentation is non-probabilistic — i.e. all distributions are point distributions. As $Reach^{+}(M^{\frac{1}{2}})$ we have that $Prob^{+}(M) > \frac{1}{2}$.

problem $Prob^{+}(M) > \mathbf{p}$ by model checking $Reach^{+}(M^{\mathbf{p}})$, we need individual finite paths of the instrumentation, $M^{\mathbf{p}}$, to correspond to finite *sets* of finite paths of M.

There are many ways in which we could realise this correspondence. We focus on an approach where instrumentations essentially perform a search over M's state space, looking for paths in $\mathbf{F}$-$FinPath_M$. We clarify this idea with an example:

Example 7.2. *Consider again the MDP M depicted in Figure 7.2. We informally depict an instrumentation of M, namely $M^{\frac{1}{2}}$, in Figure 7.3. What comprises the state space of $M^{\frac{1}{2}}$ will be discussed later — we focus on $M^{\frac{1}{2}}$'s behaviour. We do note there is a direct correspondence between the states s_i, u_i of $M^{\frac{1}{2}}$ and the states s_i of M. The path*

$$s_0 \to s_1 \to s_3 \to s_7 \to u_4 \to u_{10} \tag{7.5}$$

of $M^{\frac{1}{2}}$ (as highlighted in Figure 7.3) corresponds to a set *of paths of M. Essentially the first three transitions of (7.5) directly correspond to the exploration of the path*

$$s_0 \xrightarrow{[s_1]} s_1 \xrightarrow{\frac{4}{10}[s_3]+\frac{6}{10}[s_4]} s_3 \xrightarrow{\frac{4}{10}[s_7]+\frac{6}{10}[s_8]} s_7 \tag{7.6}$$

of M. Then, the last two transitions of (7.5), $s_7 \to u_4 \to u_{10}$, essentially "undo" the last

two transition in (7.6) and then explore the last two transitions of the path of M

$$s_0 \xrightarrow{[s_1]} s_1 \xrightarrow{\frac{4}{10}[s_3]+\frac{6}{10}[s_4]} s_4 \xrightarrow{\frac{4}{10}[s_9]+\frac{6}{10}[s_{10}]} s_{10} \ .$$

That is, the path of $M^{\frac{1}{2}}$ in (7.5) actually corresponds with $\mathbf{F}$-$FinPath^{s_0}_{M,\sigma}$ — the set of paths of M discussed in Example 7.1 (and highlighted in Figure 7.2).

Our informal example has glossed over many details. A particularly important detail is that we must never explore a path of the original MDP M more than once. This is because, in $M^{\mathbf{P}}$, we will sum the probability of all paths of M that we encounter, and we do not want to count the same probability mass twice. To realise this requirement we introduce a new concept — an ordering on the transitions of MDPs:

Definition 7.3 (Transition ordering). *Let $M = \langle S, I, T, L, R\rangle$ be an MDP and let $\preceq \subseteq (S \times \mathbb{D}S \times S) \times (S \times \mathbb{D}S \times S)$ be a partial order on M's transitions. We call $\preceq$ a* transition ordering *if and only if the following condition holds:*

- *for all transitions $s_1 \xrightarrow{\lambda_1} s_1'$ and $s_2 \xrightarrow{\lambda_2} s_2'$ of M we have that $s_1 \xrightarrow{\lambda_1} s_1'$ and $s_2 \xrightarrow{\lambda_2} s_2'$ are comparable in $\preceq$ if* and only if *$s_1 = s_2$ and $\lambda_1 = \lambda_2$.*[1]

For $\preceq$ to be a transition ordering of M it must be total whenever restricted to a fixed source state and a fixed distribution. Our instrumentation process requires that each MDP M is equipped with some transition ordering $\preceq_M$ — we can choose such an ordering arbitrarily. A transition ordering places an order on resolutions of probabilistic choice in M, but does not order resolutions of non-deterministic choice.

We illustrate the concept of transition orders with an example:

Example 7.4. *A transition order $\preceq_M$ for the MDP M depicted in Figure 7.2 is the*

[1] Transitions t_1 and t_2 are comparable in $\preceq$ iff either $t_1 \preceq t_2$ or $t_2 \preceq t_1$ holds.

reflexive closure of the relation

$$\Big\{\big\langle s_1 \xrightarrow{\frac{4}{10}[s_3]+\frac{6}{10}[s_4]} s_3, s_1 \xrightarrow{\frac{4}{10}[s_3]+\frac{6}{10}[s_4]} s_4\big\rangle, \big\langle s_2 \xrightarrow{\frac{4}{10}[s_5]+\frac{6}{10}[s_6]} s_5, s_2 \xrightarrow{\frac{4}{10}[s_5]+\frac{6}{10}[s_6]} s_6\big\rangle,$$
$$\big\langle s_3 \xrightarrow{\frac{4}{10}[s_7]+\frac{6}{10}[s_8]} s_7, s_3 \xrightarrow{\frac{4}{10}[s_7]+\frac{6}{10}[s_8]} s_8\big\rangle, \big\langle s_4 \xrightarrow{\frac{4}{10}[s_9]+\frac{6}{10}[s_{10}]} s_9, s_4 \xrightarrow{\frac{4}{10}[s_9]+\frac{6}{10}[s_{10}]} s_{10}\big\rangle,$$
$$\big\langle s_5 \xrightarrow{\frac{4}{10}[s_{11}]+\frac{6}{10}[s_{12}]} s_{11}, s_5 \xrightarrow{\frac{4}{10}[s_{11}]+\frac{6}{10}[s_{12}]} s_{12}\big\rangle, \big\langle s_6 \xrightarrow{\frac{4}{10}[s_{13}]+\frac{6}{10}[s_{14}]} s_{13}, s_6 \xrightarrow{\frac{4}{10}[s_{13}]+\frac{6}{10}[s_{14}]} s_{14}\big\rangle\Big\}\ .$$

Informally, a transition ordering induces a lexographical order over paths in which a pair of paths can be compared if and only if they start from the same initial state $s \in I$ and are both consistent with some pure strategy $\sigma \in \mathit{PureStrat}_M$ — our instrumentations adhere to this lexographical order. We will formalise this order in our soundness proof.

We note that the path sets of M considered by paths of instrumentations, $M^{\mathbf{p}}$, should be precisely those path sets that we described in Section 7.2.1 — all paths in these sets must originate from the same initial state and must be consistent with a single pure strategy of M. The restriction to a single strategy is perhaps counter-intuitive: a path of $M^{\mathbf{p}}$ explores various ways in which *probabilistic* choices of M can be resolved, but can only consider one resolution of *non-determinism*. This is because other resolutions of non-deterministic choice in M — i.e. other pure strategies of M — are considered by other paths of $M^{\mathbf{p}}$. In other words, a non-deterministic choice in M remains a non-deterministic choice in $M^{\mathbf{p}}$.

The general idea of our instrumentations is that each finite path of $M^{\mathbf{p}}$ corresponds to a finite set $\Pi \subseteq \mathbf{F}\text{-}\mathit{FinPath}^{s}_{M,\sigma}$ for some $s \in I$ and $\sigma \in \mathit{PureStrat}_M$. To decide that $\mathit{Prob}^{+}(M) > \mathbf{p}$, in accordance with Section 7.2.1, we need to establish whether the combined probability mass of paths in one such Π is greater than $\mathbf{p}$. Because this probability mass depends on the path of $M^{\mathbf{p}}$ explored so far, it needs to be stored in the states of $M^{\mathbf{p}}$. This probability mass is depicted in, say, Figure 7.3 with rational numbers next to the states. The accumulated probability mass is the only thing that distinguishes, say, state s_4 and u_4 in Figure 7.3.

To start describing the search-like behaviour of instrumentations in some more detail, we categorise transitions of instrumentations into *explorative* and *backtracking* transitions.

Informally, the idea is that *explorative* transitions of $M^{\mathbf{P}}$ mimic the behaviour of M. Exploration continues as long as we have not encountered a state satisfying $\mathbf{F}$. During the exploration phase, the only behavioural difference between M and $M^{\mathbf{P}}$ is that the *probabilistic* choices of M are resolved *non-deterministically* in $M^{\mathbf{P}}$. This means we are *not* proposing that every pure strategy of M should correspond to a single path of $M^{\mathbf{P}}$. Such an instrumentation scheme would not work when some resolutions of probabilistic choice *never* reach a state satisfying $\mathbf{F}$. A path of $M^{\mathbf{P}}$ that is exploring states of M from which no state satisfying $\mathbf{F}$ is reachable will never backtrack and explore other paths of M. For example, in Figure 7.2, once we reach s_8, there is no way in which we can reach a state satisfying $\mathbf{F}$ — we need instrumentations to be able to avoid exploring paths that lead to s_8. The ability to choose *non-deterministically* between the different resolutions of probabilistic choice of M during exploration (see, e.g., states s_1, s_3 in Figure 7.2) gives instrumentations the ability to "*skip*" certain probabilistic behaviours of the original MDP. In practice this also means there are many paths of $M^{\mathbf{P}}$ that correspond to the same pure strategy of M.

If, during exploration, we encounter a state satisfying $\mathbf{F}$, then we switch to a *backtracking* mode. Backtracking comprises a single step: we add the probability of the current path of M to the accumulated probability mass, we jump back to a probabilistic choice of M we visited earlier in our search — choose a different resolution of it — and proceed with *exploration* again. We will require that this backtracking move always chooses resolutions of probabilistic choice that are (strictly) greater than the current resolution of probabilistic choice in the transition ordering. Note that there may be many probabilistic choices we can backtrack to and each of these may have many resolutions we may choose to explore — each of these induces a backtracking transition (see, e.g., s_7 in Figure 7.3).

We now revisit our example.

Example 7.5. *Consider again the MDP M depicted in Figure 7.2 and its instrumentation $M^{\frac{1}{2}}$ in Figure 7.3 induced by the transition order $\preceq$ defined in Example 7.4. Backtracking transitions of $M^{\frac{1}{2}}$ are $s_7 \to u_8$, $s_7 \to u_4$ and $s_{12} \to u_6$ and the remaining transition are explorative. We consider the backtracking transition $s_7 \to u_4$ in a little*

more detail. Suppose the path of M we followed to s_7 is

$$s_0 \to s_1 \xrightarrow{\frac{4}{10}[s_3]+\frac{6}{10}[s_4]} s_3 \xrightarrow{\frac{4}{10}[s_7]+\frac{6}{10}[s_8]} s_7 \ .$$

In the second and third transition of this path we make a probabilistic choice. Because

$$s_1 \xrightarrow{\frac{4}{10}[s_3]+\frac{6}{10}[s_4]} s_3 \prec s_1 \xrightarrow{\frac{4}{10}[s_3]+\frac{6}{10}[s_4]} s_4$$

we backtrack to s_1 and explore the transition $s_1 \xrightarrow{\frac{4}{10}[s_3]+\frac{6}{10}[s_4]} s_4$ of M. In doing so we add

$$\textsc{Prob}_M(s_0 \to s_1 \xrightarrow{\frac{4}{10}[s_3]+\frac{6}{10}[s_4]} s_3 \xrightarrow{\frac{4}{10}[s_7]+\frac{6}{10}[s_8]} s_7) = \tfrac{4}{10} \cdot \tfrac{4}{10} = \tfrac{4}{25}$$

to the accumulated probability mass and end up in u_4. Note that the non-deterministic *choice we made in s_0 does* not *induce a backtracking transition in $M^{\frac{1}{2}}$.*

7.2.3 Model-level Instrumentation Function

The previous section informally described the semantics of model-level instrumentations. In this section we consider how these semantics can be realised. That is, we will define formally how instrumentations are constructed:

Definition 7.6 (Model-level instrumentation). *Let $M = \langle S, I, T, L, R\rangle$ be an MDP and let $\mathbf{p} \in [0,1[\cap \mathbb{Q}$. We let $M^{\mathbf{p}} = \langle S^{\mathbf{p}}, I^{\mathbf{p}}, T^{\mathbf{p}}, L^{\mathbf{p}}, R^{\mathbf{p}}\rangle$ be the MDP with*

- *$S^{\mathbf{p}} = FinPath^I_M \times \mathbb{Q}$,*
- *$I^{\mathbf{p}} = \{\langle s_i, 0\rangle \mid s_i \in I\}$,*
- *$T^{\mathbf{p}}$ is the smallest function such that for all $\langle \pi, mass\rangle \in S^{\mathbf{p}}$ we have:*
  - *(i) $[\langle \pi \xrightarrow{\lambda} s, mass\rangle] \in T^{\mathbf{p}}(\langle \pi, mass\rangle)$ if $\neg L(\textsc{Last}(\pi), \mathbf{F})$ and $\pi \xrightarrow{\lambda} s$ is a path of M*
  - *(ii) $[\langle \pi', mass + \textsc{Prob}_M(\pi)\rangle] \in T^{\mathbf{p}}(\langle \pi, mass\rangle)$ if $L(\textsc{Last}(\pi), \mathbf{F})$ and π' is a finite path of M satisfying all of the following conditions:*
    - *(a) $|\pi| \geq |\pi'|$*
    - *(b) $\textsc{Trans}(\pi, i) = \textsc{Trans}(\pi', i)$ for all $i < |\pi'| - 1$*

(c) $\textsc{Trans}(\pi, |\pi'| - 1) \prec_M \textsc{Trans}(\pi', |\pi'| - 1)$

(iii) $[\langle \pi, \mathit{mass} \rangle] \in T^{\mathbf{p}}(\langle \pi, \mathit{mass} \rangle)$ *if (i) and (ii) leave* $T^{\mathbf{p}}(\langle \pi, \mathit{mass} \rangle)$ *empty.*

– $L^{\mathbf{p}}$ *is such that, for all* $\langle \pi, \mathit{mass} \rangle \in S^{\mathbf{p}}$ *and* $a \in AP$*, we have that* $L^{\mathbf{p}}(\langle \pi, \mathit{mass} \rangle, a) =$ $\mathtt{tt}$ *if and only if* $a = \mathbf{F}$*,* $L(\textsc{Last}(\pi), \mathbf{F})$ *and* $\mathit{mass} + \textsc{Prob}_M(\pi) > \mathbf{p}$*, and*

– $R^{\mathbf{p}}$ *is defined arbitrarily.*

We first explain why the state space of $M^{\mathbf{p}}$ is $\mathit{FinPath}^I_M \times \mathbb{Q}$. Recall from the definition of MDPs that the transition function, $T^{\mathbf{p}}$, only takes the current state as input — it cannot look at the past behaviour of $M^{\mathbf{p}}$. However, to realise backtracking transitions (e.g. $s_7 \to u_4$ in Figure 7.3), we *do* need to know this past behaviour. More specifically, we need to know the probabilistic choices of M we have explored prior to reaching a target state of M. To enable this we embed *paths* of M into the state space of $M^{\mathbf{p}}$. The idea is that instrumentations store the entire path of M in the current state — much like a search stack — such that backtracking can be realised in $T^{\mathbf{p}}$ by looking at states of $M^{\mathbf{p}}$.

A similar problem arises for the definition of $L^{\mathbf{p}}$ — we need to know the total probability mass of the paths of M we have visited so far, but we cannot consider the past behaviour of $M^{\mathbf{p}}$. The embedding of M's paths is not helpful here as they represent the past behaviour of M instead of the paths of M visited in $M^{\mathbf{p}}$. As informally stated before, to keep track of probability mass we equip states of $M^{\mathbf{p}}$ with a rational value. Informally, this rational value is the combined probability of all paths M we backtracked from.

Following this definition of $S^{\mathbf{p}}$, the definitions of $I^{\mathbf{p}}$ and $L^{\mathbf{p}}$ follow naturally. The labelling function, $L^{\mathbf{p}}$, identifies a state, $\langle \pi, \mathit{mass} \rangle$, as a target state of $M^{\mathbf{p}}$ only if the last state of π is a target state in M and the mass accumulated in *mass* together with the probability mass of the current path, $\textsc{Prob}_M(\pi)$, exceeds $\mathbf{p}$. It therefore remains to explain the definition of $T^{\mathbf{p}}$. Firstly, we observe *all* transitions of $M^{\mathbf{p}}$ are *non-probabilistic*. Condition (i) of $T^{\mathbf{p}}$ corresponds with the *exploration phase* of the search, whereas condition (ii) is responsible for *backtracking*. Essentially, during backtracking, we take a path π' of M which, by conditions (ii.a) – (ii.c) is strictly greater than π in lexographical order over paths of M induced by $\preceq_M$, and add $\textsc{Prob}_M(\pi)$ to *mass*. Moreover, the definition of our transition ordering and condition (ii.c) ensure that π' is realisable with the same

pure strategy that realised π. Finally, condition (iii) prevents deadlocks from occurring in $M^{\mathbf{p}}$ by adding self-loops when we have reached a target state of M and there are no more paths to backtrack to.

Observe that the bound, $\mathbf{p}$, only has an effect on the definition of $L^{\mathbf{p}}$. We are only interested in the labelling for $\mathbf{F} \in \mathrm{AP}$ and this proposition holds in a state $\langle \pi, \mathit{mass} \rangle$ only if the accumulated probability mass of the search so far (i.e. *mass*) plus the probability of the current path in M (i.e. $\textsc{Prob}_M(\pi)$) exceeds $\mathbf{p}$.

We illustrate the definition of instrumentation by returning to our running example:

Example 7.7. *Consider again the MDP M depicted in Figure 7.2 and the informal description of its instrumentation $M^{\frac{1}{2}}$ in Figure 7.3. The path of $M^{\frac{1}{2}}$ discussed in Example 7.1 and 7.2 (and highlighted in Figure 7.3) is actually the path:*

$$
\begin{aligned}
&\langle s_0, 0\rangle \longrightarrow \langle s_0 \xrightarrow{[s_1]} s_1, 0\rangle && (\textit{condition i, explore})\\
&\quad \longrightarrow \langle s_0 \xrightarrow{[s_1]} s_1 \xrightarrow{\frac{4}{10}[s_3]+\frac{6}{10}[s_4]} s_3, 0\rangle && (\textit{condition i, explore})\\
&\quad \longrightarrow \langle s_0 \xrightarrow{[s_1]} s_1 \xrightarrow{\frac{4}{10}[s_3]+\frac{6}{10}[s_4]} s_3 \xrightarrow{\frac{4}{10}[s_7]+\frac{6}{10}[s_8]} s_7, 0\rangle && (\textit{condition i, explore})\\
&\quad \longrightarrow \langle s_0 \xrightarrow{[s_1]} s_1 \xrightarrow{\frac{4}{10}[s_3]+\frac{6}{10}[s_4]} s_4, \tfrac{4}{25}\rangle && (\textit{condition ii, backtrack})\\
&\quad \longrightarrow \langle s_0 \xrightarrow{[s_1]} s_1 \xrightarrow{\frac{4}{10}[s_3]+\frac{6}{10}[s_4]} s_4 \xrightarrow{\frac{4}{10}[s_9]+\frac{6}{10}[s_{10}]} s_{10}, \tfrac{4}{25}\rangle && (\textit{condition i, explore})
\end{aligned}
$$

To see $\mathit{Reach}^{+}(M^{\frac{1}{2}})$ holds, observe that the final state of this path is a target state in $M^{\frac{1}{2}}$ because s_{10} is a target state in M and

$$
\textsc{Prob}_M(s_0 \xrightarrow{[s_1]} s_1 \xrightarrow{\frac{4}{10}[s_3]+\frac{6}{10}[s_4]} s_4 \xrightarrow{\frac{4}{10}[s_9]+\frac{6}{10}[s_{10}]} s_{10}) + \tfrac{4}{25} = \tfrac{6}{10} \cdot \tfrac{6}{10} + \tfrac{4}{25} = \tfrac{9}{25} + \tfrac{4}{25} > \tfrac{1}{2}\,.
$$

We close our section by discussing the structure of instrumented MDPs. Firstly, we remark that instrumented MDPs often have a countably infinite state space even if the original model is a finite-state MDP. As an example consider the finite-state MDP M depicted in Figure 7.4. An infinite-state instrumentation $M^{\frac{3}{5}}$ of M is shown in Figure 7.5. The transition order we used is described in the caption of this figure. Note that, as before, for a state $\langle \pi, \mathit{mass} \rangle$, we depict $\textsc{Last}(\pi)$ inside a state and *mass* as a label attached to

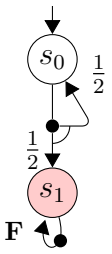


FIGURE 7.4: An MDP M with a loop and with $Prob^{+}(M) = 1$.

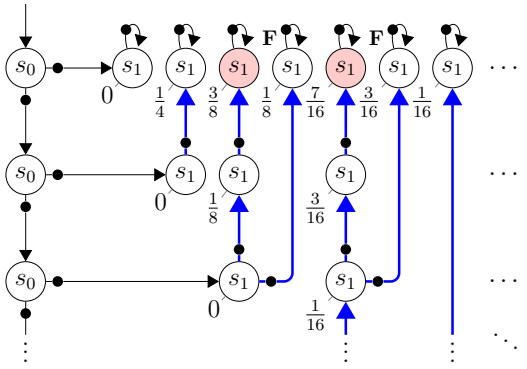


FIGURE 7.5: Instrumentation of Figure 7.4 when $s_0 \overset{\cdots}{\longrightarrow} s_0 \prec s_0 \overset{\cdots}{\longrightarrow} s_1$ and bound $\mathbf{p} = \frac{3}{5}$ — the *least* probable paths are counted first.

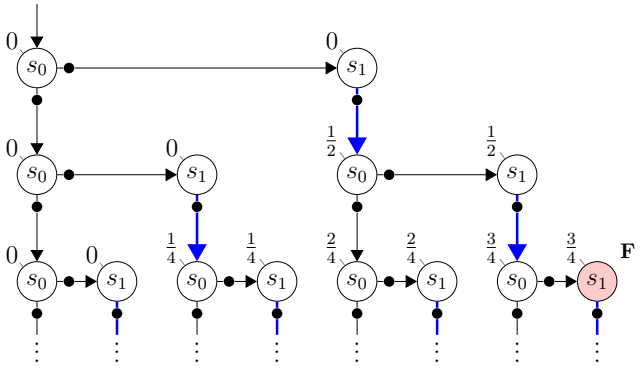


FIGURE 7.6: Instrumentation of Figure 7.4 for the order $s_0 \overset{\cdots}{\longrightarrow} s_1 \prec s_0 \overset{\cdots}{\longrightarrow} s_0$ and bound $\mathbf{p} = \frac{3}{5}$ — the *most* probable paths to the target are counted first.

the state. We have highlighted backtracking transitions.

We mention that the structure of instrumentations may be significantly affected by the chosen transition ordering. For example, if we consider an alternative transition ordering to Figure 7.5 we end up with the infinite-state instrumentation depicted in Figure 7.6. Clearly, these instrumentations are very different in terms of structure. An interesting observation is that there is no path in the instrumentation $M^{\frac{3}{5}}$ that is depicted in Figure 7.5 that explores *all* of M's paths to the target state, s_1, whereas there is such a path in the instrumentation that is depicted in Figure 7.6.

Finally, we observe that an instrumentation $M^{\mathbf{p}}$ of M, from a behavioural perspective, *only* differs from M in states where a probabilistic choice occurs or in states satisfying $\mathbf{F}$. That is, in the absence of probabilistic behaviour, the non-determinism between possible exploration steps in a state $\langle \pi, mass \rangle$ of $M^{\mathbf{p}}$ is identical to the non-determinism in the state $\text{LAST}(\pi)$ of M. This property will be exploited when we lift the instrumentation

function to the level of probabilistic programs in Section 7.3. A special case occurs when we instrument *non-probabilistic* MDPs M. It is easy to see that instrumentations of such MDPs differ only from the original MDP in their behaviour in target states. Target states in instrumentations of non-probabilistic MDPs always have a self-loop — this is because backtracking transitions cannot occur in instrumentations in the absence of probabilistic choice in M.

7.2.4 Soundness

Equation (7.2) on page 134 informally stated the main soundness requirement of the model-level instrumentation process. We now formalise this requirement:

Theorem 7.8 (Model-level soundness). *Let M be an MDP and let $M^{\mathbf{p}}$ be an instrumentation of M for some bound, $\mathbf{p} \in [0,1[\cap \mathbb{Q}$. We have that*

$$\mathit{Reach}^{+}(M^{\mathbf{p}}) \quad \textit{if and only if} \quad \mathit{Prob}^{+}(M) > \mathbf{p}\ .$$

Proof. See Section A.3.1 on page 236. □

Theorem 7.8 yields the anticipated result: it allows us to soundly reason about quantitative properties of MDPs by verifying non-probabilistic safety properties of non-probabilistic models. We will illustrate this via our running example:

Example 7.9. *Consider again the MDP M depicted in Figure 7.2. Suppose we employ the instrumentation loop depicted in Figure 7.1 to compute $\mathit{Prob}^{+}(M)$. Recall that we use an interval, $\mathbf{I}$, to describe possible values of $\mathit{Prob}^{+}(M)$. Initially, $\mathbf{I} = [0,1]$. We first pick $\frac{1}{2} \in \mathbf{I}$, and construct $M^{\frac{1}{2}}$. We showed in Example 7.7 that $\mathit{Reach}^{+}(M^{\frac{1}{2}})$ holds and, due to Theorem 7.8, this means that $\mathit{Prob}^{+}(M) \in\ ]\frac{1}{2}, 1]$. We therefore let $\mathbf{I} =]\frac{1}{2}, 1]$.*

Following the definition of $L^{\mathbf{p}}$ in Definition 7.2.2, it is easy to see that $\mathit{Reach}^{+}(M^{\mathbf{p}})$ holds for every $\mathbf{p} < \frac{13}{25}$ and fails to hold for every $\mathbf{p} \geq \frac{13}{25}$. That is, if we pick a new bound $\frac{3}{4} \in \mathbf{I}$, as $\frac{3}{4} \geq \frac{13}{25}$, we will find $\mathit{Reach}^{+}(M^{\frac{3}{4}})$ is safe and let $\mathbf{I} =]\frac{1}{2}, \frac{3}{4}]$. Subsequent iterations of the instrumentation loop with $\mathbf{p} = \frac{5}{8}$, $\frac{9}{16}$, $\frac{17}{32}$, $\frac{33}{64}$ and $\frac{67}{128}$ will result in increasingly tight intervals $\mathbf{I} =]\frac{1}{2}, \frac{5}{8}]$, $]\frac{1}{2}, \frac{9}{16}]$, $]\frac{1}{2}, \frac{17}{32}]$, $]\frac{33}{64}, \frac{17}{32}]$ and $]\frac{33}{64}, \frac{67}{128}]$, respectively.

We remark that in practice it may be possible to improve the bounds established during an iteration of the instrumentation loop by analysing, say, the counter-examples or proofs returned by the safety checker. Doing this may also lead to precise bounds instead of an interval.

7.2.5 Alternative Instrumentation Schemes

As discussed in Section 7.2.1, to verify a probabilistic property via the verification of *non-probabilistic* properties on instrumentations, we need paths of the instrumentations $M^{\mathbf{P}}$ to correspond to sets of paths of the original MDP M. We already mentioned there are many ways in which we could do this and in this section we briefly discuss alternatives to our search-based instrumentation scheme. We categorise these alternatives into *sequential* and *parallel* instrumentation schemes.

Sequential instrumentation In a sequential instrumentation the paths of M are explored one-by-one. The analogy used in this thesis is that paths of $M^{\mathbf{P}}$ perform a *search* over different resolutions of probabilistic choice in M. The main distinction between different sequential instrumentation methods is the search strategy with which they traverse the state space of M. The instrumentation scheme defined in Section 7.2.2 has many similarities to a depth-first search. This search has the advantage that, due to the way backtracking is defined, no special treatment is necessary to deal with non-deterministic choice. A disadvantage is that backtracking itself is quite complicated.

The main alternative search strategy we considered avoids this complication by exploring a new path of M from scratch every time a target state is encountered. However, with such a strategy, it is no longer automatically guaranteed that we are exploring paths of M that belong to the same pure strategy of M between different runs of M — we need to make a special provision for dealing with non-deterministic choice. It is also more involved to enforce that paths are explored in the desired order in this setting.

Parallel instrumentation We could also consider exploring multiple paths of M *simultaneously* in $M^{\mathbf{P}}$. For example, we could let $M^{\mathbf{P}}$ be a *concurrent* program and, for every possible resolution of probabilistic choice, the instrumentation would spawn a new

thread. The benefit of a parallel approach over the sequential approach is that no backtracking is required and, hence, we need not be concerned with paths that do not reach the target. The main issue of the parallel approach is that model checking instrumentations with potentially an unbounded number of threads is challenging in practice.

An analogous semantics could potentially be achieved with instrumentations that are not concurrent, by equipping $M^{\mathbf{p}}$ with symbolic data structures and by transforming the operations in M into symbolic operations in $M^{\mathbf{p}}$. This may be particularly useful if the all paths of M correpsonding to a single path of $M^{\mathbf{p}}$ are very similar in structure. For example, we could guarantee that all paths match in terms of the control-flow by some formal notion akin to *data independence* [Wol78].

7.3 Program-level Instrumentation

In Section 7.2, we presented an instrumentation method for MDPs which, in principle, can be applied to *any* type of system with MDP semantics. However, due to the complexity of software, for many probabilistic programs, P, it is not tractable to directly generate instrumentions $[\![P]\!]^{\mathbf{p}}$ of the MDP, $[\![P]\!]$, using Section 7.2. In practice, it is more appropriate to directly define instrumentation as a transformation of programs. To this end, in this section, we lift the model-level instrumentation function defined in the previous section to programs — as was originally proposed in the introduction of this chapter.

Our aim is to define an instrumentation function

$$\langle P, \mathbf{p} \rangle \mapsto P^{\mathbf{p}} \quad : \quad \mathrm{PROG} \times \mathbb{Q} \to \mathrm{PROG} \ ,$$

in such a way that the MDP-semantics of a program-level instrumentation of a probabilistic program P, i.e. $[\![P^{\mathbf{p}}]\!]$, behaves like the model-level instrumentation of the MDP-semantics of P, i.e. $[\![P]\!]^{\mathbf{p}}$. Moreover, we require our instrumented programs to be such that they can be verified with existing verification tools for non-probabilistic software.

This is not as easy as it may first appear. The model-level definition of instrumentation in Definition 7.6 augments the *states* of the instrumented model, $[\![P]\!]^{\mathbf{p}}$, with *paths* of

the original model, $[\![P]\!]$. Although our definition of programs does not restrict the types of variables we can use in P^{PP}, adding a variable to P^{PP} that directly represents *paths* of $[\![P]\!]$ results in an instrumented program that no safety checker can verify in practice. Furthermore, backtracking transitions (see Definition 7.6) have relatively complicated semantics, and it is therefore difficult to express these semantics with a single control-flow step in ANSI-C. We therefore sacrifice the simplicity of Definition 7.6 and restrict our instrumented programs to data structures and semantics that safety checkers can deal with in practice. That is, to represent paths of $[\![P]\!]$, we will use data structures such as arrays and we will realise backtracking behaviour via multiple control-flow edges.

This section is structured as follows. In Section 7.3.1, we introduce an assumption on probabilistic programs. Then, in Section 7.3.2 up to 7.3.5 we will define our program-level instrumentation function. Finally, Section 7.3.6 is dedicated to showing our program-level instrumentation is sound.

7.3.1 Assumption

Our formal definition of programs, i.e. Definition 4.1, is quite liberal. There is no restriction on the variable types we allow or the semantics we may give to control-flow edges. In this section, we introduce an assumption on the probabilistic behaviour in programs. We first make an assumption to simplify the definition of a transition ordering:

Assumption 7.10. *Let* $P = \langle\langle \mathscr{L}, \mathscr{E}\rangle, \{\mathscr{L}_{\mathrm{N}}, \mathscr{L}_{\mathrm{P}}, \mathscr{L}_{\mathrm{B}}\}, \ell_i, \mathscr{L}_{\mathrm{T}}, \mathscr{L}_{\mathrm{C}}, \mathit{Var}, \mathit{Sem}\rangle$ *be a probabilistic program. For all probabilistic locations* $\ell \in \mathscr{L}_{\mathrm{P}}$ *there is precisely one variable,* $\textsc{LValue}_\ell \in \mathit{Var}$*, that changes value in transitions from* ℓ*. Moreover, we also assume that for every probabilistic location,* $\ell \in \mathscr{L}_{\mathrm{P}}$*, we have a total order*

$$\textsc{Order}_\ell \subseteq \textsc{Type}(\textsc{LValue}_\ell) \times \textsc{Type}(\textsc{LValue}_\ell) \,,$$

on values of $\textsc{LValue}_\ell$ *and a function*

$$\textsc{Mult}_\ell : \mathscr{U}_{\mathit{Var}} \times \textsc{Type}(\textsc{LValue}_\ell) \to \mathbb{Q} \,,$$

which yields for every $u \in \mathscr{U}_{Var}$ *and* val $\in$ TYPE(LVALUE$_\ell$) *the probability with the program assigns* val *to* LVALUE$_\ell$ *in state* $\langle \ell, u \rangle$.

We remark that in practice we also require ORDER$_\ell$ and MULT$_\ell$ to be expressible in terms of ANSI-C expressions such that we can implement the instrumentation function as a source code transformation.

The first condition of the assumption ensures that each probabilistic choice only affects the value of one variable in the program. In practice, all probabilistic assignments either satisfy this assumption or can be rewritten in such a way that this requirement holds. Essentially, the only complication occurs for assignments such as `*ptr=coin(1,2)`, where the left-hand side of the assignment is not a primitive variable. We split such an assignment into two assignments `tmp=coin(1,2)` and `*ptr=tmp` and add an auxiliary variable, `tmp`, to ensure we satisfy the assumption.

The main benefit of our assumption is that we can define a simple transition ordering on probabilistic programs by only comparing the value of LVALUE$_\ell$ in the target state of transitions from probabilistic locations.

Definition 7.11. *Let* $P = \langle \langle \mathscr{L}, \mathscr{E} \rangle, \{\mathscr{L}_{\mathrm{N}}, \mathscr{L}_{\mathrm{P}}, \mathscr{L}_{\mathrm{B}}\}, \ell_i, \mathscr{L}_{\mathrm{T}}, \mathscr{L}_{\mathrm{C}}, \mathit{Var}, \mathit{Sem} \rangle$ *be a probabilistic program. We define the relation:*

$$\preceq_{[\![P]\!]} \subseteq ((\mathscr{L} \times \mathscr{U}_{Var}) \times (\mathbb{D}(\mathscr{L} \times \mathscr{U}_{Var})) \times (\mathscr{L} \times \mathscr{U}_{Var}))^2 \ ,$$

as the reflexive closure of the relation which includes a tuple of $[\![P]\!]$*-transitions,*

$$\langle \langle \ell_1, u_1 \rangle \xrightarrow{\lambda_1} \langle \ell_1', u_1' \rangle, \langle \ell_2, u_2 \rangle \xrightarrow{\lambda_2} \langle \ell_2', u_2' \rangle \rangle \ ,$$

if and only if the source state match (i.e. $\langle \ell_1, u_1 \rangle = \langle \ell_2, u_2 \rangle$*), the distributions match (i.e.* $\lambda_1 = \lambda_2$*), the source location is a probabilistic location (i.e.* $\ell_1 \in \mathscr{L}_{\mathrm{P}}$*) and if* $u_1 \neq u_2$ *and* $\langle u_1(\text{LVALUE}_{\ell_1}), u_2(\text{LVALUE}_{\ell_1}) \rangle \in \text{ORDER}_{\ell_1}$.

For $\preceq_{[\![P]\!]}$ to be a transition ordering on $[\![P]\!]$ we need to ensure that two transitions are comparable assuming they have a matching source state and distributions. This is

trivial for locations that are not probabilistic, as the distribution would necessarily be a point distribution. This implies that the target state of the two transitions must also be the same and the transitions are comparable in $\preceq_{[\![P]\!]}$ due to $\preceq_{[\![P]\!]}$'s reflexive closure.

For probabilistic locations, ℓ, the requirement holds if Assumption 7.10 holds. That is, if the source state and distributions match, then by our assumption the target state can only differ in their value for LVALUE_ℓ and, as ORDER_ℓ is a *total* order, it must be that the two transitions are comparable by the definition of $\preceq_{[\![P]\!]}$.

We further illustrate the assumption by means of an example:

Example 7.12. *Consider a program* $P = \langle\langle \mathscr{L}, \mathscr{E}\rangle, \{\mathscr{L}_{\text{N}}, \mathscr{L}_{\text{P}}, \mathscr{L}_{\text{B}}\}, \ell_i, \mathscr{L}_{\text{T}}, \mathscr{L}_{\text{C}}, \mathit{Var}, \mathit{Sem}\rangle$ *with a single probabilistic location* $P1 \in \mathscr{L}_{\text{P}}$ *induced by an assignment* `x=coin(1,3)` *where* $\text{TYPE}(\texttt{x})$ *is Boolean. We clearly have that* $\text{LVALUE}_{P1} = \texttt{x}$*, as* `x` *is the* only *variable that is changed by the assignment. For* ORDER_{P1} *we take the total order on* $\text{TYPE}(\texttt{x})$ *induced by the standard ANSI-C operator,* `<=`*. Finally, for every* $u \in \mathscr{U}_{\mathit{Var}}$ *and* $\text{val} \in \text{TYPE}(\texttt{x})$, $\text{MULT}_{P1}(u, \text{val})$ *yields* $\frac{1}{3}$ *if* $\text{val} = \texttt{true}$ *and* $\frac{2}{3}$ *if* $\text{val} = \texttt{false}$.

Having clarified the assumption we make on probabilistic programs, we can now start the definition of our program-level instrumentation function. This definition is defined over various sections. We first define the variables of instrumentations in Section 7.3.2. Then, in Section 7.3.3, we define the control-flow of instrumentations. To complete the definition we present the semantics of instrumentations in Section 7.3.4. Finally, we combine these definitions in Section 7.3.5.

7.3.2 Variables

We start by defining the set of variables, $\mathit{Var}^{\mathbf{p}}$, of any program-level instrumentation of a probabilistic program, P. As discussed, $[\![P^{\mathbf{p}}]\!]$ should behave like $[\![P]\!]^{\mathbf{p}}$ and, to do this, the variables we include in $\mathit{Var}^{\mathbf{p}}$ indirectly encode the state space of $[\![P]\!]^{\mathbf{p}}$:

Definition 7.13. *Let* $P = \langle\langle \mathscr{L}, \mathscr{E}\rangle, \{\mathscr{L}_{\text{N}}, \mathscr{L}_{\text{P}}, \mathscr{L}_{\text{B}}\}, \ell_i, \mathscr{L}_{\text{T}}, \mathscr{L}_{\text{C}}, \mathit{Var}, \mathit{Sem}\rangle$ *be a probabilistic program and let* $\mathbf{p} \in \mathbb{Q}$ *be a bound. Let* $\mathit{Var}^{\mathbf{p}}$ *be the set of variables consisting of:*

– *the variables in Var,*

– *rational variables* `mass` *and* `prob`,
– *a natural variable* `index`,
– *an infinite array* `loc` *of locations,*[2]
– *for every* $var \in Var \cup \{\texttt{prob}\}$, *an infinite array* `copy_`var *with subtype* $\textsc{Type}(var)$,
– *for every* $\ell \in \mathscr{L}_{\text{P}}$ *an infinite array* $\texttt{res}_\ell$ *with subtype* $\textsc{Type}(\textsc{LValue}_\ell)$, *and*
– *a Boolean variable* `skip` *and, for all* $\ell \in \mathscr{L}_{\text{P}}$, *a variable* $\texttt{tmp}_\ell$ *of type* $\textsc{Type}(\textsc{LValue}_\ell)$.

Let us explain the definition of $Var^{\mathbf{P}}$. Recall that states of $[\![P]\!]^{\mathbf{P}}$, $\langle \pi, mass \rangle$, comprise a path component ($\pi \in FinPath_{[\![P]\!]}^{\{\ell_i\} \times \mathscr{U}_{Var}}$) and a rational component ($mass \in \mathbb{Q}$). The rational component is easily represented in $[\![P^{\mathbf{P}}]\!]$'s state space with the rational variable `mass`. Representing the path component with variables that verification tools can deal with in practice is not that simple. We consider Definition 7.6 in more detail and identify what information contained in $[\![P]\!]$'s paths we actually need to mimic the behaviour of $[\![P]\!]^{\mathbf{P}}$ in the program-level instrumentation. We first note that, for a program-level instrumentation, it helps to store only the information contained in paths that we actually need — not so much because this would mean there is less information to store, but rather because it means we do not have to update our path component during every exploration step.

Observe that any *explorative* transition in $[\![P]\!]^{\mathbf{P}}$ is of the form

$$\langle \pi, mass \rangle \to \langle \pi \to \langle \ell, u \rangle, mass \rangle \ ,$$

and, to know which explorative transitions are available in a state $\langle \pi, mass \rangle$ we *only* need to know the last state of π. This last state is, in turn, a state of $[\![P]\!]$. The variables, Var, of P itself are included in $Var^{\mathbf{P}}$ to allow us realise exploration.

When we need to start *backtracking* in a state $\langle \pi, mass \rangle$ of $[\![P]\!]$, we may choose to backtrack to any probabilistic choice in π. To enable this in $P^{\mathbf{P}}$, we store some information about every probabilistic location that occurs in π. As we cannot put a bound on how many probabilistic locations may appear in π, we use various infinite arrays to store

[2] In practice we use ANSI-C's `enum` type to represent the locations in $\mathscr{L}_{\text{P}} \setminus \mathscr{L}_{\text{T}}$.

information about each such location and we use an integer variable `index` to index these arrays. Precisely what information we need depends on the information we need to realise backtracking.

Recall that *backtracking* transitions are of the form

$$\langle \pi, \mathit{mass} \rangle \to \langle \pi', \mathit{mass} + \textsc{Prob}_M(\pi) \rangle$$

where π' is a path of $[\![P]\!]$. By the definition of backtracking transitions and by the definition of $\preceq_{[\![P]\!]}$ it must be that the penultimate state of π' is a $[\![P]\!]$-state, $\langle \ell, u \rangle$, where ℓ is a probabilistic location. That is, other location types never satisfy our ordering constraint and hence we only ever backtrack to probabilistic locations.

When we backtrack, we need to "undo" a number of transitions. Effectively, we need to restore the state of P to what it was prior to making the probabilistic choice we are backtracking to. To restore the *program location* of $[\![P]\!]$ we add to $\mathit{Var}^{\mathbf{p}}$ the array `loc`. Similarly, to restore the *data state* of $[\![P]\!]$ in $[\![P^{\mathbf{p}}]\!]$ we add, for each $\mathit{var} \in \mathit{Var}$, an array `copy_`$\mathit{var}$ to store the value of P's variables. To be able to enforce the transition ordering in $P^{\mathbf{p}}$, for every $\ell \in \mathscr{L}_{\mathrm{P}}$, we also add an array $\texttt{res}_\ell$ that stores the value of $\textsc{LValue}_\ell$ directly *after* a probabilistic choice.

Finally, observe that during backtracking transition we also need to add the probability $\textsc{Prob}_M(\pi)$ to `mass`. We store the probability of the current path in another rational variable `prob` $\in \mathit{Var}^{\mathbf{p}}$ as it would be expensive to compute $\textsc{Prob}_M(\pi)$ on-the-fly in $P^{\mathbf{p}}$ for every backtracking transition. The remaining variables in Definition 7.13 are temporary variables, which helps simplify our definition of semantics later.

7.3.3 Control-flow

There are considerable benefits to defining our instrumentation on the level of programs. For example, the vast majority of locations in $[\![P]\!]$ are likely to be non-probabilistic. As remarked at the end of Section 7.2.2, the behaviour of $[\![P]\!]^{\mathbf{p}}$ is identical to that of $[\![P]\!]$ in the absence of probabilistic choice and this is something we can exploit.

That is, because we only store path information when we visit probabilistic program

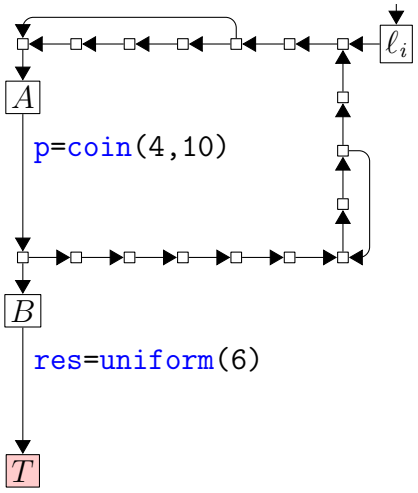


FIGURE 7.7: The control-flow of a probabilistic program, P.

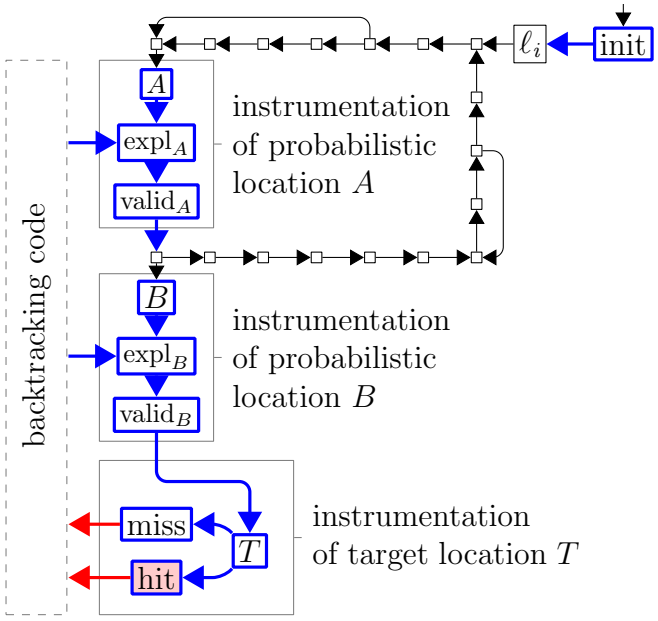


FIGURE 7.8: A schematic control-flow graph of an *instrumentation*, $P^{\mathbf{P}}$, of the program depicted in Figure 7.7.

locations, we only need to update the variables in $Var^{\mathbf{P}} \setminus Var$ when are making a probabilistic choice or when we are backtracking. This means that, in practice, we can embed most of P's code in $P^{\mathbf{P}}$ unmodified. In particular, if $\mathscr{L}_{\text{P}} = \emptyset$, our instrumentations are, besides cosmetic changes, identical to P.

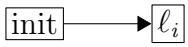
We illustrate our point with a control-flow graph of a program P in Figure 7.7 and an informal description of the control-flow graph of its instrumentation, $P^{\mathbf{P}}$, in Figure 7.8. The control-flow edges in Figure 7.8 that are not highlighted have not been affected. The instrumentation $P^{\mathbf{P}}$ only differs from P in the probabilistic locations "A" and "B" and $\mathscr{L}_{\text{T}}$, the rest of P's control-flow and semantics are left intact. We do remark that a significant amount of control-flow is added to realise backtracking — a backtacking transition is no longer realised in a single step. We also note backtracking code adds a control-flow loop that is not in the original program.

We now formalise the control-flow of program-level instrumentations. That is, we will define the structure of instrumented programs (we will define formal semantics in the next section). To improve our presentation we define some of this control-flow pictorially.

Definition 7.14. *Let* $P = \langle\langle \mathscr{L}, \mathscr{E}\rangle, \{\mathscr{L}_{\text{N}}, \mathscr{L}_{\text{P}}, \mathscr{L}_{\text{B}}\}, \ell_i, \mathscr{L}_{\text{T}}, \mathscr{L}_{\text{C}}, Var, Sem\rangle$ *be a probabilis-*

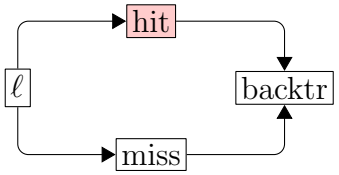
tic program and let $\mathbf{p} \in \mathbb{Q}$ *be a bound. Let* $\langle \mathscr{L}^{\mathbf{p}}, \mathscr{E}^{\mathbf{p}} \rangle$ *be the control-flow graph and let* $\{\mathscr{L}_{\mathrm{N}}^{\mathbf{p}}, \mathscr{L}_{\mathrm{P}}^{\mathbf{p}}, \mathscr{L}_{\mathrm{B}}^{\mathbf{p}}\}$ *a partition of* $\mathscr{L}^{\mathbf{p}}$ *into assignment, probabilistic and branching locations, defined as follows:*

- *We include all locations* $\ell \in \mathscr{L}$ *in* $\mathscr{L}^{\mathbf{p}}$*. Moreover, all locations that are not probabilistic locations or target locations of* P *retain their location type and their outgoing edges in* $\mathscr{E}^{\mathbf{p}}$*.*
- *We include an* instrumented initialisation step*. That is, we add an assignment location "*init*"*$\in \mathscr{L}_{\mathrm{N}}^{\mathbf{p}}$ *and a control-flow edge*



Informally, an instrumentation of P *will use "*init*" as its initial location and, from this location, it will initalise the instrumentation variables* `mass`*,* `prob` *and* `index` *prior to moving to* P*'s initial location,* ℓ_i*.*

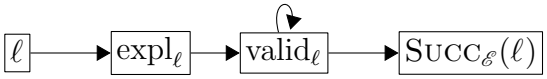
- *For every target location* $\ell \in \mathscr{L}_{\mathrm{T}}$ *we include* instrumented targets*. That is, we add to* $\mathscr{L}^{\mathbf{p}}$ *assignment locations "*miss*",* "hit*"* $\in \mathscr{L}_{\mathrm{N}}^{\mathbf{p}}$ *and a branching location "*backtr*"* $\in \mathscr{L}_{\mathrm{B}}^{\mathbf{p}}$ *and we turn every target location* $\ell \in \mathscr{L}_{\mathrm{T}}$ *into a branching location in* $P^{\mathbf{p}}$*. Finally, we add the control-flow*



Informally, the idea is that, once we get to ℓ*, the instrumentation should only reach a target location if the probability mass it has accumulated exceeds* $\mathbf{p}$*. To this end, will we go to "*hit*" only if the value of* `mass` *and* `prob` *combined exceeds* $\mathbf{p}$ *and otherwise we will go to "*miss*". In both "*hit*" and "*miss*", we will add* `prob` *to* `mass` *before proceeding with backtracking in "*backtr*".*

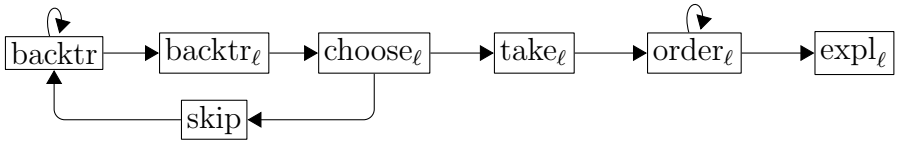
- *For every probabilistic location* $\ell \in \mathscr{L}_{\mathrm{P}} \setminus \mathscr{L}_{\mathrm{T}}$ *that is not a target location in* P *we include* exploration code*. We do this by making* ℓ *an assignment location and we add an additional assignment location "*expl_ℓ*"* $\in \mathscr{L}_{\mathrm{N}}^{\mathbf{p}}$ *and a branching location "*valid_ℓ*"*

$\in \mathscr{L}_{\mathrm{B}}^{\mathbf{P}}$. *We also add the control-flow*



Informally, in ℓ*, we increment* `index` *by 1 and store the path information we need for backtracking in the appropriate arrays. We also resolve the probabilistic choice in* ℓ *non-deterministically. Then, in "*expl_ℓ*", we update the probability in* `prob`*. Finally, in "*valid_ℓ*", we check whether we have made a valid probabilistic choice by checking* `prob` *is non-zero.*

– *Finally, for every probabilistic location* $\ell \in \mathscr{L}_{\mathrm{P}} \setminus \mathscr{L}_{\mathrm{T}}$ *that is not a target location in* P *we include* backtracking code. *That is, we add assignment location "*skip*"* $\in \mathscr{L}_{\mathrm{N}}^{\mathbf{P}}$*, assignment locations "*backtr_ℓ*", "*take_ℓ*",* $\in \mathscr{L}_{\mathrm{N}}^{\mathbf{P}}$*, and branching locations "*choose_ℓ*", "*order_ℓ*"* $\in \mathscr{L}_{\mathrm{B}}^{\mathbf{P}}$ *and the following control-flow:*



*Backtracking is realised by iterating over the probabilistic choices made so far in a loop and by choosing non-deterministically whether to backtrack to a particular probabilistic choice. This loop start with "*backtr*", where we look at* `loc` *to see which location is responsible for the most recent probabilistic choice under consideration and jump to the appropriate "*backtr_ℓ*". In "*backtr_ℓ*" we restore the variables in Var to the value they had before we took the probabilistic transition, and then, in "*choose_ℓ*", we jump non-deterministically to either "*take_ℓ*" or "*skip*".*

In "take_ℓ*", we pick a new resolution of the probabilistic transition under consideration, after which "*order_ℓ*" jumps to the exploration code under the provision that the chosen resolution of probabilistic choice in* P *respects the transition ordering.*

*Alternatively, "*skip*" skips the current probabilistic choices and modifies instrumentation variables such that we can backtrack to other choices.*

We clarify Definition 7.14 with an example.

Example 7.15. *Consider the program P depicted in Figure 4.5 (page 58). The control-flow of the* program-level *instrumentation $P^{\frac{2}{3}}$ of P is depicted in Figure 7.9. Recall that for presentational convenience we often omit self-loops on branching locations. We also depict an instrumentation of the program in Figure 4.2 (page 55) in Figure 7.3.*

7.3.4 Semantics

To complete the definition of our program-level instrumentations we need to define their semantics — we formalise the informal discussion on semantics in the previous section:

Definition 7.16. *Let $P = \langle\langle \mathscr{L}, \mathscr{E}\rangle, \{\mathscr{L}_{\mathrm{N}}, \mathscr{L}_{\mathrm{P}}, \mathscr{L}_{\mathrm{B}}\}, \ell_i, \mathscr{L}_{\mathrm{T}}, \mathscr{L}_{\mathrm{C}}, Var, Sem\rangle$ be a probabilistic program and let $\mathbf{p} \in \mathbb{Q}$ be a bound. Let $Var^{\mathbf{p}}$ be a set of variables as defined in Definition 7.13 and let $\langle \mathscr{L}^{\mathbf{p}}, \mathscr{E}^{\mathbf{p}}\rangle$ and $\{\mathscr{L}_{\mathrm{N}}, \mathscr{L}_{\mathrm{P}}, \mathscr{L}_{\mathrm{B}}\}$ be a control-flow graph and a partition of $\mathscr{L}^{\mathbf{p}}$ as defined in Definition 7.14. We let*

$$Sem^{\mathbf{p}} : \mathscr{E}^{\mathbf{p}} \to (\mathscr{U}_{Var^{\mathbf{p}}} \to \mathbb{PD}(\mathscr{U}_{Var^{\mathbf{p}}}))$$

be the function such that, for every $u \in \mathscr{U}_{Var^{\mathbf{p}}}$, we have:

- *For all $\ell, \ell' \in \mathscr{L}$ where ℓ is not a probabilistic location or a target location in P we let $Sem^{\mathbf{p}}(\langle \ell, \ell'\rangle)$ be $Sem(\langle \ell, \ell'\rangle)$.*[3]
- *For the instrumented initialisation step we let*
  - *$Sem^{\mathbf{p}}(\langle \mathrm{init}, \ell_i\rangle)(u) = \{[u']\}$ where u' is u with*
    - *$u'(\texttt{mass}) = 0$, $u'(\texttt{prob}) = 1$ and $u'(\texttt{index}) = 0$.*
- *For the instrumented targets $\ell \in \mathscr{L}_{\mathrm{T}}$ we let*
  - *$Sem^{\mathbf{p}}(\langle \ell, \mathrm{hit}\rangle)(u)$ is $\{[u]\}$ if $u(\texttt{mass}) + u(\texttt{prob}) > \mathbf{p}$ and $\emptyset$ otherwise,*
  - *$Sem^{\mathbf{p}}(\langle \ell, \mathrm{miss}\rangle)(u)$ is $\{[u]\}$ if $u(\texttt{mass}) + u(\texttt{prob}) \leq \mathbf{p}$ and $\emptyset$ otherwise, and*
  - *$Sem^{\mathbf{p}}(\langle \mathrm{miss}, \mathrm{backtr}\rangle)(u) = Sem^{\mathbf{p}}(\langle \mathrm{hit}, \mathrm{backtr}\rangle)(u) = \{[u']\}$ where u' is u with*

[3] We lift the semantics over $\mathscr{U}_{Var}$ to semantics over $\mathscr{U}_{Var^{\mathbf{p}}}$ in the obvious way.

FIGURE 7.9: Instrumentation of the program depicted in Figure 4.5 (page 58).

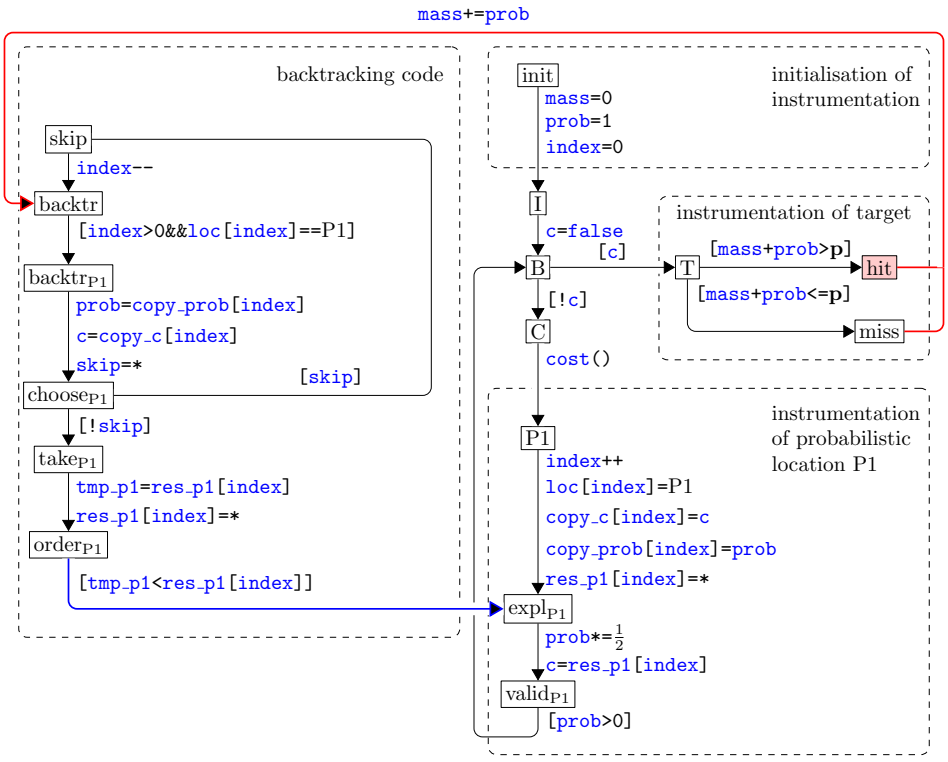


FIGURE 7.10: Instrumentation of the program depicted in Figure 4.2 (page 55).

- $u'(\texttt{mass}) = u(\texttt{mass}) + u(\texttt{prob})$.

– *For every probabilistic location* $\ell \in \mathscr{L}_{\mathrm{P}} \setminus \mathscr{L}_{\mathrm{T}}$ *we let*

- $Sem^{\mathbf{P}}(\langle \ell, \mathrm{expl}_\ell \rangle)(u) = \{[u^{\mathrm{val}}] \mid \mathrm{val} \in \textsc{Type}(\textsc{LValue}_\ell)\}$ *where* u^{val} *is* u *with*
  - $u^{\mathrm{val}}(\texttt{index}) = u(\texttt{index}) + 1$,
  - $u^{\mathrm{val}}(\texttt{res}_\ell)\,[u(\texttt{index}) + 1] = \mathit{val}$,
  - $u^{\mathrm{val}}(\texttt{loc})\,[u(\texttt{index}) + 1] = \ell$,
  - $\forall \mathit{var} \in \mathit{Var} \cup \{\texttt{prob}\} : u^{\mathrm{val}}(\texttt{copy\_}\mathit{var})\,[u(\texttt{index}) + 1] = u(\mathit{var})$,
- $Sem^{\mathbf{P}}(\langle \mathrm{expl}_\ell, \mathrm{valid}_\ell \rangle)(u) = \{[u']\}$ *where* u' *is* u *with*
  - $u'(\textsc{LValue}_\ell) = u(\texttt{res}_\ell)\,[u(\texttt{index})]$,
  - $u'(\texttt{prob}) = u(\texttt{prob}) \cdot \textsc{Mult}_\ell(u'', u(\texttt{res}_\ell)\,[u(\texttt{index})])$, *where* u'' *is* u *restricted to variables in* Var,
- $Sem^{\mathbf{P}}(\langle \mathrm{valid}_\ell, \textsc{Succ}_{\mathscr{E}}(\ell) \rangle)(u)$ *is* $\{[u]\}$ *if* $u(\texttt{prob}) > 0$ *and* $\emptyset$ *otherwise, and*
- $Sem^{\mathbf{P}}(\langle \mathrm{valid}_\ell, \mathrm{valid}_\ell \rangle)(u)$ *is* $\{[u]\}$ *if* $u(\texttt{prob}) \leq 0$ *and* $\emptyset$ *otherwise.*

– *For every probabilistic location* $\ell \in \mathscr{L}_{\mathrm{P}} \setminus \mathscr{L}_{\mathrm{T}}$ *we let*

- $Sem^{\mathbf{P}}(\langle \mathrm{backtr}, \mathrm{backtr}_\ell \rangle)(u)$ *is* $\{[u]\}$ *if* $u(\texttt{index}) > 0 \wedge u(\texttt{loc})\,[u(\texttt{index})] = \ell$ *and* $\emptyset$ *otherwise.*
- $Sem^{\mathbf{P}}(\langle \mathrm{backtr}, \mathrm{backtr} \rangle)(u)$ *is* $\{[u]\}$ *if either* $u(\texttt{index}) \leq 0$ *or there is no* $\ell \in \mathscr{L}_{\mathrm{P}} \setminus \mathscr{L}_{\mathrm{T}}$ *with* $u(\texttt{loc})\,[u(\texttt{index})] = \ell$, *and* $\emptyset$ *otherwise.*
- $Sem^{\mathbf{P}}(\langle \mathrm{backtr}_\ell, \mathrm{choose}_\ell \rangle)(u)$ *is* $\{[u^{\mathrm{val}}] \mid \mathrm{val} \in \mathbb{B}\}$ *with*
  - $u^{\mathrm{val}}(\texttt{skip}) = \mathit{val}$,
  - $\forall \mathit{var} \in \mathit{Var} \cup \{\texttt{prob}\} : u^{\mathrm{val}}(\mathit{var}) = u(\texttt{copy\_}\mathit{var})\,[u(\texttt{index})]$,
- $Sem^{\mathbf{P}}(\langle \mathrm{choose}_\ell, \mathrm{take}_\ell \rangle)(u)$ *is* $\{[u]\}$ *if* $\neg u(\texttt{skip})$ *and* $\emptyset$ *otherwise.*
- $Sem^{\mathbf{P}}(\langle \mathrm{choose}_\ell, \mathrm{skip} \rangle)(u)$ *is* $\{[u]\}$ *if* $u(\texttt{skip})$ *and* $\emptyset$ *otherwise.*
- $Sem^{\mathbf{P}}(\langle \mathrm{take}_\ell, \mathrm{order}_\ell \rangle)(u) = \{[u^{\mathrm{val}}] \mid \mathrm{val} \in \textsc{Type}(\textsc{LValue}_\ell)\}$ *with*
  - $u^{\mathrm{val}}(\texttt{tmp}_\ell) = u(\texttt{res}_\ell)\,[u(\texttt{index})]$,
  - $u^{\mathrm{val}}(\texttt{res}_\ell)\,[u(\texttt{index})] = \mathit{val}$,

- $Sem^{\mathbf{p}}(\langle \text{order}_\ell, \text{expl}_\ell \rangle)(u)$ *is* $\{[u]\}$ *if*
  - $u(\texttt{tmp}_\ell) \neq u(\texttt{res}_\ell)[u(\texttt{index})]$ *and*
  - $\langle u(\texttt{tmp}_\ell), u(\texttt{res}_\ell)[u(\texttt{index})] \rangle \in \textsc{Order}_\ell$,

  and $\emptyset$ *otherwise,*
- $Sem^{\mathbf{p}}(\langle \text{order}_\ell, \text{order}_\ell \rangle)(u)$ *is* $\{[u]\}$ *if*
  - $u(\texttt{tmp}_\ell) = u(\texttt{res}_\ell)[u(\texttt{index})]$ *or*
  - $\langle u(\texttt{tmp}_\ell), u(\texttt{res}_\ell)[u(\texttt{index})] \rangle \notin \textsc{Order}_\ell$,

  and $\emptyset$ *otherwise,*
- *Finally,* $Sem^{\mathbf{p}}(\langle \text{skip}, \text{backtr} \rangle)(u)$ *is* $\{[u']\}$ *where* u' *is* u *with*
  - $u'(\texttt{index}) = u(\texttt{index}) - 1$.

To further illustrate our semantics we refer back to Example 7.15. The control-flow of the instrumentation in this example is annotated with syntax labels which give an intuitive account of the semantics of this instrumentation.

7.3.5 Program-level Instrumention Function

For clarity we combine the definition of the variables, control-flow and semantics in the previous sections into a single definition:

Definition 7.17 (Program-level Instrumentation). *Let* P *be a probabilistic program* $\langle\langle \mathscr{L}, \mathscr{E} \rangle, \{\mathscr{L}_{\text{N}}, \mathscr{L}_{\text{P}}, \mathscr{L}_{\text{B}}\}, \ell_i, \mathscr{L}_{\text{T}}, \mathscr{L}_{\text{C}}, Var, Sem \rangle$ *and let* $\mathbf{p} \in [0, 1[\cap \mathbb{Q}$ *be a bound. We let* $P^{\mathbf{p}} = \langle\langle \mathscr{L}^{\mathbf{p}}, \mathscr{E}^{\mathbf{p}} \rangle, \{\mathscr{L}_{\text{N}}^{\mathbf{p}}, \mathscr{L}_{\text{P}}^{\mathbf{p}}, \mathscr{L}_{\text{B}}^{\mathbf{p}}\}, \ell_i^{\mathbf{p}}, \mathscr{L}_{\text{T}}^{\mathbf{p}}, \mathscr{L}_{\text{C}}^{\mathbf{p}}, Var^{\mathbf{p}}, Sem^{\mathbf{p}} \rangle$ *be the probabilistic program defined as:*

- $\langle \mathscr{L}^{\mathbf{p}}, \mathscr{E}^{\mathbf{p}} \rangle$ *is as defined in Definition 7.14,*
- $\{\mathscr{L}_{\text{N}}^{\mathbf{p}}, \mathscr{L}_{\text{P}}^{\mathbf{p}}, \mathscr{L}_{\text{B}}^{\mathbf{p}}\}$ *is as defined in Definition 7.14*
- $\ell_i^{\mathbf{p}} = \text{init}$, $\mathscr{L}_{\text{T}}^{\mathbf{p}} = \{\text{hit}\}$, $\mathscr{L}_{\text{C}}^{\mathbf{p}} = \{\mathscr{L}_{\text{C}}\}$,
- $Var^{\mathbf{p}}$ *is as defined in Definition 7.13 and*
- $Sem^{\mathbf{p}}$ *is as defined in Definition 7.16.*

Like the model-level instrumentations, a program-level instrumentation $P^{\mathbf{p}}$ of a program P is entirely non-probabilistic (i.e. $\mathscr{L}_{\mathrm{P}}^{\mathbf{p}}$ is empty). For model-level instrumentations we observed that small finite models can become infinite-state once instrumented. We remark this added complexity is not reflected in the control-flow of program-level instrumentations: the size of the control-flow graph of $P^{\mathbf{p}}$ grows linearly in the number of probabilistic locations in P. The main complexity of $P^{\mathbf{p}}$ is its data space $\mathscr{U}_{Var^{\mathbf{p}}}$ which contains both rationals and an unbounded number of copies of P's data space $\mathscr{U}_{Var}$ in the form of infinite arrays.

7.3.6 Soundness

We have not yet shown that the program-level instrumentations we have defined in the previous section actually adheres to the soundness criterion first described in Equation 7.2. We further formalise this requirement in the following theorem:

Theorem 7.18 (Program-level soundness). *Let P be any probabilistic program and let $\mathbf{p} \in [0,1[\cap \mathbb{Q}$ be an arbitrary bound. Let $P^{\mathbf{p}}$ be the program-level instrumentation of P for bound $\mathbf{p}$. We have*

$$\mathit{Reach}^{+}([\![P^{\mathbf{p}}]\!]) \quad \textit{if and only if} \quad \mathit{Prob}^{+}([\![P]\!]) > \mathbf{p}\ .$$

Proof. See Section A.3.2 on page 242. □

The proof of Theorem 7.18 relies on the soundness for model-level instrumentations (see Theorem 7.8). That is, we show that $\mathit{Reach}^{+}([\![P^{\mathbf{p}}]\!])$ if and only if $\mathit{Reach}^{+}([\![P]\!]^{\mathbf{p}})$. To do this we use the stuttering equivalence defined in Section 3.20.

7.4 Experimental Results & Extensions

To validate our approach, we have implemented the instrumentation loop described in this chapter in a tool called PROBITY (see Appendix B.2). We first discuss the model checker we use to verify non-probabilistic safety properties of instrumented programs

in Section 7.4.1. Then, in Section 7.4.2, we describe some important adaptations and optimisations we made to the model checking algorithm as well as our instrumentation process. Following this, we will present our main experimental results in Section 7.4.3. In Section 7.4.4, we present a prototype extension called "*template invariants*". Finally, in Section 7.4.5, we will compare our experimental results with the results in Chapter 5.

7.4.1 Model Checking Instrumented Programs

To model check instrumentations we use an implementation of the interpolation-based model checking algorithm in [McM06]. In this section we give a high-level description of this model checking algorithm — we refer the reader to [McM06] for technical details.

The aim of the model checker is either to prove the program is safe (i.e. Reach^+ does not hold) by finding sufficiently strong *inductive invariants* [Flo67, Hoa69], or to demonstrate it is not safe (i.e. Reach^+ holds) by finding a counter-example. The model checker obtains invariants by repeatedly calling an *interpolating decision procedure*. In this context, this decision procedures takes control-flow paths from the initial state, $\ell_i^{\mathbf{P}}$, to a target state, $\ell_t^{\mathbf{P}} \in \mathscr{L}_{\mathrm{T}}^{\mathbf{P}}$, and decides whether such paths are feasible. If a path is feasible, then the decision procedure returns a counter-example. If infeasible, then the decision procedure generates interpolants that prove the infeasibility of the path — these interpolants are used to strengthen the inductive invariants. The model checking algorithm has further mechanisms to deal with loops (called COVER and FORCECOVER in [McM06]). For a discussion on these mechanisms we refer to [McM06]. However, we do remark that these mechanisms are mostly only effective when the invariants found by the interpolating decision procedure are in fact loop invariants. The main data structure used by the model checker is a tree unwinding of the control-flow graph of the program. The model checker stops once it finds a feasible path to a target state or once it obtains invariants that are strong enough to prove no target state is reachable.

The interpolating decision procedure used by our model checker is the one described in [KW09]. This decision procedure can deal with equalities, inequalities and uninterpreted functions. Unfortunately, it cannot deal with any kind of arithmetic unless this arithmetic can be resolved by propagating constants.

7.4.2 Adaptations & Optimisations

For our approach to work on real probabilistic programs in practice we have made some adaptations to the basic model checking and instrumentation process. These are in addition to basic adaptations such as adding support for rationals in the model checker.

Incremental backtracking In practice, we use an incremental version of the backtracking mechanism described in Section 7.3. That is, for a probabilistic location, ℓ, we only copy and restore (an over-approximation of) those variables that may change value before another probabilistic location is encountered. We perform a standard analysis to over-approximate this set of variables for each probabilistic location. Our instrumentations already restore the data state incrementally — i.e. "backtr$_\ell$" precedes "choose$_\ell$".

Feasibility of paths In practice the interpolating decision procedure we employ is not always able to establish that a path is infeasible. To avoid returning false negatives, we use a SAT solver to ensure a path is indeed feasible when the interpolating decision procedure is not able to find a proof. Directly using rational or integer variables in SAT is not possible. Hence, to reason about the feasibility of paths in our instrumentations via SAT, we have adapted the instrumentation process in such a way that the arithmetic over rationals and integers can be eliminated via constant propagation. This means that, instead of resolving a probabilistic choice of the original program with a one-step non-deterministic choice in the instrumented program, we *iterate* over all possible resolutions in a loop and non-deterministically choose for each iteration whether to use it. Moreover, we are currently unable to deal with probabilistic choices where the probabilities depend on the data state because of this reason.

Counter-examples and proofs When the model checker establishes an instrumentation, $P^{\mathbf{p}}$, is not safe it provides a *counter-example* $\langle \ell_i^{\mathbf{p}}, u\rangle \to \ldots \langle \ell_t^{\mathbf{p}}, u'\rangle$. This is a finite path in $[\![P^{\mathbf{p}}]\!]$ from its initial location, $\ell_i^{\mathbf{p}}$, to a target location, $\ell_t^{\mathbf{p}} \in \mathscr{L}_{\mathrm{T}}^{\mathbf{p}}$. By construction of $P^{\mathbf{p}}$, the value of $u'(\texttt{mass}) + u'(\texttt{prob})$ is a lower bound on $Prob^{+}(P)$. Moreover, by definition of $P^{\mathbf{p}}$, this lower bound is strictly greater than $\mathbf{p}$. We therefore use this lower bound instead of $\mathbf{p}$. We remark that, unlike $\mathbf{p}$, this improved lower bound is never a strict

lower bound.

If $P^{\mathbf{p}}$ is safe then the model checker produces a proof in the form of inductive invariants for every control-flow location $\ell \in \mathscr{L}^{\mathbf{p}}$. Now observe that, in program-level instrumentations, the only thing that is affected by the bound, $\mathbf{p}$, is the semantics at locations $\ell \in \mathscr{L}_{\mathrm{T}}$. For every location $\ell \in \mathscr{L}_{\mathrm{T}}$, the condition labelling the edge $\langle \ell, \mathrm{hit} \rangle$ is `[mass+prob>p]`. Moreover, these edges are the only way of reaching $P^{\mathbf{p}}$'s target locations. The idea is that we look at the invariants at locations $\mathscr{L}_{\mathrm{T}}$ to find an upper bound on the value of the sum `mass` + `prob`. If we can show that this sum is always bounded from above by some constant C, then evidently "hit" cannot be reached in P^C either. Moreover, if C is smaller than $\mathbf{p}$, then we have found a tighter upper bound.

We find C in practice by checking for the presence of common invariants such as (`mass` == C_1) and (`prob` < C_2) at every location $\ell \in \mathscr{L}_{\mathrm{T}}$. We remark that we may not always be able to obtain a good value for C. To improve the effectiveness of this optimisation we have adapted the interpolating decision procedure to always return invariants that yield the tightest upper bound on `mass` and `prob`.

Greedy exploration Recall that, in a model-level instrumentation $M^{\mathbf{p}}$, a path of $M^{\mathbf{p}}$ essentially performs a search over the paths of M (see Section 7.2.2). Distinct searches may result in states of $M^{\mathbf{p}}$ that are distinguishable only by the accumulated probability mass. This is illustrated by states s_4 and u_4 in Figure 7.3, page 138 — the search that led to s_4 explored fewer paths of the original MDP and accumulated less probability mass than u_4. The same phenomenon occurs in program-level instrumentations. Due to the changes to the instrumentation described above (see "Feasibility of paths"), each way we can search over the probabilistic choices of P actually corresponds to a distinct control-flow path in the instrumentation $P^{\mathbf{p}}$. The heuristic we propose adapts the order in which the model checker considers control-flow paths of $P^{\mathbf{p}}$ and ensures that paths of $P^{\mathbf{p}}$ that skip the fewest behaviours of P are explored first. The main reason for this is that the invariants we use to show the infeasibility of a control-flow path of $P^{\mathbf{p}}$ may also be sufficient to prove the infeasibility of control-flow paths of $P^{\mathbf{p}}$ that correspond to searches that skip more behaviours of P. The model checking algorithm we use has sophisticated methods to enable invariant reuse (see COVER and FORCECOVER in [McM06]) but, to

	PING	TFTP	NTP	FREI	HERM	MGALE	AMP	FAC	BRP	ZERO	CONS
A	✓	DIV	✓	DIV	LIVE	✓	LIVE	LOOP	DIV	LIVE	LIVE
B	✓	DIV	✓	LIVE	COST	LIVE	LIVE	COST	DIV	COST	DIV
C	COST	COST	COST		LOOP	COST	✓		LIVE		
D	✓				LIVE						

FIGURE 7.11: An overview of the probabilistic programs and properties we considered in Chapter 5 and their applicability with the instrumentation-based approach.

exploit this, the control-flow paths of $P^{\mathbf{p}}$ need to be considered in the order specified by our heuristic.

7.4.3 Experiments & Analysis

We have evaluated our implementation against the probabilistic programs and properties discussed in Chapter 5 (see Appendix C and Section 5.5 for descriptions of these case studies). All experiments were run on an Intel Core Duo 2 7200 with 2GB RAM on Fedora 8. We use a timeout setting of 600 seconds for every experiment. We always pick the bound, $\mathbf{p} \in \mathbf{I}$, precisely in the middle of the interval, $\mathbf{I}$, and we terminate the instrumentation loop whenever the absolute error is smaller than or equal to 10^{-4}. All timings are reported in seconds.

We note that our model checker is run with FORCECOVER enabled (see [McM06]). Moreover, in addition to checking the feasibility of paths to the target location we also check the feasibility of paths ending in a control-flow edge from a branching location. We mention that, due to technical reasons, the source code of the programs checked in this chapter have certain minor alterations compared to Chapter 5. For example, because the interpolating decision procedure we use cannot currently deal with disjunctions we split conditionals in the program where the expression is a disjunction or conjunction. We remark that there are very few of these alterations and they do not affect the property at hand. Moreover, these alterations could easily be automated.

Our main results are presented in Figure 7.11 and 7.12. In Figure 7.11, we show which of the properties and probabilistic programs discussed in Chapter 5 we have managed to compute — we discuss the content of this table in more details below. Of the adaptations

			NORMAL					GREEDY					
			LB/UB ITERS	LB/UB TIME	UNW	INT	TIME	LB/UB ITERS	LB/UB TIME	UNW	INT	TIME	RESULT
PING	A	1	1/1	24%/57%	191	39%	6.48	1/1	35%/55%	195	47%	9.05	[99/1250, 99/1250]
		2	1/1	8%/89%	510	44%	52.5	1/1	43%/55%	445	40%	73.9	[99/15625, 99/15625]
		3	–/–	–/–	–	–	>600	1/1	43%/55%	830	31%	373	[198/390625, 198/390625]
	B	1	0/1	0%/72%	179	36%	4.19	0/1	0%/91%	181	42%	5.21	[0, 0]
		2	1/1	48%/50%	744	39%	360	1/1	28%/70%	479	49%	87.7	[2277/15625, 2277/15625]
	D	1	1/1	40%/52%	189	39%	8.45	1/1	10%/63%	193	43%	10.2	[2277/2500, 2277/2500]
		2	1/1	42%/56%	407	36%	59.6	1/1	5%/91%	463	47%	46.5	[52371/62500, 52371/62500]
		3	1/1	47%/52%	714	30%	253	–/–	–/–	–	–	>600	[1204533/1562500, 1204533/1562500]
NTP	A	1	1/1	34%/51%	188	68%	11.3	1/1	50%/40%	214	72%	16.5	[2/25, 2/25]
		2	2/1	55%/39%	403	81%	60.6	1/1	38%/56%	395	78%	40.8	[96/625, 96/625]
		3	2/1	57%/40%	732	87%	212	2/1	49%/46%	573	80%	104	[3458/15625, 3458/15625]
		4	–/–	–/–	–	–	>600	2/1	48%/47%	751	82%	175	[110784/390625, 110784/390625]
	B	1	1/1	22%/49%	118	49%	5.01	1/1	17%/52%	112	49%	5.23	[23/25, 23/25]
		2	2/1	41%/50%	263	71%	29.6	2/1	26%/59%	195	65%	18.6	[621/625, 621/625]
		3	3/1	49%/47%	564	83%	190	3/1	34%/56%	275	71%	43.5	[15617/15625, 15617/15625]
		4	4/0	99%/0%	1,094	88%	533	4/0	92%/0%	257	72%	36.2	[390609/390625, 1]
		6	–/–	–/–	–	–	>600	4/0	90%/0%	257	68%	38.8	[390609/390625, 1]
MGALE	A	10	1/1	9%/86%	247	63%	3.05	1/1	18%/73%	169	59%	1.89	[1/8, 1/8]
		100	1/1	7%/90%	511	70%	13.2	1/1	7%/90%	325	69%	6.28	[1/64, 1/64]
		1,000	1/1	6%/92%	775	72%	36.9	1/1	6%/91%	481	68%	15.5	[1/512, 1/512]
		10,000	1/1	7%/91%	1,127	71%	92.3	1/1	5%/92%	689	69%	37.8	[1/8192, 1/8192]
		100,000	0/1	0%/99%	1,391	70%	145	0/1	0%/98%	845	67%	61.5	[0, 1/65536]
		1,000,000	0/1	0%/99%	1,655	70%	185	0/1	0%/98%	1,001	67%	81.1	[0, 1/524288]
	B+	10	2/0	99%/0%	576	75%	13.1	1/0	93%/0%	169	68%	1.30	[1, 1]
		100	4/0	99%/0%	2,940	79%	463	1/0	97%/0%	343	76%	4.91	[1, 1]
		1,000	–/–	–/–	–	–	>600	1/0	99%/0%	517	79%	15.2	[1, 1]
		10,000	–/–	–/–	–	–	>600	1/0	99%/0%	749	80%	39.9	[1, 1]
		100,000	–/–	–/–	–	–	>600	1/0	99%/0%	923	81%	51.1	[1, 1]
		1,000,000	–/–	–/–	–	–	>600	1/0	99%/0%	1,097	80%	87.4	[1, 1]
AMP	A+	2	3/0	99%/0%	702	75%	24.7	3/0	94%/0%	176	63%	1.54	[1, 1]
		3	–/–	–/–	–	–	>600	4/0	99%/0%	420	68%	19.1	[1, 1]
		4	–/–	–/–	–	–	>600	6/0	99%/0%	920	61%	259	[1, 1]
	B+	2	4/0	99%/0%	645	74%	29.6	3/0	94%/0%	172	62%	1.51	[1, 1]
		3	3/0	99%/0%	2,517	74%	258	4/0	99%/0%	412	67%	20.2	[1, 1]
		4	–/–	–/–	–	–	>600	6/0	99%/0%	904	60%	279	[1, 1]
	C	2	1/1	9%/73%	176	46%	0.91	1/1	3%/92%	265	45%	2.77	[9/16, 9/16]
		3	1/1	6%/91%	367	56%	4.25	1/1	0%/98%	751	42%	25.7	[27/64, 27/64]
		4	1/1	1%/96%	738	59%	21.5	1/1	0%/99%	1,635	33%	195	[81/256, 81/256]

FIGURE 7.12: Main experimental results. We run each experiment both with the greedy exploration option enabled (GREEDY) and disabled (NORMAL). We display for each run the number of times we obtain a lower bound (LB ITERS) and an upper bound (UB ITERS) as well as the the percentage of the total time spent on obtaining lower bounds (LB TIME) and upper bounds (LB TIME), respectively. We show the maximum number of nodes in the control-flow tree constructed by the model checker over all model checks (UNW) and the percentage of the total time spent interpolating (INT). We show the total time (TIME) and indicate a timeout with a label ">600". Finally, we also show the final approximation of the property (RESULT) — this results coincide for GREEDY and NORMAL for each experiment.

and optimisations described in the previous section, the only optimisation we consider to be optional is that of greedy exploration. In Figure 7.12 we give detailed statistics for runs with this optimisation enabled and disabled, respectively. The statistics we show in each figure are explained in the captions of the figures.

Applicability It is promising that we are able to deal with two of the three main case studies (i.e. the network clients). We remark that the precision of the approximation given by our instrumentation loop is generally very good. In contrast to Chapter 5, our bounds are not numerical approximations. Moreover, for the most part, the lower bound

and upper bound of the interval coincide.

We briefly discuss Figure 7.11. Our first result is a negative one — with the approach described in this chapter we can only model check 7 of 31 of the properties we model checked with the method in Chapter 5 — these properties are marked with "✓". Five of these properties concern our main case studies, i.e. the network clients PING and NTP, as opposed to the much smaller pGCL or PRISM case studies. In fact, Figure 7.11 suggests the success of our approach is currently more determined by the nature of the probabilistic behaviours in programs than the complexity of the program in terms of, say, the amount of bit-level operations, arrays, pointers or functions that the program uses.

The remaining 24 of 30 properties cannot be verified with our current implementation. This is in a large part because 16 of these properties are not probabilistic safety properties and hence cannot be verified with the instrumentation-based method presented in this chapter. That is, properties marked with "LIVE" are probabilistic liveness properties and properties marked with "COST" are cost properties. A further 8 properties are probabilistic safety properties but cannot be verified with the current implementation. More specifically, properties marked with "LOOP" need to reason about infinite loops with probabilistic choices. Because the interpolating decision procedure we use does not produce quantified invariants, we cannot currently prove instrumentations of such programs are safe. Properties marked with "DIV" fail because of the amount of control-flow paths the model checker needs to consider. For example, programs such as TFTP and BRP are fault-tolerant. That is, in TFTP, we make five attempts to send a data packet before we give up. In practice, this means that there are many different resolutions of probabilistic choice in this program that either lead to success or failure. This translates to a large number of control-flow paths in instrumentations.

We now will discuss some specific points regarding the results in Figure 7.12.

Control-flow loops Recall that the parameters of, say, NTP and PING, essentially correspond to the number of iterations of the main control-flow loop that we need to take into account during model checking. For most programs we obtain precise results for all parameters, except for NTP B and MGALE A, where our termination criterion stops us

from obtaining more precise results. Akin to Chapter 5, we can analyse larger parameter values at no added cost as long as we are not expecting a more precise approximation. In contrast to Chapter 5, though, we are able to do this only when unfolding the *first* few loop iterations of the program enables us to establish sufficiently precise bounds on the property (as opposed to the *last* few loop iterations).

Transition ordering The transition ordering has a direct impact on the performance of our method. If there are many paths to the target location in the original program, then the transition order usually dictates whether we first explore the least probable path or the most probable paths of the original program in the instrumentations.

In NTP B, say, the most probable paths are considered first. Exploring the most probable path in this program is necessary and sufficient to exceed the initial bound ($\mathbf{p} = \frac{1}{2}$). Our model checker would never yield a counter-example that also explores less probable paths of the original program because, due to the transition ordering, any such counter-example is prefixed by the shorter counter-example that only explores the most probable path. In practice this means we need multiple iterations of the instrumentation loop to obtain a good lower bound.

In contrast, in MGALE B+, say, the transition ordering is such that the *least probable* paths are considered first. In practice, this means there is a counter-example that *first* explores all the unlikely paths to the target (without constituting a counter-example) before exploring the most probable path. With the greedy exploration option we find this counter-example and obtain a good lower bound with the first instrumentation. Without greedy exploration we may find counter-examples where we have skipped some of the paths and obtain a poorer lower bound.

Counter-examples and proofs Recall the instrumentation loop depicted in Figure 7.1. Without analysing counter-examples or proofs we halve the interval, $\mathbf{I}$, in each iteration of this loop. Hence we normally expect to perform 14 iterations of the instrumentation loop to ensure the difference between the upper and lower bound of $\mathbf{I}$ is below 10^{-4}. In Figure 7.12 we show that, through the analysis of counter-examples and proofs, the required number of iterations is usually much smaller than this.

For the lower bound we see that the number of iterations is dependent on the program and property under consideration. Informally, if there is only a single path to the target location in the original program, then the first counter-example in an instrumentation will always consist of this path, and hence yield the greatest possible lower bound (see PING A, D; MGALE A; AMP C). In contrast, if there are *many* paths to the target location then there may exist counter-examples that do not give optimal lower bounds, and multiple instrumentations may be necessary to establish a good lower bound (see NTP A, B; MGALE B+; AMP A+, B+). We never need more than one iteration to establish an upper bound. This is because of our adaptation of the interpolating decision procedure described in Section 7.4.2 — in general the invariants returned by interpolating decision procedures may not yield the tightest possible upper bound.

Finally, we mention that another advantage of analysing counter-examples and proofs is that we tend to obtain more meaningful approximations of the property at hand. That is, the lower and upper bound need not be multiples of 2^{-i} (where $i \in \mathbb{N}$ is the number of iterations of the instrumentation loop) and the lower bound is never strict. Moreover, in practice, the lower bound and upper bound often coincide.

Greedy exploration An indirect consequence of the greedy exploration heuristic is that we consider paths in the order dictated by the transition ordering. As a side-effect of this, if in the original program there is a single path to the target location which happens to be, say, the *last* path in this ordering, then enabling greedy exploration does not help (see PING A). Conversely, if there is a single path that is the *first* path in this ordering, then the heuristic can have a very positive effect (see MGALE A).

However, as described in Section 7.4.2, we considered the greedy exploration heuristic mostly because it enables the model checker to reuse invariants when *multiple* paths of the original program reach the target location. An indication of this reuse is the number of nodes in the control-flow trees constructed by the model checker (i.e. "UNW"). In, e.g., NTP B; PING B and NTP A, B we see a significant reduction in the number of nodes considered. We remark that the greedy exploration heuristic may reduce the number of nodes considered by the model checker exponentially (see, e.g, NTP B).

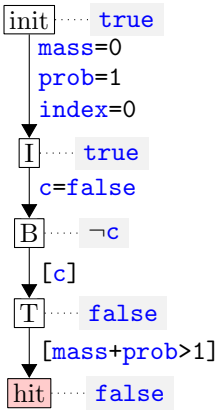


FIGURE 7.13: An infeasible path labelled with simple invariants of the instrumentation depicted in Figure 7.10.

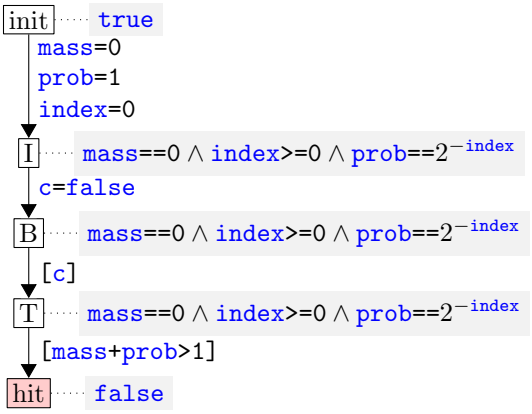


FIGURE 7.14: The same path of Figure 7.10 labelled with invariants that show a whole range of paths is infeasible.

Only for AMP C and PING D the number of nodes increase when we apply our heuristic — for these properties there is only a single path to the target location in the original program, and hence we cannot expect any reuse of invariants.

7.4.4 Template Invariants

Recall that, in Chapter 5, the size of the game abstraction of the program increases as we consider more iterations of the main program loop (unless considering just a few loop iterations of the program yields a satisfactory approximation). In Figure 7.12, we observe a similar trend in the size of the control-flow trees considered by the model checker. However, in contrast to Chapter 5, the scalability issues in this chapter can be improved within the existing framework. That is, the root cause of our problem is that the interpolating decision procedure we use is not normally able to find loop invariants for the loops that occur in instrumented program. As a consequence, the model checker unwinds the loop. This also explains why our current implementation is usually unable to deal with infinite loops in programs.

We illustrate our point with an example:

Example 7.19. *In Figure 7.13 and 7.14 we depict an infeasible control-flow path of the instrumentation in Figure 7.10. With our current interpolating decision procedure we*

typically obtain invariants like those depicted in Figure 7.13. These invariants show that the exit condition of the loop, `[c]`*, cannot possibly hold without executing the loop body at once. The invariants in Figure 7.13 are not loop invariants — after one loop iteration* `c` *may hold at location B. In practice this means that with these invariants the model checker will unwind the loop. If, instead, we had the invariants depicted in Figure 7.14, then we would not have to do this.*

Part of the problem is that the interpolating decision procedure we use is not able to deal with any kind of arithmetic — it cannot generate the predicates in Figure 7.14 because it cannot reason about exponentiation. Moreover, it cannot deal with any form of quantification, although, due to the infinite arrays in instrumented programs, we often require program invariants that quantify over array elements.

We have built a prototype extension of our basic instrumentation framework in which, in addition to using interpolation, we also produce invariants by matching control-flow paths against a number of *templates*. These templates detect whether a certain argument of infeasibility holds in the control-flow path under consideration. This argument is typically parametised in, say, the particular variables involved and the location of control-flow edges in the path that are relevant to the argument. If such a match succeeds then this template generates custom invariants to show the path is infeasible. Typically these invariants contain arithmetic or quantifiers. For example, the control-flow path in Example 7.19 matches with a template which generates the invariants in Figure 7.14.

In addition to this, we have also augmented the interpolating decision procedure [KW09] with *axioms* that can deal with the arithmetic and quantifiers we use in template invariants. For example, we have a rule that helps the interpolator deduce that $2^{-(\texttt{index})}$ equals $\frac{1}{2} \cdot 2^{-(\texttt{index}-1)}$). These axioms are necessary for the model checker to reuse the loop invariants (i.e. to apply FORCECOVER successfully, see [McM06]).

To evaluate our prototype extension we consider property D on a simplified version of PING, called PING-. Compared to PING, the program PING- has a simplified control-flow and some pointer dereferencing has been eliminated. Moreover, a probabilistic choice in the beginning of the program has been turned into a non-deterministic choice and the

			GREEDY					GREEDY + TEMPLATES					
			LB/UB ITERS	LB/UB TIME	UNW	INT	TIME	LB/UB ITERS	LB/UB TIME	UNW	INT	TIME	RESULT
PING-	D	1	1/1	38%/43%	118	65%	7.58	1/1	2%/95%	206	92%	54.3	[1/2, 1/2]
		2	1/1	53%/44%	263	71%	28.6	1/1	1%/97%	290	93%	127	[1/4, 1/4]
		3	1/1	48%/50%	526	66%	94.1	1/1	1%/98%	467	95%	456	[1/8, 1/8]
		4	1/1	49%/49%	1,030	61%	483	1/1	1%/98%	452	95%	513	[1/16, 1/16]
		6	–/–	–/–	–	–	>600	1/1	2%/96%	452	95%	528	[1/64, 1/64]
		10	–/–	–/–	–	–	>600	1/1	7%/92%	452	95%	552	$[2^{-10}, 2^{-10}]$
		12	–/–	–/–	–	–	>600	1/1	10%/89%	489	95%	565	$[2^{-12}, 2^{-12}]$
		14	–/–	–/–	–	–	>600	0/1	0%/99%	452	95%	522	$[0, 2^{-14}]$
		100	–/–	–/–	–	–	>600	0/1	0%/99%	452	95%	514	$[0, 2^{-100}]$
		1,000	–/–	–/–	–	–	>600	0/1	0%/99%	452	96%	537	$[0, 2^{-1000}]$
		10,000	–/–	–/–	–	–	>600	0/1	0%/99%	452	95%	525	$[0, 2^{-10000}]$
		100,000	–/–	–/–	–	–	>600	0/1	0%/99%	452	89%	572	$[0, 2^{-100000}]$

FIGURE 7.15: Experimental results for template invariants. We show the results of a normal run with the greedy heuristic turned on (GREEDY) and a run where we employ template invariants (GREEDY + TEMPLATES). For each run we show the same columns as in Figure 7.12. Due to space limitations we sometimes write, say, 2^{-10} instead of 1/1024 in the result column.

remaining probabilistic choices now fail with probability $\frac{1}{2}$ instead of with probability $\frac{92}{100}$. We note that these simplifications are required because of the prototypical nature of our extension and they do not affect the nature of the case study.

Results & analysis In Figure 7.15, we present the experimental results for our prototype extension. The main result is that the verification time does not depend on the number of loop iterations when using templates. That is, for larger parameter values, the time spend model checking PING- D for 100,000 loop iterations is approximately the same as for 100 loop iterations. This is not because we obtain a sufficiently good approximation with just a few loop iterations — it is because, with the right invariants, the size of the proof is independent of the number of loop iterations. The result is increasingly precise for large parameter values.

Unfortunately, we do pay a price for obtaining lower bounds for larger parameter values. To establish a lower bound we need to find a counter-example and, for this program, the length of counter-examples in the instrumentation depends on the number of loop iterations we consider. This is reflected in Figure 7.15 where, for parameters 4 up to 12, the increase in model checking time is due to the time required to obtain the lower bound. As the parameter value increases, the model checking cost actually drops slightly because the upper bound itself is sufficiently small to satisfy the termination criterion. The subsequent increase, for very large parameter values, is due to the cost of computing

rational arithmetic we perform when computing the upper bound.

7.4.5 Comparison to Abstraction Refinement

To conclude the experimental evaluation, we compare the approach in this chapter with the abstraction-refinement method in Chapter 5. Recall that, in terms of applicability, the abstraction-refinement approach is able to verify many more properties and programs than the instrumentation-based approach described in this chapter (see Figure 7.11).

To compare the methods in terms of performance we take the best runs of each method on those properties and programs that both methods can handle. For the abstraction-refinement method, we consider all predicate propagation methods and all refinable state selection method for each property and parameter. In addition to this, for each property, we select a configuration that performs best and run the *predicate initialisation* and *reachable state restriction* extensions with these configurations.[4] For the instrumentation method, we consider both runs with and without the greedy exploration heuristic. For PING- D we also consider runs with the *template invariant* extension.

We show the comparison of the two methods in Figure 7.16.

Precision We observe that one of the main differences between the methods are the resulting approximations of the properties at hand. That is, due to the model checker we use in the abstraction-refinement approach, the results returned by this approach are imprecise,[5] whereas the results from the instrumentation-based approach are not. This is particularly useful in PING- D, where the probabilities are so small that using a precise method for computing this probability is essential.

Performance With regards to the total verification time, we see that the different methods work well on different properties. For the main case studies, the abstraction-refinement method works better on PING A, D and NTP B, and, in terms of scalability and performance, the instrumentation-based approach outperforms the abstraction-

[4] This is COARSEST/TRACEADD for PING A, B, NTP B, MGALE B+, COARSEST/PRECADD for MGALE A, NEAREST/TRACEADD for AMP B+ and NEAREST/PRECADD for PING D, NTP A, AMP A+, C and PING- D.

[5] We remark that, for MGALE B+ and AMP B+, this can be improved by using precomputation algorithms.

			QPROVER (Chapter 5)								PROBITY (Chapter 7)				
			TIME	RESULT	TRACEADD	PRECADD	COARSEST	NEAREST	PRED. INIT.	REACH. RESTR.	TIME	RESULT	NORMAL	GREEDY	TEMPLATE INV.
PING	A	1	7.27	[0.0792, 0.0792]	✓	–	✓	–	✓	–	6.48	[99/1250, 99/1250]	✓	–	–
		2	14.1	[0.006336, 0.006336]	✓	–	✓	–	✓	–	52.5	[99/15625, 99/15625]	✓	–	–
		3	45.7	[0.00050688, 0.00050688]	✓	–	✓	–	✓	–	373	[198/390625, 198/390625]	–	✓	–
	B	1	0.81	[0, 0]	–	✓	–	✓	–	–	4.19	[0, 0]	✓	–	–
		2	–	–	–	–	–	–	–	–	87.7	[2277/15625, 2277/15625]	–	✓	–
		3	–	–	–	–	–	–	–	–	–	–	–	–	–
	D	1	6.75	[0.9108, 0.9108]	–	✓	✓	–	–	–	8.45	[2277/2500, 2277/2500]	✓	–	–
		2	14.2	[0.837936, 0.837936]	–	✓	–	✓	–	✓	46.5	[52371/62500, 52371/62500]	–	✓	–
		3	45.8	[0.770901, 0.770901]	–	✓	–	✓	–	✓	253	[204533/1562500, 204533/1562500]	✓	–	–
NTP	A	1	71.4	[0.08, 0.08]	–	✓	–	✓	–	–	11.3	[2/25, 2/25]	✓	–	–
		2	136	[0.1536, 0.1536]	–	✓	–	✓	–	✓	40.8	[96/625, 96/625]	–	✓	–
		4	217	[0.283607, 0.283607]	–	✓	–	✓	–	✓	175	[110784/390625, 110784/390625]	–	✓	–
		6	330	[0.393645, 0.393645]	–	✓	–	✓	–	✓	–	–	–	–	–
	B	1	7.37	[0.92, 0.92]	✓	–	✓	–	–	✓	5.01	[23/25, 23/25]	✓	–	–
		2	15.1	[0.9936, 0.9936]	✓	–	✓	–	–	✓	18.6	[621/625, 621/625]	–	✓	–
		4	18.4	[0.999959,1]	✓	–	✓	–	–	✓	36.2	[390609/390625, 1]	–	✓	–
		6	18.6	[0.999959,1]	✓	–	✓	–	–	✓	38.8	[390609/390625, 1]	–	✓	–
MGALE	A	100	1.79	[0.015625, 0.015625]	–	✓	✓	–	–	✓	6.28	[1/64, 1/64]	–	✓	–
		1,000	5.64	[0.00195312, 0.00195312]	–	✓	✓	–	–	✓	15.5	[1/512, 1/512]	–	✓	–
		10,000	26.0	[0.00012207, 0.00012207]	–	✓	✓	–	–	✓	37.8	[1/8192, 1/8192]	–	✓	–
		100,000	50.0	[0,6.10352e-05]	–	✓	✓	–	–	✓	61.5	[0, 1/65536]	–	✓	–
		1,000,000	202	[0,6.10352e-05]	–	✓	✓	–	–	✓	81.1	[0, 1/524288]	–	✓	–
	B+	100	0.07	[0.999998,1]	✓	–	–	✓	–	–	4.91	[1, 1]	–	✓	–
		1,000	0.07	[0.999998,1]	✓	–	✓	–	–	–	15.2	[1, 1]	–	✓	–
		10,000	0.07	[0.999998,1]	✓	–	✓	–	–	–	39.9	[1, 1]	–	✓	–
		100,000	0.07	[0.999998,1]	✓	–	✓	–	–	–	51.1	[1, 1]	–	✓	–
		1,000,000	0.07	[0.999998,1]	✓	–	✓	–	–	–	87.4	[1, 1]	–	✓	–
AMP	A+	4	0.20	[1, 1]	–	✓	✓	–	–	–	259	[1, 1]	–	✓	–
		40	7.55	[1, 1]	–	✓	–	✓	–	–	–	–	–	–	–
	B+	4	0.14	[0.999997,1]	✓	–	–	✓	–	–	279	[1, 1]	–	✓	–
		40	0.13	[0.999997,1]	✓	–	✓	–	–	–	–	–	–	–	–
	C	4	0.18	[0.316406, 0.316406]	✓	–	✓	–	–	–	195	[81/256, 81/256]	✓	–	–
		40	5.17	[0,7.53393e-05]	–	✓	–	✓	–	–	–	–	–	–	–
PING-	D	10	24.2	[0.000976562, 0.000976562]	–	✓	–	✓	–	–	552	$[2^{-10}, 2^{-10}]$	–	✓	✓
		20	110	[0, 2.00272e-05]	–	✓	–	✓	–	–	515	$[0, 2^{-20}]$	–	✓	✓
		40	–	–	–	–	–	–	–	–	512	$[0, 2^{-40}]$	–	✓	✓
		100,000	–	–	–	–	–	–	–	–	572	$[0, 2^{-100000}]$	–	✓	✓

FIGURE 7.16: A comparison between QPROVER and PROBITY. We show the total verification time in seconds (TIME) and the final approximation given by the tool (RESULT). We also show which run was responsible for the fastest time.

refinement approach for NTP A and PING- D. We also find it promising we were able to obtain a result for PING B (2) with our instrumentation-based approach, as we did not obtain this result for the abstraction-refinement method.

Observe that the abstraction-refinement approach works well on MGALE B+. This property computes the maximum probability of termination. This is 1 for this property because all paths of the program eventually terminate. Essentially, with the abstraction-refinement approach, we find a relatively small abstraction that shows that the program almost surely terminates — independently of the parameter. For our instrumentation-

based approach we cannot use this kind of reasoning and we must find counter-examples to establish lower bounds. Moreover, the counter-examples in the instrumented programs grow larger as the parameter increases because there are more paths to the terminating state in the original model.

In contrast, for MGALE A, the instrumentation-based approach outperforms the abstraction-refinement approach. For this property, both methods essentially unfold the loop iterations of the program. For the abstraction-refinement approach, as the parameter value increases, this gets quite expensive because of the cost of computing the abstraction. This is due to non-linear arithmetic in the program and the predicates it discovers. In the instrumentation-based approach the non-linear arithmetic is dealt with by propagating constants. This makes the instrumentation-based approach much more scalable in this instance.

We mention that part of the reason that NTP and MGALE work well with the instrumentation-based approach is that many resolutions of probabilistic choice lead directly to a target location or an end location. In practice, this keeps the number of control-flow paths we need to consider when model checking instrumented programs relatively small. This is not the case for AMP or PING, where we always make a certain number of probabilistic choices and only after these choices are made we decide whether the target location is reached. This latter behaviour is less suited to the current instrumentation scheme because the number of control-flow paths in instrumented programs grows very fast as we consider larger programs with more probabilistic choices.

7.5 Conclusions

In this chapter, we introduced a verification method for probabilistic software called *instrumentation*-based verification. We first discussed a theoretical framework through which we can compute probabilistic safety properties of probabilistic programs by verifying a number of non-probabilistic safety properties of instrumented, non-probabilistic programs. We developed this framework on the level of MDPs, making the theory applicable to any type of system with MDP semantics. We then demonstrated how this MDP-level

instrumentation process could be lifted to the level of programs. We demonstrated how to generate instrumented programs that verification tools can verify in practice.

The main attraction of our approach is that we can verify quantitative properties of probabilistic software by *directly* employing standard verification techniques for non-probabilistic software. That is, in principle we can apply anything from, say, classical Hoare logic [Flo67, Hoa69], abstract interpretation [CC77] or model checking [BCCY99, CGJ+00, McM06] to reason about quantitative properties of probabilistic programs.

We can also directly benefit from theoretical work on non-probabilistic verification problems. For example, in the non-probabilistic setting, there exist several abstraction frameworks that are known to be complete for both the verification and refutation of non-probabilistic safety properties [KP00, DN04, DN05]. We are not aware of any abstraction framework for MDPs that is complete.

We described an implementation of our approach which uses the interpolant-based model checking algorithm in [McM06] and the interpolating decision procedure from [KW09] to verify or refute probabilistic safety properties of instrumented programs, and we presented an extensive experimental evaluation for this implementation. We were able to verify properties of complex case studies such as NTP and PING as well as some smaller case studies. In terms of the state space and the language features they use, these complex case studies are significantly beyond the scope of existing probabilistic verification tools.

We now discuss loop invariants in more detail and follow this discussion with suggestions for future research directions.

Detecting loop invariants As previously discussed, our current model checker is often unable to find good loop invariants for instrumentations. Because of this, our current verification back-end often simply enumerates various control-flow paths of instrumented programs. These paths, in turn, correspond to *sets* of paths in the original uninstrumented program — we essentially enumerate potential probabilistic counter-examples and check the feasibility of these counter-examples with a decision procedure.

In this light, our verification procedure bears some resemblance to the probabilistic bounded model checking method in [FHT08] and the probabilistic counter-example

generation methods in, e.g., [HK07, AL09, WBB09]. However, in our approach, unlike in [FHT08, HK07, AL09, WBB09], we are able to deal with real, compilable programs instead of either low-level or abstract formal models.

More importantly, from a theoretical perspective, enumerating counter-examples is not necessary with our approach. Various non-probabilistic model checking methods are capable of preventing the enumeration of control-flow paths. In fact, we chose the interpolation-based model checking method of [McM06] for this very reason — the enumeration we experience in the experimental results is due to the inability of the underlying interpolating decision procedure to find good loop invariants. The discussion in the next section includes potential ways to improve the detection of good loop invariants.

Verification of instrumentations Our current implementation is able to verify fewer programs and properties than the abstraction-refinement approach presented in Chapter 5. In part, this is an implementation issue — instrumented programs contain a mix of infinite arrays, rational arithmetic and bit-level arithmetic and, as such, it is sometimes challenging to prove these programs are safe. We emphasise that we believe there is much scope to improve performance and applicability of our implementation by improving the process of verifying instrumented programs.

Our first suggestion is to make the verification step incremental. That is, instrumented programs for different bounds are extremely similar in nature and we think it is possible to exploit this similarity in successive iterations of the instrumentation loop.

Secondly, we believe it is possible to improve the performance and applicability of our current model checker by using interpolating decision procedures for theories that better suit our instrumented programs. Interpolating decision procedures for a wide range of theories are gaining an increasing amount of interest in the software verification community (see, e.g., [CGS08, BZM08, BKRW10]). If one such procedure is better able to find loop invariants of instrumented programs with, say, infinite loops, then this would improve the range of properties we can handle. For the same reason, we think there is also scope to further generalise and automate our "*template invariants*" extension. The template-based discovery of invariants for non-probabilistic software is an active area of

research (see, e.g., [CSS03, BHMR07, SG09, GSV09]). Compared to a general purpose model checker, a template-based approach is potentially better able to exploit the fact that we are verifying a particular class of programs, namely instrumented programs, instead of an arbitrary non-probabilistic program.

There is also significant scope to consider alternative verification techniques, such as bounded model checking or abstract interpretation, for the verification of instrumented programs. How well such techniques would be able to deal with our instrumented programs requires further investigation.

Instrumentation schemes A simple adaptation of our instrumentation method that we think may improve the performance of our of method is to enable backtracking from end locations (without updating the probability mass). Whether or not this improves performance requires further investigation — the additional backtracking would introduce more behaviours to the instrumented programs but, at the same time, we argue that the resulting instrumented programs will have a more regular structure and may potentially be cheaper to verify.

An issue with our instrumented programs is that they have the ability to "skip" certain behaviours of the original program. This induces a lot of non-determinism in instrumented programs (in the form of control-flow paths) and directly affects the scalability of our approach. It would be interesting to try instrumentation schemes where the ability to skip behaviours has been removed (for example by first removing behaviours in the original program that never reach the target state).

In this chapter, we limited our attention to probabilistic safety properties and, as such, we cannot verify the full range of properties we considered in Chapter 5. An interesting direction of research would be to extend our approach in such a way that we can compute probabilistic liveness properties by verifying a number of non-probabilistic liveness properties of instrumented programs. We remark that this is not possible with our current instrumentation scheme because of the ability of instrumentations to skip behaviours of the original model. We also believe our approach can be extended to deal with cost properties, but this requires further investigation.

Chapter 8

Conclusions

In this thesis, we set out to develop verification techniques for computing quantitative properties of real, compilable software that contains probabilistic behaviour. Our main motivation to do so is the success of various non-probabilistic software verification techniques that are able to verify certain classes of computer programs directly from source code in a scalable and fully automated way.

In Chapter 5, we introduced a probabilistic adaptation of non-probabilistic abstraction-refinement techniques, using game abstractions. With this approach, we were able to verify a wide range of properties on a range of case studies. Most promising is the verification of reliability properties of real network programs, PING, TFTP and NTP, which are approximately 1,000 lines of complex ANSI-C code each. These case studies cannot directly be verified with existing techniques. We also found that we do pay a price for dealing with *probabilistic* behaviour. For example, we were not able to adapt standard techniques to approximate transition functions of abstract models, or standard techniques for using interpolation-based refinements, due to probabilistic behaviours.

The main problem in approximating transition functions of game abstractions is the original formulation of game abstraction, which, for a given abstraction function, only considers one game abstraction. In Chapter 6, we addressed this problem by developing a much more fine-grained notion of abstraction for games through an abstraction preorder. Our preorder opens up the possibility of extending the applicability and scalability of the method in Chapter 5 by approximating game abstractions.

We took an entirely different approach in Chapter 7. The work in this chapter is motivated by our inability to fully exploit the state-of-the-art techniques that make software model checking so successful. We therefore reduce the problem of computing a probabilistic property of a probabilistic program to a series of non-probabilistic model checks that can *directly* be verified with existing state-of-the-art verification techniques and tools for non-probabilistic software. We again validated our approach by running it on a large number of properties and programs. Although we were not able to verify all the case studies we considered in Chapter 5, promisingly, we *were* able to verify PING and NTP competitively. We argue that there is significant scope to improve the results of our implementation further by considering alternative techniques and tools to perform non-probabilistic model checks verification.

Future Work There are various opportunities for further work. An obvious suggestion is to apply the abstraction preorder in Chapter 6 to the approach of Chapter 5. More specifically, we propose to adapt the abstraction and refinement procedures to compute the abstraction function of game abstractions incrementally, akin to, e.g., [DD01, BMR01, CKSY05, JM05, KS06]. Such an extension may help us deal more effectively with programs that are difficult to abstract with the current method and, in particular, may help improve the range of probabilistic choices we can handle.

Another important direction of research is the development of better refinement methods and heuristics for Chapter 5. Our refinement scheme yields relatively simple and local refinements using weakest preconditions — this is known not to work on all programs in the non-probabilistic setting [JM06]. In the non-probabilistic setting state-of-the-art model checkers use interpolating decision procedures for refinement [HJMM04, JM05]. We believe that the applicability and performance of our approach could be significantly improved by adapting interpolant-based refinement methods to a probabilistic setting. We remark that such an adaptation was employed in [HWZ08]. However, in [HWZ08], an abstract probabilistic counter-example is decomposed into a set of finite abstract paths, each of which is refined separately. This can lead to the introduction of many predicates along the refinement step. An interesting direction of research would be to find ways in which we can employ interpolating decision procedures directly on probabilistic

counter-examples.

To broaden the applicability of our instrumentation-based technique in Chapter 7, it would be interesting to adapt the instrumentation process to extend the range of properties that can be verified through instrumentations. We also think it would be interesting to consider alternative instrumentation schemes for the properties we already handle. Moreover, an important direction of research is to search for non-probabilistic verification techniques that are better able to verify instrumented programs in practice. We think it is possible that our approach could benefit from recent developments in interpolating decision procedures [CGS08, BZM08, BKRW10]. Moreover, we believe augmenting our current implementation with a more elaborate template-based invariant generation methods, such as [SG09, GSV09, BHMR07, CSS03], could significantly improve the applicability and scalability of our approach.

Unrelated to our verification methods, we argue that another important future direction of research is to consider different types of quantitative properties on probabilistic programs. For example, there are many programs that are designed to never terminate. For such programs, instead of computing expected total costs, it may be more meaningful to compute expected long run average costs (see, e.g., [Put94]). It is also often meaningful to consider conditional reachability probabilities. For example, on real programs the minimum probability of reaching any target location is usually 0. This is because the user may provide invalid arguments and the program terminates in its initialisation phase. Hence, in such cases, we may want to check the minimum probability of reaching the target location conditional on the fact that valid user arguments have been given.

Finally, we also think it would be interesting to consider computing parameterised values of quantitative properties. That is, instead of computing a property for many parameter values — as we did in our experiments — it is much more interesting to compute the value of the property as a function of the given parameters.

Our final remark on future work regards the fact that our current model of probabilistic programs is not able to deal with either recursive programs or concurrency. Also, our verification techniques are not tailored to deal with programs that use the heap. However, there is a lot of work on verifying recursive, concurrent or heap-intensive programs

in the non-probabilistic setting. Our final suggestion is to extend the verification methods described in this thesis in this direction.

Conclusion In this thesis, we introduced verification for quantitative properties of probabilistic software. We described two such techniques: one is a probabilistic adaptation of abstraction-refinement methods and the other is via a reduction to non-probabilistic verification problems. We provided extensive experimental results to demonstrate that, with our techniques, we can verify programs and properties that are far beyond the capabilities of existing probabilistic verification techniques. Our results demonstrate it is feasible to formally establish quantitative properties of probabilistic software in a fully automated way. Our work is, to the best of our knowledge, the first to verify probabilistic properties of real, compilable software in a fully automated way.

Bibliography

[AH90] J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, 1990.

[AHKV98] R. Alur, T. A. Henzinger, O. Kupferman, and M. Y. Vardi. Alternating refinement relations. In D. Sangiorgi and R. de Simone, editors, *Proceedings of the 9th International Conference on Concurrency Theory (CONCUR '98)*, volume 1466 of *LNCS*, pages 163–178. Springer, 1998.

[AL09] H. Aljazzar and S. Leue. Generation of counterexamples for model checking of Markov decision processes. In *Proceedings of the 6th International Conference on the Quantitative Evaluation of Systems (QEST '09)*, pages 197–206. IEEE Computer Society, 2009.

[AMP06] A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. In A. Valmari, editor, *Model Checking Software: Proceedings of the 13th International SPIN Workshop (SPIN '06)*, volume 3925 of *Lecture Notes in Computer Science*, pages 146–162. Springer, 2006.

[AQR+04] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: Exploiting program structure for model checking concurrent software. In P. Gardner and N. Yoshida, editors, *Proceedings of the 15th International Conference on Concurrency Theory (CONCUR '04)*, volume 3170 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2004.

[Bai96] C. Baier. Polynomial time algorithms for testing probabilistic bisimulation and simulation. In R. Alur and T. A. Henzinger, editors, *Proceedings of the 8th International Conference on Computer Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 50–61. Springer, 1996.

[BBKO10] T. Brázdil, V. Brozek, A. Kucera, and J. Obdrzálek. Qualitative reachability in stochastic BPA games. *Information and Computation*, 2010. (submitted).

[BBv+08] J. Barnat, L. Brim, I. Černá, M. Češka, and J. Tůmová. ProbDiVinE-MC: Multi-core LTL model checker for probabilistic systems. In *Quantitative Evaluation of Systems, 2008. QEST '08. Fifth International Conference on*, pages 77–78, 14-17 2008.

[BCCY99] A. Biere, A. Cimatti, E. Clarke, and Y. Yhu. Symbolic model checking without BDDs. In W. R. Cleaveland, editor, *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[BCDR04] T. Ball, B. Cook, S. Das, and S. K. Rajamani. Refining approximations in software predicate abstraction. In K. Jensen and A. Podelski, editors, *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 388–403. Springer, 2004.

[BCLR04] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. In E. A. Boiten, J. Derrick, and G. Smith, editors, *Proceedings of the 4th International Conference on Integrated Formal Methods (IFM '04)*, volume 2999 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2004.

[BdA95] A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In P. S. Thiagarajan, editor, *Proceedings of the 15th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTC '95)*, volume 1026 of *Lecture Notes in Computer Science*, pages 499–513. Springer, 1995.

[Bel77] N. D. Belnap. A useful four-valued logic. In J. M. Dunn and G. Epstein, editors, *Modern Uses of Multiple-valued Logics*, pages 8–37, 1977.

[BHMR07] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. In B. Cook and A. Podelski, editors, *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '07)*, volume 4349 of *Lecture Notes in Computer Science*, pages 378–394. Springer, 2007.

[Bil86] P. Billingsley. *Probability and Measure*. Wiley, New York, NY, 1986. Second Edition.

[BKRW10] A. Brillout, D. Kroening, P. Rümmer, and T. Wahl. An interpolating sequent calculus for quantifier-free Presburger arithmetic. In J. Giesl and R. Hähnle, editors, *Proceedings of the 5th International Joint Conference on Automated Reasoning (IJCAR '10)*, volume 6173 of *Lecture Notes in Computer Science*, pages 384–399. Springer, 2010.

[BMR01] T. Ball, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. *SIGPLAN Notices*, 36(5):203–213, 2001.

[BPR03] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. *Software Tools for Technology Transfer*, 5(1):49–58, 2003.

[BR01] T. Ball and S. K. Rajamani. The SLAM tookit. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV '01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264. Springer, 2001.

[Bry91] R. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40:205–213, 1991.

[BW10] D. Barsotti and N. Wolovick. Automatic probabilistic program verification through random variable abstraction. *Electronic Proceedings in Theoretical Computer Science*, 28, 2010.

[BZM08] D. Beyer, D. Zufferey, and R. Majumdar. CSIsat: Interpolation for LA+EUF. In A. Gupta and S. Malik, editors, *Proceedings of the 20th International Conference Computer Aided Verification (CAV '08)*, volume 5123 of *Lecture Notes in Computer Science*, pages 304–308. Springer, 2008.

[CAG05] S. Cheshire, B. Aboba, and E. Guttman. Dynamic configuration of IPv4 link-local addresses. http://www.ietf.org/rfc/rfc3927.txt, May 2005.

[CB06] F. Ciesinski and C. Baier. LiQuor: A tool for qualitative and quantitative linear time analysis of reactive systems. In *Proceedings of the 3rd International Conference on Quantitative Evaluation of Systems (QEST '06)*, pages 131–132. IEEE Computer Society, 2006.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*, pages 238–252. ACM, 1977.

[CCF+05] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyser. In M. Sagiv, editor, *Proceedings of the 14th European Symposium on Programming for the Construction and Analysis of Systems (ESOP '05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[CCF+07] R. Cavada, A. Cimatti, A. Franzén, K. Kalyanasundaram, M. Roveri, and R. K. Shyamasundar. Computing predicate abstractions by integrating bdds and SMT solvers. In *Proceedings of the 7th International Conference on Formal Methods in Computer-Aided Design (FMCAD '07)*, pages 69–76. IEEE Computer Society, 2007.

[CCG+04] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *Transactions on Software Engineering*, 30(6):338–402, 2004.

[CE81] E. Clarke and E. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In D. Kozen, editor, *Logic of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*. Springer, 1981.

[CGJ+00] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. Emerson and A. Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV '00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[CGL94] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[CGS08] A. Cimatti, A. Griggio, and R. Sebastiani. Efficient interpolant generation in satisfiability modulo theories. In C. R. Ramakrishnan and J. Rehof, editors, *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*, volume 4963 of *Lecture Notes in Computer Science*, pages 397–412. Springer, 2008.

[CKL04] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[CKRW10] B. Cook, D. Kroening, P. Rümmer, and C. M. Wintersteiger. Ranking function synthesis for bit-vector relations. In J. Esparza and R. Majumdar, editors, *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '10)*, volume 6015 of *Lecture Notes in Computer Science*, pages 236–250. Springer, 2010.

[CKSY04] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction

of ANSI-C programs using SAT. *Formal Methods in System Design*, 25:105–127, 2004.

[CKSY05] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In N. Halbwachs and L. D. Zuck, editors, *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '05)*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer, 2005.

[Con92] A. Condon. The complexity of stochastic games. *Information and Computation*, 96(2):203–224, 1992.

[Con93] A. Condon. On algorithms for simple stochastic games. *Series in Discrete Mathematics and Theoretical Computer Science*, 13:51–73, 1993.

[CPR05] B. Cook, A. Podelski, and A. Rybalchenko. Abstraction refinement for termination. In C. Hankin and I. Siveroni, editors, *Proceedings of the 12th International Symposium on Static Analysis (SAS '05)*, volume 3672 of *Lecture Notes in Computer Science*, pages 87–101. Springer, 2005.

[CPR06] B. Cook, A. Podelski, and A. Rybalchenko. Terminator: Beyond safety. In T. Ball and R. B. Jones, editors, *Proceedings of the 18th International Conference on Computer Aided Verification (CAV '06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 415–418. Springer, 2006.

[CR79] P. Cousot and C. R. Systematic design of program analysis frameworks. In *Proceedings of the 6th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '79)*, pages 269–282. ACM, 1979.

[CSS03] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In W. A. H. Jr. and F. Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification (CAV '03*, volume 2725 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 2003.

[CV10] R. Chadha and M. Viswanathan. A counterexample guided abstraction-refinement framework for Markov decision processes. *ACM Transactions on Computational Logic*, 12(1), 2010. To appear.

[CVWY90] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In E. Clarke and R. Kurshan, editors, *Proceedings of the 2nd International Workshop on Computer Aided Verification (CAV 190)*, volume 531 of *Lecture Notes in Computer Science*, pages 233–242. Springer, 1990.

[CW02] H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In V. Atluri, editor, *Proceedings of the 9th ACM Conference on Computer and Sommunications Security (ACM CCS '02)*, pages 235–244. ACM, 2002.

[CY90] C. Courcoubetis and M. Yannakakis. Markov decision processes and regular events. In M. Paterson, editor, *Proceedings of the 17th International Colloquium of Automata, Languages and Programming (ICALP '90)*, volume 443 of *Lecture Notes in Computer Science*, pages 336–349. Springer, 1990.

[CY98] C. Courcoubetis and M. Yannakakis. Markov decision processes and regular events. *IEEE Transactions on Automatic Control*, 43(10):1399–1418, 1998.

[dA99] L. de Alfaro. Computing minimum and maximum reachability times in probabilistic systems. In J. Baeten and S. Mauw, editors, *Proceedings of the 10th International Conference on Concurrency Theory (CONCUR '99)*, volume 1664 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 1999.

[dAGJ04] L. de Alfaro, P. Godefroid, and R. Jagadeesan. Three-valued abstractions of games: Uncertainty, but with precision. In *Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS '04)*, pages 170–179. IEEE CS, 2004.

[dAR07a] L. de Alfaro and P. Roy. Magnifying-lens abstraction for Markov decision processes. In W. Damm and H. Hermanns, editors, *Proceedings of the 19th*

International Conference on Computer Aided Verification (CAV '07), volume 4590 of *Lecture Notes in Computer Science*, pages 325–338. Springer, 2007.

[dAR07b] L. de Alfaro and P. Roy. Solving games via three-valued abstraction refinement. In L. Caires and V. T. Vasconcelos, editors, *Proceedings of the 18th International Conference on Concurrency Theory (CONCUR '07)*, volume 4703 of *Lecture Notes in Computer Science*, pages 74–89. Springer, 2007.

[DD01] S. Das and D. L. Dill. Successive approximation of abstract transition relations. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 51–60. IEEE Computer Society, 2001.

[DDP99] S. Das, D. L. Dill, and S. Park. Experience with predicate abastraction. In N. Halbwachs and D. Peled, editors, *Proceedings of the 11th International Conference on Computer-Aided Verification (CAV '99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 160–171. Springer, 1999.

[Des99] J. Desharnais. *Labelled Markov Processes*. PhD thesis, McGill University, 1999.

[DGG97] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):252–291, 1997.

[dHdV02] J. den Hartog and E. P. de Vink. Verifying probabilistic programs using a Hoare like logic. *International Journal of Foundations of Computer Science*, 13(3):315–340, 2002.

[Dij75] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[DJJL01] P. R. D'Argenio, B. Jeannet, H. E. Jensen, and K. G. Larsen. Reachability analysis of probabilistic systems by successive refinements. In L. d. Alfaro and S. Gilmore, editors, *Proceedings of the Joint International Workshop on*

Process Algebra and Probabilistic Methods. Performance Modeling and Verification (PAPM-PROBMIV '01), volume 2165 of *Lecture Notes in Computer Science*, pages 39–56. Springer, 2001.

[DJJL02] P. R. D'Argenio, B. Jeannet, H. E. Jensen, and K. G. Larsen. Reduction and refinement strategies for probabilistic systems. In H. Hermanns and R. Segala, editors, *Proceedings of the 2nd Joint International Workshop on Process Algebra and Probabilistic Methods. Performance Modeling and Verification (PAPM-PROBMIV '02)*, volume 2399 of *Lecture Notes in Computer Science*, pages 57–76. Springer, 2002.

[DKNP06] M. Duflot, M. Kwiatkowska, G. Norman, and D. Parker. A formal analysis of Bluetooth device discovery. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(6):621–632, 2006.

[DLT08] J. Desharnais, F. Laviolette, and M. Tracol. Approximate analysis of probabilistic processes: Logic, simulation and games. In *Proceedings of the 5th International Conference on the Quantitative Evaluaiton of Systems (QEST '08)*, pages 264–273. IEEE Computer Society, 2008.

[DN04] D. Dams and K. S. Namjoshi. The existence of finite abstractions for branching time model checking. In *Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS '04)*, pages 335–344. IEEE CS, 2004.

[DN05] D. Dams and K. S. Namjoshi. Automata as abstractions. In R. Cousot, editor, *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI '05)*, volume 3385 of *Lecture Notes in Computer Science*, pages 216–232. Springer, 2005.

[DNV95] R. De Nicola and F. Vaandrager. Three logics for branching bisimulation. *Journal of the ACM*, 42(2):458–487, 1995.

[DP02] B. Davey and H. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, second edition, 2002.

[DPW00] A. Di Pierro and H. Wiklicky. Concurrent constraint programming: Towards probabilistic abstract interpretation. In F. Pfenning, editor, *Proceedings of the 2nd International ACM/SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP '00)*, pages 127 – 138. ACM, 2000.

[DPW01] A. Di Pierro and H. Wiklicky. Measuring the precision of abstract interpretation. In K.-K. Lau, editor, *Proceedings of the 10th International Workshop on Logic Based Program Synthesis and Transformation (LOPSTR '00)*, volume 2042 of *Lecture Notes in Computer Science*, pages 147–164. Springer, 2001.

[FHT08] M. Fränzle, H. Hermanns, and T. Teige. Stochastic satisfiability modulo theory: A novel technique for the analysis of probabilistic hybrid systems. In M. Egerstedt and B. Mishra, editors, *Proceedings of 11th International Workshop on Hybrid Systems: Computation and Control (HSCC '08)*, volume 4981 of *Lecture Notes in Computer Science*, pages 172–186. Springer, 2008.

[Flo67] R. Floyd. Assigning meaning to programs. In J. Schartz, editor, *Proceedings of Symposium on Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967.

[FLW06] H. Fecher, M. Leucker, and V. Wolf. Don't know in probabilistic systems. In A. Valmari, editor, *Proceedings of the 13th International SPIN Workshop (SPIN '06)*, volume 3925 of *Lecture Notes in Computer Science*, pages 71–88. Springer, 2006.

[GC06] A. Gurfinkel and M. Chechik. Why waste a perfectly good abstraction? In H. Hermanns and J. Palsberg, editors, *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '06)*, volume 3920 of *Lecture Notes in Computer Science*, pages 212–226. Springer, 2006.

[God05] P. Godefroid. Software model checking: The Verisoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[GSV09] S. Gulwani, S. Srivastava, and R. Venkatesan. Constraint-based invariant inference over predicate abstraction. In N. D. Jones and M. Müller-Olm, editors, *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '09)*, volume 5403 of *Lecture Notes in Computer Science*, pages 120–135. Springer, 2009.

[HD01] J. Hatcliff and M. B. Dwyer. Using the bandera tool set to model-check properties of concurrent java software. In K. G. Larsen and M. Nielsen, editors, *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR '01)*, volume 2154 of *Lecture Notes in Computer Science*, pages 39–58. Springer, 2001.

[Her90] T. Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67, 1990.

[HHWZ10] E. M. Hahn, H. Hermanns, B. Wachter, and L. Zhang. PASS: Abstraction refinement for infinite probabilistic models. In J. Esparza and R. Majumdar, editors, *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '10)*, volume 6015 of *Lecture Notes in Computer Science*, pages 353–357. Springer, 2010.

[HJM03] T. A. Henzinger, R. Jhala, and R. Majumdar. Counterexample-guided control. In J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, editors, *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP '03)*, volume 2719 of *Lecture Notes in Computer Science*, pages 886–902. Springer, 2003.

[HJMM04] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstraction from proofs. In N. D. Jones and X. Leroy, editors, *Proceedings of the 31st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '04)*. ACM, 2004.

[HJMS03] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In T. Ball and S. K. Rajamani, editors, *Model Checking Software: Proceedings of the 10th International SPIN Workshop (SPIN '03)*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer, 2003.

[HK07] T. Han and J.-P. Katoen. Counterexamples in probabilistic model checking. In O. Grumberg and M. Huth, editors, *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '07)*, volume 4424 of *Lecture Notes in Computer Science*, pages 72–86. Springer, 2007.

[HKN+08] J. Heath, M. Z. Kwiatkowska, G. Norman, D. Parker, and O. Tymchyshyn. Probabilistic model checking of complex biological pathways. *Theoretical Computer Science*, 391(3):239–257, 2008.

[HKNP06] A. Hinton, M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In H. Hermanns and P. Jens, editors, *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '06)*, volume 3920 of *Lecture Notes in Computer Science*, pages 441–444. Springer, 2006.

[Hoa69] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hol03] G. Holzmann. *SPIN model checker, the: primer and reference manual*. Addison-Wesley Professional, 2003.

[HS00] G. J. Holzmann and M. H. Smith. Automatic software feature verification. *Bell Labs Technical Journal*, 5(2):72–87, 2000.

[Hut05] M. Huth. On finite-state approximants for probabilistic computation tree logic. *Theoretical Computer Science*, 346(1):113–134, 2005.

[HWZ08] H. Hermanns, B. Wachter, and L. Zhang. Probabilistic CEGAR. In A. Gupta and S. Malik, editors, *Proceedings of the 20th International Conference on Computer Aided Verification (CAV '08)*, volume 5123 of *Lecture Notes in Computer Science*, pages 162–175. Springer, 2008.

[JDL02] B. Jeannet, P. R. D'Argenio, and K. G. Larsen. RAPTURE: A tool for verifying Markov decision processes. In L. Brim, P. Jančar, M. Křetínský, and A. Kučera, editors, *Proceedings of the 13th International Conference on Concurrency Theory (CONCUR '02)*, 2002.

[JL91] B. Jonsson and K. G. Larsen. Specification and refinement of probabilistic processes. In *Proceedings of Sixth Annual IEEE Symposium on Logic in Computer Science (LICS '91)*, pages 266–277. IEEE, 1991.

[JM05] R. Jhala and K. L. McMillan. Interpolant-based transition relation approximation. In K. Etessami and S. Rajamani, editors, *Proceedings of the 17th International Conference on Computer Aided Verification (CAV '05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 39–51. Springer, 2005.

[JM06] R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In H. Hermanns and J. Palsberg, editors, *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '06)*, volume 3920 of *Lecture Notes in Computer Science*, pages 459–473. Springer, 2006.

[KH09] M. Kattenbelt and M. Huth. Verification and refutation of probabilistic specifications via games. In R. Kannan and K. N. Kumar, editors, *Proceedings*

of the 29th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2009), volume 4 of *LIPIcs*, pages 251–262. Schloss Dagstuhl, 2009.

[KKLW07] J.-P. Katoen, D. Klink, M. Leucker, and V. Wolf. Three-valued abstraction for continuous-time Markov chains. In W. Damm and H. Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 311–324. Springer, 2007.

[KKNP08] M. Kattenbelt, M. Kwiatkowska, G. Norman, and D. Parker. Game-based probabilistic predicate abstraction in PRISM. In A. Aldini and C. Baier, editors, *Proceedings of the 6th Workshop on Quantitative Aspects of Programming Languages (QAPL '08)*, volume 220 of *Electronic Notes in Theoretical Computer Science*, pages 5–21. Elsevier, 2008.

[KKNP09] M. Kattenbelt, M. Kwiatkowska, G. Norman, and D. Parker. Abstraction refinement for probabilistic software. In N. D. Jones and M. Müller-Olm, editors, *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '09)*, volume 5403 of *Lecture Notes in Computer Science*, pages 182–197. Springer, 2009.

[KKNP10] M. Kattenbelt, M. Kwiatkowska, G. Norman, and D. Parker. A game-based abstraction-refinement framework for Markov decision processes. *Formal Methods in System Design*, 36(3):246–280, 2010.

[KMMM10] J.-P. Katoen, A. McIver, L. Meinicke, and C. C. Morgan. Linear-invariant generation for probabilistic programs: - automated support for proof-based methods. In R. Cousot and M. Martel, editors, *Proceedings of the 17th International Symposium on Static Analysis (SAS '10)*, volume 6337 of *Lecture Notes in Computer Science*, pages 390–406. Springer, 2010.

[KN02] M. Z. Kwiatkowska and G. Norman. Verifying randomized byzantine agreement. In D. Peled and M. Y. Vardi, editors, *Proceedings of 22nd IFIP WG*

6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE '02), volume 2529 of *Lecture Notes in Computer Science*, pages 194–209. Springer, 2002.

[KNP06] M. Kwiatkowska, G. Norman, and D. Parker. Game-based abstraction for Markov decision processes. In *Proceedings of the 3rd International Conference on Quantitative Evaluation of Systems (QEST '06)*, pages 157–166. IEEE Computer Society, 2006.

[KNP09] M. Z. Kwiatkowska, G. Norman, and D. Parker. Stochastic games for verification of probabilistic timed automata. In J. Ouaknine and F. W. Vaandrager, editors, *Proceedings of the 7th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS '09)*, volume 5813 of *Lecture Notes in Computer Science*, pages 212–227. Springer, 2009.

[KNP10] M. Kwiatkowska, G. Norman, and D. Parker. A framework for verification of software with time and probabilities. In K. Chatterjee and T. Henzinger, editors, *Proceedings of the 8th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS '10)*, volume 6246 of *Lecture Notes in Computer Science*, pages 25–45. Springer, 2010.

[KNS01] M. Z. Kwiatkowska, G. Norman, and R. Segala. Automated verification of a randomized distributed consensus protocol using cadence smv and prism. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th International Conference Computer Aided Verification (CAV '01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 194–206. Springer, 2001.

[KNS03] M. Kwiatkowska, G. Norman, and J. Sproston. Probabilistic model checking of deadline properties in the IEEE 1394 FireWire root contention protocol. *Formal Aspects of Computing*, 14(3):295–318, 2003.

[KP00] Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Information and Computation*, 163(1):203–243, 2000.

[Kro10a] D. Kroening. Personal communication, 2010.

[Kro10b] D. Kroening. GOTO-CC: a C/C++ front-end for cerification, 2010. `http://www.cprover.org/goto-cc/`.

[KS06] D. Kroening and N. Sharygina. Approximating predicate images for bit-vector logic. In H. Hermanns and P. Jens, editors, *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '06)*, volume 3920 of *Lecture Notes in Computer Science*, pages 441–444. Springer, 2006.

[KS08] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 2008.

[KSK76] J. G. Kemeny, J. L. Snell, and A. W. Knapp. *Denumerable Markov Chains*. Springer-Verlag, 2 edition, 1976.

[Kur94] R. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, 1994.

[KW09] D. Kroening and G. Weissenbacher. An interpolating decision procedure for transitive relations with uninterpreted functions. In K. Namjoshi and A. Zeller, editors, *Proceedings of the 5th Haifa Verification Conference (HVC '10)*, 2009. To appear.

[KZH+09] J.-P. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen. The ins and outs of the probabilistic model checker mrmc. In *Proceedings of the 6th International Conference on Quantitative Evaluation of Systems (QEST '09)*, pages 167–176. IEEE Computer Society, 2009.

[LBC03] S. Lahiri, R. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In W. Hunt and F. Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification (CAV '03)*, volume 2725 of *Lecture Notes in Computer Science*, pages 141–153. Springer, 2003.

[LBC05] S. Lahiri, T. Ball, and B. Cook. Predicate abstraction via symbolic decision procedures. In K. Etessami and S. Rajamani, editors, *Proceedings of the 17th International Conference on Computer Aided Verification (CAV '05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 24–38. Springer, 2005.

[LL02] T. Lethbridge and R. Laganiere. *Object-Oriented Software Engineering: Practical Software Development using UML and Java*. McGraw-Hill, Inc., New York, NY, USA, 2002.

[LMOW08] A. Legay, A. S. Murawski, J. Ouaknine, and J. Worrell. On automated verification of probabilistic programs. In C. Ramakrishnan and J. Rehof, editors, *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*, volume 4963 of *Lecture Notes in Computer Science*, pages 173–187. Springer, 2008.

[LNO06] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras. SMT techniques for fast predicate abstraction. In T. Ball and R. B. Jones, editors, *Proceedings of the 18th International Conference on Computer Aided Verification (CAV '06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 424–437. Springer, 2006.

[LS91] K. G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94(1):1–28, 1991.

[LT88] K. G. Larsen and B. Thomsen. A modal process logic. In *Proceedings of the 3rd Annual Symposium on Logic in Computer Science (LICS '88)*, pages 203–210. IEEE Computer Society, 1988.

[LTMP09] S. La Torre, P. Madhusudan, and G. Parlato. Reducing context-bounded concurrent reachability to sequential reachability. In A. Bouajjani and O. Maler, editors, *Proceedings of the 21st International Conference on Computer Aided Verification (CAV'09)*, volume 5643 of *Lecture Notes in Computer Science*, pages 477–492. Springer, 2009.

[LX90] K. G. Larsen and L. Xinxin. Equation solving using modal transition systems. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science (LICS '90)*, pages 108–117. IEEE Computer Society, 1990.

[McM92] K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, 1992.

[McM06] K. L. McMillan. Lazy abstraction with interpolants. In T. Ball and R. B. Jones, editors, *Proceedings of the 18th International Conference on Computer Aided Verification (CAV '06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.

[Mil99] R. Milner. *Communicating and mobile systems: The π-calculus*. Cambridge University Press, 1999.

[MM05] A. McIver and C. Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Monographs in Computer Science. Springer, 2005.

[Mon00] D. Monniaux. Abstract interpretation of probabilistic semantics. In J. Palsberg, editor, *Proceedings of the 7th International Static Analysis Symposium (SAS '00)*, volume 1824 of *Lecture Notes in Computer Science*, pages 322–339. Springer, 2000. Extended version on the author's web site.

[Mon01] D. Monniaux. *Analyse de programmes probabilistes par interprétation abstraite*. PhD thesis, Université Paris IX Dauphine, 2001.

[MPC+02] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. *ACM SIGOPS Operating Systems Review*, 36(SI):75–88, 2002.

[MTLT10] S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. Automatic numeric abstractions for heap-manipulating programs. In M. V. Hermenegildo and J. Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*, pages 211–222. ACM, 2010.

[Nam03] K. Namjoshi. Abstraction for branching time properties. In W. J. Hunt and F. Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification (CAV '03)*, volume 2725 of *Lecture Notes in Computer Science*, pages 288–300. Springer, 2003.

[NM10] U. Ndukwu and A. McIver. An expectation transformer approach to predicate abstraction and data independence for probabilistic programs. *Electronic Proceedings in Theoretical Computer Science*, 28, 2010.

[NNH05] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2005.

[Par02] D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, 2002.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on the Foundations of Computer Science (FOCS '77)*, pages 46–57. Springer, 1977.

[Put94] M. L. Puterman. *Markov Decision Processes*. John Wiley & Sons, 1994.

[QS82] J. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Proceedings of the 5th International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982.

[QW04] S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*, pages 14–24. ACM, 2004.

[Ric53] H. G. Rice. Classes of recursively enumerable sets and thei decision problems. *Transactions of the American Mathematical Society*, 72:358–366, 1953.

[SB04] V. Schuppan and A. Biere. Efficient reduction of finite state model checking to reachability analysis. *International Journal on Software Tools for Technology Transfer*, 5(2–3):185–204, 2004.

[Sch93] R. Schneider. *Convex Bodies: The Brunn–Minkowski Theory*. Cambridge University Press, Cambridge, 1993.

[Seg95] R. Segala. *Modelling and Verification of Randomized Distributed Protocols*. PhD thesis, Massachusetts Institute of Technology, 1995.

[Seg06] R. Segala. Probability and nondeterminism in operational models of concurrency. In C. Baier and H. Hermanns, editors, *Proceedings of the 17th International Conference on Concurrency Theory (CONCUR '06)*, volume 4137 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2006.

[SG06] S. Shoham and O. Grumberg. 3-valued abstraction: More precision at less cost. In *Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science (LICS '06)*, pages 399–410. IEEE Computer Society, 2006.

[SG07] S. Shoham and O. Grumberg. A game-based framework for CTL counterexamples and 3-valued abstraction-refinement. *ACM Transactions on Computational Logic*, 9(1), 2007.

[SG09] S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In M. Hind and A. Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*, pages 223–234. ACM, 2009.

[Sha53] L. Shapley. Stochastic games. *Proceedings of the National Academy of Sciences of the United States of America*, 39:1095–1100, 1953.

[Shm02] V. Shmatikov. Probabilistic analysis of anonymity. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW '02)*, pages 119–128. IEEE Computer Society, 2002.

[SL94] R. Segala and N. A. Lynch. Probabilistic simulations for probabilistic processes. In B. Jonsson and J. Parrow, editors, *Proceedings of the 5th International Conference on Concurrency Theory (CONCUR '94)*, Lecture Notes in Computer Science, pages 481–496. Springer, 1994.

[Smi08] M. J. A. Smith. Probabilistic abstract interpretation of imperative programs using truncated normal distributions. *Electronic Notes in Theoretical Computer Science*, 220(3):43–59, 2008.

[SS99] H. Saïdi and N. Shankar. Abstract and model check while you prove. In N. Halbwachs and D. Peled, editors, *Proceedings of the 11th International Conference on Computer-Aided Verification (CAV '99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 443–454. Springer, 1999.

[Ste06] G. Steel. Formal analysis of pin block attacks. *Theoretical Computer Science*, 367(1-2):257–270, 2006.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[Var85] M. Y. Vardi. Verification of probabilistic concurrent finite-state programs. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science (FOCS '85)*, pages 327–338. IEEE, 1985.

[VHG+02] W. Visser, K. Havelund, B. Guillaume, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2002.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the 1st IEEE Symposium on Logic in Computer Science (LICS '86)*, pages 322–331. IEEE Computer Society, 1986.

[WBB09] R. Wimmer, B. Braitling, and B. Becker. Counterexample generation for discrete-time markov chains using bounded model checking. In N. D. Jones and M. Müller-Olm, editors, *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '09)*, volume 5403 of *Lecture Notes in Computer Science*, pages 366–380. Springer, 2009.

[Wol78] P. Wolper. Expressing interesting properties of programs in propositional temporal togic. In *Proceedings of the 13th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '86)*, pages 184–193. ACM, 1978.

[WZ10] B. Wachter and L. Zhang. Best probabilistic transformers. In G. Barthe and M. V. Hermenegildo, editors, *Proceedings of the 11th International Conference on Verification, Model Checking, and Astract Interpretation (VMCAI '10)*, volume 5944 of *Lecture Notes in Computer Science*, pages 362–379. Springer, 2010.

[WZH07] B. Wachter, L. Zhang, and H. Hermanns. Probabilistic model checking modular theories. In *Proceedings of the 4rd International Conference on Quantitative Evaluation of Systems (QEST '07)*, 2007. To Appear.

[Zha09] L. Zhang. *Decision Algorithms for Probabilistic Simulations*. PhD thesis, Saarland University, 2009.

[ZvB10] X. Zhang and F. van Breugel. Model checking randomized algorithms with java pathfinder. In *Proceedings of the 7th International Conference on Quantitative Evaluation of Systems (QEST '10)*, 2010. To appear.

Appendix A

Proofs

A.1 Proofs of Chapter 3

A.1.1 Proof of Lemma 3.2

In this section we will prove Lemma 3.2 on page 26 holds.

Lemma 3.2. *Let S_1, S_2 and S_3 be sets and let $R, R' \subseteq S_1 \times S_2$ and $R'' \subseteq S_2 \times S_3$ be relations over these sets. The following statements hold:*

(i) *If R is left or right-total then so is $\mathcal{L}(R)$.*

(ii) *If R is left or right-unique then so is $\mathcal{L}(R)$.*

(iii) *$\mathcal{L}(R^{-1})$ equals $\mathcal{L}(R)^{-1}$.*

(iv) *$R \subseteq R'$ implies $\mathcal{L}(R) \subseteq \mathcal{L}(R')$.*

(v) *$\mathcal{L}(R'') \circ \mathcal{L}(R)$ is contained in $\mathcal{L}(R'' \circ R)$.*

(vi) *Suppose I is a countable index set and $\{\lambda_i^1\}_{i \in I}$ and $\{\lambda_i^2\}_{i \in I}$ are families of distributions in $\mathbb{D}S_1$ and $\mathbb{D}S_2$, respectively, with $\langle \lambda_i^1, \lambda_i^2 \rangle \in \mathcal{L}(R)$ for each $i \in I$, then for every family of weights $\{w_i\}_{i \in I}$, we have that $\langle \sum_{i \in I} w_i \cdot \lambda_i^1, \sum_{i \in I} w_i \cdot \lambda_i^2 \rangle \in \mathcal{L}(R)$, also.*

Proof. We prove each claim separately.

(i) We first show that if R is right-total, then so is $\mathcal{L}(R)$. As R is right-total then for

every $s_2 \in S_2$ there is an element of S_1, say $\overline{s_2}$, such that $\langle \overline{s_2}, s_2 \rangle \in R$. To show $\mathcal{L}(R)$ is right-total we need to show that for every $\lambda_2 \in \mathbb{D}S_2$ there is some $\overline{\lambda_2} \in \mathbb{D}S_1$ such that $\langle \overline{\lambda_2}, \lambda_2 \rangle \in \mathcal{L}(R)$. It is easy to verify this holds for $\overline{\lambda_2}$ if, for every $s_1 \in S_1$, $\overline{\lambda_2}(s_1)$ is $\sum_{s_1 = \overline{s_2}} \lambda_2(s_2)$. The weight function δ witnessing this is defined as $\delta(s_1, s_2)$ being $\lambda_2(s_2)$ if $\overline{s_2} = s_2$ and 0 if $\overline{s_2} \neq s_1$ for every $\langle s_1, s_2 \rangle \in S_1 \times S_2$.

The case for left-totalness is symmetric.

(ii) We first show that if R is right-unique, then so is $\mathcal{L}(R)$. Assume R is right-unique. To show $\mathcal{L}(R)$ is right-unique we need to show that for every $\lambda_1 \in \mathbb{D}S_1$ there at most one $\lambda_2 \in \mathbb{D}S_2$ such that $\langle \lambda_1, \lambda_2 \rangle \in \mathcal{L}(R)$. Hence, let us take an arbitrary $\lambda_1 \in \mathbb{D}S_1$ and an arbitrary $\lambda_2 \in \mathbb{D}S_2$ such that $\langle \lambda_1, \lambda_2 \rangle \in \mathcal{L}(R)$. Suppose $\langle \lambda_1, \lambda_2 \rangle \in \mathcal{L}(R)$ is witnessed by a weight function, $\delta \in S_1 \times S_2 \to [0, 1]$. If R is right-unique, then, for all $\langle s_1, s_2 \rangle \in S_1 \times S_2$ we must have that $\delta(\langle s_1, s_2 \rangle)$ is $\lambda_1(s_1)$ if $\langle s_1, s_2 \rangle \in R$ and 0 if $\langle s_1, s_2 \rangle \notin R$, in order to satisfy (3.1) for s_1. As a result, without having made *any* assumption on λ_2 we have for all $s_2 \in S_2$ that

$$\begin{aligned}
\lambda_2(s_2) &= \sum_{s_1 \in S_1} \delta(\langle s_1, s_2 \rangle) && \text{(Definition } \delta\text{)} \\
&= \sum_{s_1 \in R^{-1}(s_2)} \delta(\langle s_1, s_2 \rangle) && \text{(Definition } \delta\text{)} \\
&= \sum_{s_1 \in R^{-1}(s_2)} \lambda_1(s_1)\ . && (R \text{ is right-unique})
\end{aligned}$$

That is, λ_2 is uniquely defined by λ_1 and R, meaning $\mathcal{L}(R)$ is right-unique.

The case for left-uniqueness is symmetric.

(iii) Follows from the symmetry in Definition 3.1 and the definition of relational inverse.

(iv) Suppose $\langle \lambda_1, \lambda_2 \rangle \in \mathcal{L}(R)$ is witnessed by a weight function δ. If $R \subseteq R'$ then we can use δ to witness $\langle \lambda_1, \lambda_2 \rangle \in \mathcal{L}(R')$. The only condition affected by the switch from R to R' is (3.3), which becomes weaker as $\langle s_1, s_2 \rangle \notin R'$ implies $\langle s_1, s_2 \rangle \notin R$.

(v) Suppose that $\langle \lambda_1, \lambda_3 \rangle \in (\mathcal{L}(R'') \circ \mathcal{L}(R))$. By definition there must be a distribution $\lambda_2 \in \mathbb{D}S_2$ s.t. $\langle \lambda_1, \lambda_2 \rangle \in \mathcal{L}(R)$ and $\langle \lambda_2, \lambda_3 \rangle \in \mathcal{L}(R'')$ as witnessed by weight functions δ and δ'', respectively. Remaining to show is that $\langle \lambda_1, \lambda_3 \rangle \in \mathcal{L}(R'' \circ R)$. We claim

this is witnessed by the weight function $\overline{\delta}$, defined for every $\langle s_1, s_3\rangle \in S_1 \times S_3$ as

$$\overline{\delta}(\langle s_1, s_3\rangle) = \sum_{s_2 \in \text{SUPP}(\lambda_2)} \frac{\delta(\langle s_1, s_2\rangle) \cdot \delta''(\langle s_2, s_3\rangle)}{\lambda_2(s_2)} \ .$$

To show this we need to ensure conditions (3.1) up to (3.3) hold for λ_1, λ_3 and $\overline{\delta}$. Condition (3.3) holds because if $\langle s_1, s_3\rangle \notin (R'' \circ R)$ then there is no $s_2 \in S$ for which both $\langle s_1, s_2\rangle \in R$ and $\langle s_2, s_3\rangle \in R''$ and hence the numerators in $\overline{\delta}(\langle s_1, s_3\rangle)$'s sum are 0 for every $s_2 \in S_2$. We now show (3.1) holds for arbitrary $s_1 \in S_1$:

$$\begin{aligned}
\sum_{s_3 \in S_3} \overline{\delta}(\langle s_1, s_3\rangle) &= \sum_{s_3 \in S_3} \sum_{s_2 \in \text{SUPP}(\lambda_2)} \frac{\delta''(\langle s_2, s_3\rangle)}{\lambda_2(s_2)} \cdot \delta(\langle s_1, s_2\rangle) && \text{(Definition } \overline{\delta}\text{)} \\
&= \sum_{s_2 \in \text{SUPP}(\lambda_2)} \left(\sum_{s_3 \in S_3} \frac{\delta''(\langle s_2, s_3\rangle)}{\lambda_2(s_2)} \right) \cdot \delta(\langle s_1', s_2\rangle) && \text{(Rearranging)} \\
&= \sum_{s_2 \in \text{SUPP}(\lambda_2)} \left(\frac{\lambda_2(s_2)}{\lambda_2(s_2)} \right) \cdot \delta(\langle s_1', s_2\rangle) && \text{(Definition } \delta''\text{)} \\
&= \sum_{s_2 \in \text{SUPP}(\lambda_2)} \delta(\langle s_1, s_2\rangle) && \text{(Rearranging)} \\
&= \sum_{s_2 \in S_2} \delta(\langle s_1, s_2\rangle) = \lambda_1(s_1) \ . && \text{(Def. SUPP, } \delta\text{)}
\end{aligned}$$

We can extend the sum from $\text{SUPP}(\lambda_2)$ to S_2 because $\delta''(\langle s_2, s_3\rangle) = 0$ for $s_2 \notin \text{SUPP}(\lambda_2)$ due to condition (3.1). The proof for (3.2) follows by symmetry.

(vi) Suppose that for each $i \in I$ the fact that the tuple $\langle \lambda_i^1, \lambda_i^2\rangle$ is in $\mathcal{L}(R)$ is witnessed by a weight function δ_i. We claim that

$$\langle \sum_{i \in I} w_i \cdot \lambda_i^1, \sum_{i \in I} w_i \cdot \lambda_i^2 \rangle \in \mathcal{L}(R)$$

is witnessed by the weight function $\overline{\delta} = \sum_{i \in I} w_i \cdot \delta_i$, defined pointwise. To show this we need to ensure conditions (3.1) up to (3.3) hold for these distributions and $\overline{\delta}$. Condition (3.3) holds because for any $\langle s_1, s_2\rangle \notin R$ we have that $\delta_i(\langle s_1, s_2\rangle) = 0$ for every $i \in I$ and hence $\overline{\delta}(\langle s_1, s_2\rangle) = \sum_{i \in I} w_i \cdot \delta_i(\langle s_1, s_2\rangle) = 0$, also. To show (3.1),

observe that for any $s_1 \in S_1$:

$$\begin{aligned}
\sum_{s_2 \in S_2} \overline{\delta}(\langle s_1, s_2 \rangle) &= \sum_{s_2 \in S_2} \sum_{i \in I} w_i \cdot \delta_i(\langle s_1, s_2 \rangle) && \text{(Definition } \overline{\delta}\text{)} \\
&= \sum_{i \in I} w_i \cdot \left(\sum_{s_2 \in S_2} \delta_i(\langle s_1, s_2 \rangle) \right) && \text{(Rearranging)} \\
&= \sum_{i \in I} w_i \cdot (\lambda_1(s_1)) = \left(\sum_{i \in I} w_i \cdot \lambda_1 \right)(s_1) \,. && \text{(Definition } \delta_i\text{)}
\end{aligned}$$

The proof that condition (3.2) also holds follows by symmetry. □

A.1.2 Proof of Lemma 3.10

In this section we will prove Lemma 3.10 on page 29 holds, which we now recall:

Lemma 3.10. *Let S and $\hat{S}$ be sets, let $\langle L, \leq \rangle$ be a complete lattice and let $\mathscr{R} \subseteq \hat{S} \times S$ be a relation. Moreover, let $\alpha^+ : (S \to L) \to (\hat{S} \to L)$ and $\gamma^+ : (\hat{S} \to L) \to (S \to L)$ be functions defined, for every $v : S \to L$, $\hat{v} : \hat{S} \to L$, $s \in S$ and $\hat{s} \in \hat{S}$, as*

$$\alpha^+(v)(\hat{s}) = \sup\{v(s) \mid s \in \mathscr{R}(\hat{s})\} \quad \text{and} \quad \gamma^+(\hat{v})(s) = \inf\{\hat{v}(\hat{s}) \mid \hat{s} \in \mathscr{R}^{-1}(s)\} \,.$$

We have that $\langle S \to L, \leq \rangle \underset{\alpha^+}{\overset{\gamma^+}{\leftrightarrows}} \langle \hat{S} \to L, \leq \rangle$ is a Galois connection.

Proof. The monotonicity of α^+ and γ^+ follows immediately from the definitions. Remaining to show is that $v \leq \gamma^+(\alpha^+(v))$ and $\alpha^+(\gamma^+(\hat{v})) \leq \hat{v}$ for all $v : S \to L$, $\hat{v} : \hat{S} \to L$. We first show $v \leq \gamma^+(\alpha^+(v))$. Following the pointwise ordering on $\langle S \to L, \leq \rangle$, we take an arbitrary $s \in S$ and show $v(s) \leq (\gamma^+(\alpha^+(v)))(s)$ as follows:

$$\begin{aligned}
(\gamma^+(\alpha^+(v)))(s) &= \inf\{\alpha^+(v)(\hat{s}) \mid \hat{s} \in \mathscr{R}^{-1}(s)\} && \text{(Definition } \gamma^+\text{)} \\
&= \inf\{\sup\{v(s) \mid s \in \mathscr{R}(\hat{s})\} \mid \hat{s} \in \mathscr{R}^{-1}(s)\} && \text{(Definition } \alpha^+\text{)} \\
&\geq v(s) \,.
\end{aligned}$$

The last inequality is perhaps non-obvious. Observe that if $\mathscr{R}^{-1}(s) \neq \emptyset$ then the infimum

would be over an empty set, trivially yielding the top element of $\langle L, \leq \rangle$ and trivially satisfying the inequality. However, if $\mathscr{R}^{-1}(s) \neq \emptyset$ then we are taking the infimum over a non-empty set comprising a value $\sup\{v(s) \mid s \in \mathscr{R}(\hat{s})\}$ for every $\hat{s} \in \mathscr{R}^{-1}(s)$. However, for every such value, as $\hat{s} \in \mathscr{R}^{-1}(s)$, we have that $\sup\{v(s) \mid s \in \mathscr{R}(\hat{s})\} \geq v(s)$. As every value over which we take the infimum is greater or equal to $v(s)$, the inequality holds.

The proof of $\alpha^+(\gamma^+(\hat{v})) \leq \hat{v}$ is entirely symmetric. □

A.1.3 Proof of Lemma 3.19

In this section we prove Lemma 3.19 on page 38 holds.

Lemma 3.19. *Let $M = \langle S, I, T, L, R \rangle$ be an MDP. Let $\mathbf{F}$-$FinPath_M \subseteq FinPath_M$ be precisely the set of finite paths, $\pi \in FinPath_M$, for which $L(\textsc{Last}(\pi), \mathbf{F})$ holds and $\neg L(\pi^i, \mathbf{F})$ for all $i < |\pi|$. For a fixed initial state $s \in I$ and a fixed strategy $\sigma \in Strat_M$ we have:*

$$\mathbf{Pr}^s_{M,\sigma}(\{\pi \in InfPath^s_{M,\sigma} \mid \exists i \in \mathbb{N} : L(\pi^i, \mathbf{F})\}) = \sum_{\pi \in \mathbf{F}\text{-}FinPath^s_{M,\sigma}} \textsc{Prob}_{M,\sigma}(\pi) \ .$$

Proof. We trivially have that

$$\{\pi \in InfPath^s_{M,\sigma} \mid \exists i \in \mathbb{N} : L(\pi^i, \mathbf{F})\} = \bigcup_{\pi \in \mathbf{F}\text{-}FinPath^s_{M,\sigma}} \textsc{Cyl}^s_{M,\sigma}(\pi) \ .$$

Moreover, for any two paths $\pi, \pi' \in FinPath_M$ such that $\pi \neq \pi'$ we have that $\textsc{Cyl}^s_{M,\sigma}(\pi)$ and $\textsc{Cyl}^s_{M,\sigma}(\pi')$ are disjoint. Due to the additivity of probability measures (Definition 3.4, item iii) we therefore get that:

$$\mathbf{Pr}^s_{M,\sigma}(\{\pi \in InfPath^s_{M,\sigma} \mid \exists i \in \mathbb{N} : L(\pi^i, \mathbf{F})\}) = \sum_{\pi \in \mathbf{F}\text{-}FinPath^s_{M,\sigma}} \mathbf{Pr}^s_{M,\sigma}(\textsc{Cyl}^s_{M,\sigma}(\pi)) \ .$$

Finally, recall that, by definition of $\mathbf{Pr}^s_{M,\sigma}$, we have $\mathbf{Pr}^s_{M,\sigma}(\textsc{Cyl}^s_{M,\sigma}(\pi)) = \textsc{Prob}_{M,\sigma}(\pi)$,

trivially yielding the desired equality. □

A.2 Proofs of Chapter 6

A.2.1 Proof of Lemma 6.12

In this section we prove Lemma 6.12 on page 122 holds.

Lemma 6.12. *Let $M = \langle S, I, T, L, R\rangle$ be an MDP for which there is a bound $r \in \mathbb{R}$ s.t. $R(s) \leq r$ for all $s \in S$. Moreover, let $\hat{S}'$ be an abstract state space and let $\alpha : S \to \hat{S}'$ be an abstraction function. The game $\alpha(M)$ as defined by Definition 3.28 is such that $\alpha(M) \sqsubseteq \rho(M)$.*

Proof. Let $\alpha(M) = \langle \hat{S}, \hat{I}, \hat{T}, \hat{L}, \hat{R}\rangle$ and let $\mathscr{R} \subseteq \alpha(S) \times S$ be the relation with

$$\mathscr{R} = \{\langle \hat{s}, s\rangle \in \alpha(S) \times S \mid \alpha(s) = \hat{s}\} .$$

By Definition 3.28 and the definition of $\mathscr{R}$ we have that $\hat{I} = \mathscr{R}^{-1}(I)$ and $I \subseteq \mathscr{R}(\hat{I})$. Hence, to show $\alpha(M) \sqsubseteq \rho(M)$, we only need to prove that $\mathscr{R}$ is a strong probabilistic game simulation on $\alpha(M) \uplus \rho(M)$. To show this, let $\langle \hat{s}, s\rangle$ be an arbitrary tuple in $\mathscr{R}$. We have to show condition (i) up to (iv) of Definition 6.10 hold for $\langle \hat{s}, s\rangle$. Given Definition 3.27 and 6.1 conditions (i) and (ii) are trivially satisfied for $\langle \hat{s}, s\rangle$. Remaining to show is that conditions (iii) and (iv) are satisfied for $\langle \hat{s}, s\rangle$. Recall that by the definition of embeddings any player A transition from s in $\rho(M)$ is of the form $s \to T(s)$. Moreover, by Definition 3.28, in $\alpha(M)$ there is a transition $\hat{s} \to \{\alpha(\lambda) \mid \lambda \in T(s)\}$. Suppose we match the player A transition $s \to \{T(s)\}$ in $\rho(M)$ with the player A transition $\hat{s} \to \{\alpha(\lambda) \mid \lambda \in T(s)\}$ in $\alpha(M)$ for both condition (iii) and (iv). Given the definition of $\mathscr{R}$ and Definition 3.1 and 3.27 it is easy to see $\langle \alpha(\lambda), \lambda\rangle \in \mathcal{L}(\mathscr{R})$ for every $\lambda \in T(s)$. Hence, the conditions (iii) and (iv) are trivially satisfied. This means that all conditions hold for all tuples in $\mathscr{R}$ and hence $\mathscr{R}$ is a strong probabilistic game simulation. With the additional observation on initial states this means that $\alpha(M) \sqsubseteq \rho(M)$. □

A.2.2 Proof of Lemma 6.14

In this section we will show that $\sqsubseteq$ is a preorder. That is, we will show:

Lemma 6.14. *The abstraction relation $\sqsubseteq$ is a* preorder.

Mainly due to the presence of combined transitions in Definition 6.10, proving that $\sqsubseteq$ is transitive is not straightforward. To deal with combined transitions, we first introduce lemmas that guarantee the existence of certain combined player A and player C transitions (namely Lemma A.1, A.2 and A.3). Then, we will lift the result in Lemma A.1.1 (item vi) to player A transitions (Lemma A.4). As a final preparation for proving Lemma 6.14 we will show that, in Definition 6.10, we can replace the universal quantification over normal player A and player C transitions with a universal quantification over *combined* transitions (Lemma A.5). This will then finally allow us to turn our attention to Lemma 6.14.

We start by showing that weighted combinations of combined transitions are combined transitions again. We start with player C:

Lemma A.1. *Let $\Lambda \in \overline{\mathbb{P}}\mathbb{D}S$ be some set of distributions on some set, S, and let $\{\lambda_i\}_{i\in I}$ be a family of distributions over S such that $\Lambda \xrightarrow{Cmb} \lambda_i$ is a combined player C transition for each $i \in I$. For any family of weights $\{w_i\}_{i\in I}$ we have that*

$$\Lambda \xrightarrow{Cmb} \sum_{i\in I} w_i \cdot \lambda_i$$

is a valid combined player C transition, also.

Proof. Let $\{K_i\}_{i\in I}$ be a family of index sets and, for each $i \in I$, let $\{\lambda_k^i\}_{k\in K_i}$ and $\{w_k^i\}_{k\in K_i}$ be a family of sets of distributions and weights, respectively, such that $\lambda_i = \sum_{k\in K_i} w_k^i \cdot \lambda_k^i$

and $\lambda_k^i \in \Lambda$ for each $k \in K_i$. We have the equality

$$\begin{aligned}
\sum_{i \in I} w_i \cdot \lambda_i &= \sum_{i \in I} w_i \cdot \left(\sum_{k \in K_i} w_k^i \cdot \lambda_k^i \right) && \text{(Definition } \lambda_i\text{)} \\
&= \sum_{i \in I} \sum_{k \in K_i} w_i \cdot w_k^i \cdot \lambda_k^i && \text{(Rearranging)} \\
&= \sum_{i \in I, k \in K_i} w_i \cdot w_k^i \cdot \lambda_k^i && \text{(Rearranging)}
\end{aligned}$$

Due to this equality it is easy to see that $\Lambda \xrightarrow{\text{Cmb}} \sum_{i \in I} w_i \cdot \lambda_i$ is the combined player C transition introduced by the family of weights $\{w_i \cdot w_k^i\}_{i \in I, k \in K_i}$ and distributions $\{\lambda_k^i\}_{i \in I, k \in K_i}$. □

We remark that, in the proof of Lemma A.1, we use families that are indexed by both $i \in I$ and $k \in K_i$ simultaneously. This is merely a question of notation — they can easily be rewritten using a single (countable) index set.

We now show that weighted combinations of combined transitions for player A are combined player A transitions again:

Lemma A.2. *Let $s \in S$ be a state and let $\{\Lambda_i\}_{i \in I}$ be a* finite *family of sets of distributions in $\overline{\mathbb{P}}\mathbb{D}S$ such that $s \xrightarrow{Cmb} \Lambda_i$ is a combined player A transition for each $i \in I$. For any* finite *family of weights $\{w_i\}_{i \in I}$ we have that*

$$s \xrightarrow{Cmb} \sum_{i \in I} w_i \cdot \Lambda_i$$

is a valid combined player A transition, also.

Proof. Let $\{K_i\}_{i \in I}$ be a finite family of finite index sets and, for each $i \in I$, let $\{\Lambda_k^i\}_{k \in K_i}$ and $\{w_k^i\}_{k \in K_i}$ be finite families of sets of distributions and weights, respectively, such that $\Lambda_k^i \in T(s)$ for each $k \in K_i$ and $\Lambda_i = \sum_{k \in K_i} w_k^i \cdot \Lambda_k^i$. We will show that the combined transition induced by the families $\{w_i \cdot w_k^i\}_{i \in I, k \in K_i}$ and and $\{\Lambda_k^i\}_{i \in I, k \in K_i}$ yields the desired combined player A transition. We show this with the following equalities (by applying

Definition 6.8 repeatedly):

$$\begin{aligned}
\textstyle\sum_{i\in I}\ w_i\cdot\Lambda_i &= \textstyle\sum_{i\in I}\ w_i\cdot\left(\sum_{k\in K_i} w_k^i\cdot\Lambda_k^i\right) && \text{(Definition } \Lambda_i\text{)}\\
&= \textstyle\sum_{i\in I}\ w_i\cdot\left\{\sum_{k\in K_i} w_k^i\cdot\lambda_k^i \mid \{\lambda_k^i\}_{k\in K_i} \text{ s.t. } \lambda_k^i\in\Lambda_k^i\right\} && \text{(Definition 6.8)}\\
&\overset{(1)}{=} \left\{\textstyle\sum_{i\in I,k\in K_i} w_i\cdot w_k^i\cdot\lambda_k^i \mid \{\lambda_k^i\}_{i\in I,k\in K_i} \text{ s.t. } \lambda_k^i\in\Lambda_k^i\right\} && \text{(Definition 6.8)}\\
&= \textstyle\sum_{i\in I,k\in K_i}\left(w_i\cdot w_k^i\cdot\Lambda_k^i\right) && \text{(Definition 6.8)}
\end{aligned}$$

Note that, according to Definition 6.8, the set on the right-hand side of equality (1) should include a distribution $\sum_{i\in I} w_i\cdot\lambda_i$ for every family of distributions $\{\lambda_i\}_{i\in I}$ with

$$\lambda_i\in\left\{\textstyle\sum_{k\in K_i} w_k^i\cdot\lambda_k^i \mid \{\lambda_k^i\}_{k\in K_i} \text{ such that } \lambda_k^i\in\Lambda_k^i\right\}$$

for each $i\in I$. Instead we (equivalently) include the distribution $\sum_{i\in I} w_i\cdot\left(\sum_{k\in K_i} w_k^i\cdot\lambda_k^i\right)$ for every family $\{\lambda_k^i\}_{i\in I,k\in K_i}$ with $\lambda_k^i\in\Lambda_k^i$ for every $i\in I$ and $k\in K_i$.

With the equality above, the combined player A transition induced by $\{w_i\cdot w_k^i\}_{i\in I,k\in K_i}$ and $\{\Lambda_k^i\}_{i\in I,k\in K_i}$ is $s\xrightarrow{\text{Cmb}}\sum_{i\in I}\ w_i\cdot\Lambda_i$. To see this is a valid combined player A transition, recall that both I is finite and, for every $i\in I$, the index set K_i is finite. □

Again, in the proof of Lemma A.2, we use families that are indexed by both $i\in I$ and $k\in K_i$ simultaneously. In this instance we easily rewrite this to families over a single finite index set. We now strengthen Lemma A.1 by considering the available combined transitions from a player C state $\sum_{i\in I}\ w_i\cdot\Lambda_i$:

Lemma A.3. *Let $\{\Lambda_i\}_{i\in I}$ and $\{\lambda_i\}_{i\in I}$ be a* finite *family of sets of distributions in $\overline{\mathbb{P}}\mathbb{D}S$ and a* finite *family of distributions in $\mathbb{D}S$, respectively, such that $\Lambda_i\xrightarrow{Cmb}\lambda_i$ is a combined player C transition for each $i\in I$, then, for each family of weights $\{w_i\}_{i\in I}$ we have that*

$$\sum_{i\in I} w_i\cdot\Lambda_i\xrightarrow{Cmb}\sum_{i\in I} w_i\cdot\lambda_i \tag{A.1}$$

is a combined player C transition, also.

Proof. Let *Fam* the countable set of all families of distributions $\{\mu_i\}_{i\in I}$ with $\mu_i\in\Lambda_i$

for all $i \in I$. Intuitively, elements of *Fam* represent the player C states available from $\sum_{i\in I} w_i \cdot \Lambda_i$. That is, every family $\{\mu_i\}_{i\in I} \in \mathit{Fam}$ induces the (normal) player C transition

$$\sum_{i\in I} w_i \cdot \Lambda_i \to \sum_{i\in I} w_i \cdot \mu_i \ .$$

Hence, by Lemma A.1, all we need to show is that (A.1) is a weighted combination of such transitions. Because I is finite it must be the case that *Fam* is countable. This means we can use *Fam* as an index set for a combined player C transition.

To define this combined transition, let us suppose that each player C transition $\Lambda_i \xrightarrow{\text{Cmb}} \lambda_i$ is induced by a family of distributions $\{\lambda_k^i\}_{k\in K_i}$ in Λ_i and a family of weights $\{w_k^i\}_{k\in K_i}$. That is, for each $i \in I$ we have:

$$\lambda_i = \sum_{k\in K_i} w_k^i \cdot \lambda_k^i \ .$$

We will show that (A.1) is a combined transition from $\sum_{i\in I}\ w_i \cdot \Lambda_i$ indexed by elements of *Fam*. That is, we define a family of distributions $\{\lambda_f\}_{f\in \mathit{Fam}}$ available in $\sum_{i\in I}\ w_i \cdot \Lambda_i$ and weights $\{w_f\}_{f\in \mathit{Fam}}$ such that (A.1) is

$$\sum_{i\in I} w_i \cdot \Lambda_i \xrightarrow{\text{Cmb}} \sum_{f\in \mathit{Fam}} w_f \cdot \lambda_f \ .$$

For each family $f = \{\mu_i\}_{i\in I}$ in *Fam* let us define

$$\lambda_f = \sum_{i\in I} w_i \cdot \mu_i \ , \qquad \text{and}$$
$$w_f = \textstyle\prod_{i\in I} \left(\sum_{\lambda_k^i = \mu_i} w_k^i\right) \ .$$

Note that I is both non-empty and finite. For a given $i \in I$, we use a shorthand notation to sum over those indices, $k \in K_i$, for which $\lambda_k^i = \mu_i$.

It remains to show that $\sum_{f\in \mathit{Fam}} w_f \cdot \lambda_f$ is equal to $\sum_{i\in I}\ w_i \cdot \lambda_i$. Before we show this we first define, for every $i \in I$, the set Fam_i as the set of all families of distributions

$\{\mu_j\}_{j\in I\setminus\{i\}}$ with $\mu_j \in \Lambda_j$ for all $j \in I \setminus \{i\}$.

$$
\begin{aligned}
\sum_{f\in Fam} w_f \cdot \lambda_f &\overset{(1)}{=} \textstyle\sum_{(\{\mu_n\}_{n\in I})\in Fam} \left(\left(\prod_{j\in I} \left(\sum_{\lambda_k^j=\mu_j} w_k^j\right)\right) \cdot \left(\sum_{i\in I}\ w_i \cdot \mu_i\right)\right) \\
&= \textstyle\sum_{i\in I}\ \sum_{(\{\mu_n\}_{n\in I})\in Fam} \left(\left(\prod_{j\in I} \left(\sum_{\lambda_k^j=\mu_j} w_k^j\right)\right) \cdot w_i \cdot \mu_i\right) \\
&\overset{(2)}{=} \textstyle\sum_{i\in I}\ \sum_{\lambda_i'\in\Lambda_i}\ \sum_{(\{\mu_n\}_{n\in I\setminus\{i\}})\in Fam_i} \left(\left(\prod_{j\in I\setminus\{i\}} \left(\sum_{\lambda_k^j=\mu_j} w_k^j\right)\right) \cdot \right. \\
&\qquad \left. \left(\sum_{\lambda_k^i=\lambda_i'} w_k^i\right) \cdot w_i \cdot \lambda_i'\right) \\
&= \textstyle\sum_{i\in I}\ \sum_{\lambda_i'\in\Lambda_i} \left(\left(\sum_{(\{\mu_n\}_{n\in I\setminus\{i\}})\in Fam_i} \left(\prod_{j\in I\setminus\{i\}} \left(\sum_{\lambda_k^j=\mu_j} w_k^j\right)\right)\right) \cdot \right. \\
&\qquad \left. \left(\sum_{\lambda_k^i=\lambda_i'} w_k^i\right) \cdot w_i \cdot \lambda_i'\right) \\
&\overset{(3)}{=} \textstyle\sum_{i\in I}\ \sum_{\lambda_i'\in\Lambda_i} \left(\left(\prod_{j\in I\setminus\{i\}} \left(\sum_{\mu_j\in\Lambda_j} \left(\sum_{\lambda_k^j=\mu_j}\ w_k^j\right)\right)\right) \cdot \right. \\
&\qquad \left. \left(\sum_{\lambda_k^i=\lambda_i'} w_k^i\right) \cdot w_i \cdot \lambda_i'\right) \\
&\overset{(4)}{=} \textstyle\sum_{i\in I}\ \sum_{\lambda_i'\in\Lambda_i} \left(\left(\sum_{\lambda_k^i=\lambda_i'} w_k^i\right) \cdot w_i \cdot \lambda_i'\right) \\
&= \textstyle\sum_{i\in I}\ w_i \cdot \left(\sum_{k\in K_i} w_k^i \cdot \lambda_k^i\right) = \sum_{i\in I}\ w_i \cdot \lambda_i
\end{aligned}
$$

Equality (1) simply expands the definition of w_f and λ_f. Equality (2) splits the sum over all families $(\{\mu_n\}_{n\in I}) \in \mathit{Fam}$ into a sum over all families $(\{\mu_n\}_{n\in I}) \in \mathit{Fam}_i$ (i.e. families without a distribution for i) and all distributions λ_i' and updates the product accordingly. In equality (3) we use the fact that for a finite number of countable sums that are *absolutely converging*[1] the product of these countable sums is a Cauchy product. The summation over families Fam_i is a rearrangement of this Cauchy product. Finally, equality (4) holds because, for each $j \in I \setminus \{i\}$, we have that

$$\textstyle\sum_{\mu_j\in\Lambda_j} \sum_{\lambda_k^j=\mu_j}\ w_k^j = \sum_{k\in K_i}\ w_k^j = 1\ .$$

Hence, we can eliminate the product term.

As $\sum_{f\in Fam} w_f$ equals $\sum_{i\in I}\ w_i \cdot \lambda_i$ we have found a combined player C transition that satisfies our lemma. □

[1] In our setting, where all terms are non-negative, this means the supremum over all partial sums is finite.

Finally, before we prove that we can substitute the universal quantification over combined transitions with normal transitions, we need a player A analogue of Lemma A.1.1 (item vi):

Lemma A.4. *Let S be a set and let $\mathscr{R} \subseteq \hat{S} \times S$ be a relation. Let $\{\hat{\Lambda}_i\}_{i\in I}$ and $\{\Lambda_i\}_{i\in I}$ be* finite *families of sets of distributions in $\overline{\mathbb{P}}\mathbb{D}\hat{S}$ and $\overline{\mathbb{P}}\mathbb{D}S$, respectively, such that*

$$\forall \Lambda_i \to \lambda : \exists \hat{\Lambda}_i \xrightarrow{Cmb} \hat{\lambda} : \langle \hat{\lambda}, \lambda \rangle \in \mathcal{L}(\mathscr{R}) \tag{A.2}$$

for all $i \in I$. For every family of weights, $\{w_i\}_{i\in I}$, the player C states, $\Lambda = \sum_{i\in I} w_i \cdot \Lambda_i$ and $\hat{\Lambda} = \sum_{i\in I} w_i \cdot \hat{\Lambda}_i$, are such that

$$\forall \Lambda \to \lambda : \exists \hat{\Lambda} \xrightarrow{Cmb} \hat{\lambda} : \langle \hat{\lambda}, \lambda \rangle \in \mathcal{L}(\mathscr{R}) \; . \tag{A.3}$$

Proof. To show (A.3), let us consider an arbitrary player C transition $\Lambda \to \lambda$. By definition of Λ it must be that λ is of the form $\sum_{i\in I} w_i \cdot \lambda_i$ for some family of distributions $\{\lambda_i\}_{i\in I}$ with $\lambda_i \in \Lambda_i$ for all $i \in I$. For each $i \in I$, $\Lambda_i \to \lambda_i$ is a player C transition and we can use (A.2) to obtain a player C transition $\hat{\Lambda}_i \xrightarrow{\text{Cmb}} \lambda_i$ such that $\langle \lambda_i, \lambda_i \rangle \in \mathcal{L}(\mathscr{R})$. From Lemma A.3 we learn $\sum_{i\in I} w_i \cdot \hat{\Lambda}_i \xrightarrow{\text{Cmb}} \sum_{i\in I} w_i \cdot \hat{\lambda}_i$ is a valid combined player C transition and from Lemma 3.2 (item vi) we learn $\langle \lambda, \sum_{i\in I} w_i \cdot \hat{\lambda}_i \rangle \in \mathcal{L}(\mathscr{R})$. Because we did not make any assumptions on the transition $\Lambda \to \lambda$, condition (A.3) must hold. □

We are now finally in a position to show we can interchange the universal quantification of normal player A and player C transitions in Definition 6.10 with a universal quantification over *combined* transitions:

Lemma A.5. *For games that are finitely branching for player A we can replace condition (iii) and (iv) of Definition 6.10 with:*

(iii) $\forall s \xrightarrow{Cmb} \Lambda : \exists \hat{s} \xrightarrow{Cmb} \hat{\Lambda} : \forall \Lambda \xrightarrow{Cmb} \lambda : \exists \hat{\Lambda} \xrightarrow{Cmb} \hat{\lambda} : \langle \hat{\lambda}, \lambda \rangle \in \mathcal{L}(\mathscr{R})$ *and*

(iv) $\forall s \xrightarrow{Cmb} \Lambda : \exists \hat{s} \xrightarrow{Cmb} \hat{\Lambda} : \forall \hat{\Lambda} \xrightarrow{Cmb} \hat{\lambda} : \exists \Lambda \xrightarrow{Cmb} \lambda : \langle \hat{\lambda}, \lambda \rangle \in \mathcal{L}(\mathscr{R})$.

Proof. Let $G = \langle S, I, T, L, R \rangle$ be a game and let $\mathscr{R} \subseteq S \times S$ be a relation on S. We will show that (iii) and (iv) of Definition 6.10 can be replaced with the conditions above.

We first show that, for any non-empty sets of distributions $\hat{\Lambda}, \Lambda \in \overline{\mathbb{P}}\mathbb{D}S$, we have:

$$\begin{aligned} \forall \Lambda \to \lambda : \exists \hat{\Lambda} \xrightarrow{\text{Cmb}} \hat{\lambda} : \langle \hat{\lambda}, \lambda \rangle \in \mathcal{L}(\mathscr{R}) \iff \\ \forall \Lambda \xrightarrow{\text{Cmb}} \lambda : \exists \hat{\Lambda} \xrightarrow{\text{Cmb}} \hat{\lambda} : \langle \hat{\lambda}, \lambda \rangle \in \mathcal{L}(\mathscr{R}) \end{aligned} \tag{A.4}$$

and

$$\begin{aligned} \forall \hat{\Lambda} \to \hat{\lambda} : \exists \Lambda \xrightarrow{\text{Cmb}} \lambda : \langle \hat{\lambda}, \lambda \rangle \in \mathcal{L}(\mathscr{R}) \iff \\ \forall \hat{\Lambda} \xrightarrow{\text{Cmb}} \hat{\lambda} : \exists \Lambda \xrightarrow{\text{Cmb}} \lambda : \langle \hat{\lambda}, \lambda \rangle \in \mathcal{L}(\mathscr{R}) \,. \end{aligned} \tag{A.5}$$

We first show (A.4). The direction ($\Leftarrow$) holds trivially: every player C transition is a combined player C transition. To show ($\Rightarrow$) let us take an arbitrary combined player C transition $\Lambda \xrightarrow{\text{Cmb}} \sum_{i \in I} w_i \cdot \lambda_i$, for some family of weights $\{w_i\}_{i \in I}$ and family of distributions $\{\lambda_i\}_{i \in I}$ in Λ, and show that there exists a combined player C transition $\hat{\Lambda} \xrightarrow{\text{Cmb}} \hat{\lambda}$ such that $\langle \hat{\lambda}, \sum_{i \in I} w_i \cdot \lambda_i \rangle \in \mathcal{L}(\mathscr{R})$.

For each $i \in I$, we know $\Lambda \to \lambda_i$ is a normal player C transition and hence we can use the premise of the implication to see there exists a combined player C transition $\hat{\Lambda} \xrightarrow{\text{Cmb}} \hat{\lambda}_i$ such that $\langle \lambda_i, \hat{\lambda}_i \rangle \in \mathcal{L}(\mathscr{R})$. From Lemma A.1 we get that $\hat{\Lambda} \xrightarrow{\text{Cmb}} \sum_{i \in I} w_i \cdot \hat{\lambda}_i$ is a valid combined player C transition and from Lemma 3.2 (item vi) we get that $\langle \sum_{i \in I} w_i \cdot \hat{\lambda}_i, \sum_{i \in I} w_i \cdot \lambda_i \rangle \in \mathcal{L}(\mathscr{R})$. As we made no assumption on the player C transition this means (A.4) holds. By employing (A.4) on $\mathscr{R}^{-1}$ and due to Lemma 3.2 (item iii) we get that (A.5) holds, also.

We now show that (iii) and (iv) in Definition 6.10 are equivalent to (iii) and (iv) as defined above. Let $\hat{s}, s \in S$ be arbitrary states. Due to (A.4) and (A.5), it is sufficient to show that:

$$\begin{aligned} \forall s \to \Lambda : \exists \hat{s} \xrightarrow{\text{Cmb}} \hat{\Lambda} : \forall \Lambda \to \lambda : \exists \hat{\Lambda} \xrightarrow{\text{Cmb}} \hat{\lambda} : \langle \hat{\lambda}, \lambda \rangle \in \mathcal{L}(\mathscr{R}) \iff \\ \forall s \xrightarrow{\text{Cmb}} \Lambda : \exists \hat{s} \xrightarrow{\text{Cmb}} \hat{\Lambda} : \forall \Lambda \to \lambda : \exists \hat{\Lambda} \xrightarrow{\text{Cmb}} \hat{\lambda} : \langle \hat{\lambda}, \lambda \rangle \in \mathcal{L}(\mathscr{R}) \end{aligned} \tag{A.6}$$

$$\begin{aligned} \forall s \to \Lambda : \exists \hat{s} \xrightarrow{\text{Cmb}} \hat{\Lambda} : \forall \hat{\Lambda} \to \hat{\lambda} : \exists \Lambda \xrightarrow{\text{Cmb}} \lambda : \langle \hat{\lambda}, \lambda \rangle \in \mathcal{L}(\mathscr{R}) \iff \\ \forall s \xrightarrow{\text{Cmb}} \Lambda : \exists \hat{s} \xrightarrow{\text{Cmb}} \hat{\Lambda} : \forall \hat{\Lambda} \to \hat{\lambda} : \exists \Lambda \xrightarrow{\text{Cmb}} \lambda : \langle \hat{\lambda}, \lambda \rangle \in \mathcal{L}(\mathscr{R}) \,. \end{aligned} \tag{A.7}$$

We first show (A.6). Again, the direction ($\Leftarrow$) is trivial. To show direction ($\Rightarrow$) consider an arbitrary player A transition $s \xrightarrow{\text{Cmb}} \sum_{i\in I} w_i \cdot \Lambda_i$ for some finite family of weights $\{w_i\}_{i\in I}$ and finite family of sets of distributions $\{\Lambda_i\}_{i\in I}$ in $T(s)$. For each $i \in I$, we know $s \to \Lambda_i$ is a normal player A transition and hence (from the premise of the implication) there exists a combined player A transition $\hat{s} \xrightarrow{\text{Cmb}} \hat{\Lambda}_i$ such that

$$\forall \Lambda_i \to \lambda : \exists \hat{\Lambda}_i \xrightarrow{\text{Cmb}} \hat{\lambda} : \langle \hat{\lambda}, \lambda \rangle \in \mathcal{L}(\mathscr{R}) \ .$$

We can apply Lemma A.2 to get that $\hat{s} \xrightarrow{\text{Cmb}} \sum_{i\in I} w_i \cdot \hat{\Lambda}_i$ is a valid combined player A transition of G and, from Lemma A.4, we get that

$$\forall \sum_{i\in I} w_i \cdot \Lambda_i \to \lambda : \exists \sum_{i\in I} w_i \cdot \hat{\Lambda}_i \xrightarrow{\text{Cmb}} \hat{\lambda} : \langle \hat{\lambda}, \lambda \rangle \in \mathcal{L}(\mathscr{R}) \ .$$

Hence, the combined player A transition $\hat{s} \xrightarrow{\text{Cmb}} \sum_{i\in I} w_i \cdot \hat{\Lambda}_i \xrightarrow{\text{Cmb}} \hat{\lambda}$ satisfies condition (A.6) for the player A transition $s \xrightarrow{\text{Cmb}} \sum_{i\in I} w_i \cdot \Lambda_i \to \lambda$. As we made no assumption about the transition $s \xrightarrow{\text{Cmb}} \sum_{i\in I} w_i \cdot \Lambda_i$ this means that the direction ($\Rightarrow$) is also satisfied and (A.6) holds. This implies that (iii) of Definition 6.10 are equivalent to condition (A.6) as stated in the definition of this lemma. To show (A.7) we use again Lemma A.2 and apply Lemma A.4 on $\mathscr{R}^{-1}$ (using Lemma 3.2, item iii). This implies the same argument holds for (iv) of Definition 6.10. □

Through Lemma A.5 we are finally in a position to prove that $\sqsubseteq$ is a preorder:

Lemma 6.14. *The abstraction relation $\sqsubseteq$ is a* preorder.

Proof. We first remark that, if $\hat{G} \sqsubseteq G$, then a strong probabilistic game simulation $\mathscr{R} \subseteq \hat{S} \times S$ with $I \subseteq \mathscr{R}(\hat{I})$ and $\hat{I} \subseteq \mathscr{R}^{-1}(I)$ always exists. To see this consider any two games $\hat{G} = \langle \hat{S}, \hat{I}, \hat{T}, \hat{L}, \hat{R} \rangle$ and $G = \langle S, I, T, L, R \rangle$ and a strong probabilistic game simulation $\mathscr{R} \subseteq (\hat{S} \uplus S) \times (\hat{S} \uplus S)$ on $\hat{G} \uplus G$ with $I \subseteq \mathscr{R}(\hat{I})$ and $\hat{I} \subseteq \mathscr{R}^{-1}(I)$. The relation $\mathscr{R} \cap (\hat{S} \times S)$ is trivially also a strong probabilistic game simulation on $\hat{G} \uplus G$ satisfying the same property on initial states.

To show $\sqsubseteq$ is a preorder we need to show it is both reflexive and transitive. The

reflexivity of $\sqsubseteq$ follows trivially. To show $\sqsubseteq$ is transitive let G_1, G_2 and G_3 be games such that $G_1 \sqsubseteq G_2$ and $G_2 \sqsubseteq G_3$. Suppose that $G_i = \langle S_i, I_i, T_i, L_i, R_i \rangle$ for every $i \in \{1, 2, 3\}$. Let $\mathscr{R}_{1,2} \subseteq S_1 \times S_2$ and $\mathscr{R}_{2,3} \subseteq S_2 \times S_3$ be strong probabilistic game simulations on $G_1 \uplus G_2$ and $G_2 \uplus G_3$, respectively, such that $I_2 \subseteq \mathscr{R}_{1,2}(I_1)$, $I_1 \subseteq \mathscr{R}_{1,2}^{-1}(I_2)$, $I_3 \subseteq \mathscr{R}_{2,3}(I_2)$ and $I_2 \subseteq \mathscr{R}_{2,3}^{-1}(I_3)$.

We will show that $G_1 \sqsubseteq G_3$ via a relation $\mathscr{R} \subseteq S_1 \times S_3$ defined as $\mathscr{R} = \mathscr{R}_{2,3} \circ \mathscr{R}_{1,2}$. It is easy to see that $I_1 \subseteq \mathscr{R}(I_3)$ and $I_3 \subseteq \mathscr{R}^{-1}(I_1)$ (as I_2 is necessarily non-empty). It remaining to show that $\mathscr{R}$ is indeed a strong probabilistic game simulation. That is, we have to show that condition (i) up to (iv) of Definition 6.10 hold for every $\langle s_1, s_3 \rangle \in \mathscr{R}$.

We first show conditions (i) and (ii). For $\langle s_1, s_3 \rangle$ to be in $\mathscr{R}$, there must be some $s_2 \in S_2$ such that $\langle s_1, s_2 \rangle \in \mathscr{R}_{1,2}$ and $\langle s_2, s_3 \rangle \in \mathscr{R}_{2,3}$. As $\mathscr{R}_{1,2}$ and $\mathscr{R}_{2,3}$ are strong probabilistic game simulations, it follows that $L_1(s_1, a) \leq L_2(s_2, a) \leq L_3(s_3, a)$ for every $a \in \mathrm{AP}$ and $R_1(s_1) \leq R_2(s_2) \leq R_3(s_3)$. Hence, conditions (i) and (ii) hold for $\langle s_1, s_3 \rangle$ due to the transitivity of $\leq$.

It remains to show that (iii) and (iv) of Definition 6.10 hold for every $\langle s_1, s_3 \rangle \in \mathscr{R}$ in Definition 3.8. To do this, we will use the fact that conditions (iii) and (iv) of Definition 6.10 can be equivalently defined with (iii) and (iv) in Lemma A.5. To show this we use Lemma A.5. Again, for $\langle s_1, s_3 \rangle$ to be in $\mathscr{R}$, there must be some $s_2 \in S_2$ such that $\langle s_1, s_2 \rangle \in \mathscr{R}_{1,2}$ and $\langle s_2, s_3 \rangle \in \mathscr{R}_{2,3}$.

We first focus on (iii) of Lemma A.5. Let $s_1 \xrightarrow{\mathrm{Cmb}} \Lambda_1$ be any combined player A transition in G_1 from s_1. Because $\mathscr{R}_{1,2}$ and $\mathscr{R}_{2,3}$ are strong probabilistic game simulations there is a combined player A transition $s_2 \xrightarrow{\mathrm{Cmb}} \Lambda_2$ in G_2 and a combined player A transition $s_3 \xrightarrow{\mathrm{Cmb}} \Lambda_3$ in G_3 such that

$$\forall \Lambda_1 \xrightarrow{\mathrm{Cmb}} \lambda_1 : \exists \Lambda_2 \xrightarrow{\mathrm{Cmb}} \lambda_2 : \langle \lambda_1, \lambda_2 \rangle \in \mathcal{L}(\mathscr{R}_{1,2}) \text{ and}$$
$$\forall \Lambda_2 \xrightarrow{\mathrm{Cmb}} \lambda_2 : \exists \Lambda_3 \xrightarrow{\mathrm{Cmb}} \lambda_3 : \langle \lambda_2, \lambda_3 \rangle \in \mathcal{L}(\mathscr{R}_{2,3}) \,.$$

Moreover, because of Lemma 3.2 (item v), we have that if $\langle \lambda_1, \lambda_2 \rangle \in \mathscr{R}_{1,2}$ and $\langle \lambda_2, \lambda_3 \rangle \in \mathscr{R}_{2,3}$ then $\langle \lambda_1, \lambda_3 \rangle \in \mathscr{R}$. Hence, $s_3 \xrightarrow{\mathrm{Cmb}} \Lambda_3$, satisfies (iii) for the transition $s_1 \xrightarrow{\mathrm{Cmb}} \Lambda_2$.

The condition (iv) in Lemma A.5 can be shown similarly. That is, let $s_1 \xrightarrow{\mathrm{Cmb}} \Lambda_1$

be any combined player A transition in G_1 from s_1. Because $\mathscr{R}_{1,2}$ and $\mathscr{R}_{2,3}$ are strong probabilistic game simulations there is a combined player A transition $s_2 \xrightarrow{\text{Cmb}} \Lambda_2$ in G_2 and a combined player A transition $s_3 \xrightarrow{\text{Cmb}} \Lambda_3$ in G_3 such that

$$\forall \Lambda_3 \xrightarrow{\text{Cmb}} \lambda_3 : \exists \Lambda_2 \xrightarrow{\text{Cmb}} \lambda_2 : \langle \lambda_2, \lambda_3 \rangle \in \mathcal{L}(\mathscr{R}_{2,3}) \text{ and}$$
$$\forall \Lambda_2 \xrightarrow{\text{Cmb}} \lambda_2 : \exists \Lambda_1 \xrightarrow{\text{Cmb}} \lambda_1 : \langle \lambda_1, \lambda_2 \rangle \in \mathcal{L}(\mathscr{R}_{1,2}) \,.$$

Hence, it follows that $s_3 \xrightarrow{\text{Cmb}} \Lambda_3$, satisfies (iv) for the transition $s_1 \xrightarrow{\text{Cmb}} \Lambda_2$.

As all conditions hold it must be that $\mathscr{R}$ is a strong probabilistic game simulation on $G_1 \uplus G_3$. This implies that $G_1 \sqsubseteq G_3$ and, as we took arbitrary games, G_1, G_2 and G_3, this means that $\sqsubseteq$ is transitive and, hence, that $\sqsubseteq$ is a preorder. □

A.2.3 Proof of Lemma 6.15

In this section we prove Lemma 6.15 on page 125 holds.

Lemma 6.15. *Let $M = \langle S, I, T, L, R \rangle$ be an MDP for which there is a bound $r \in \mathbb{R}$ s.t. $R(s) \leq r$ for all $s \in S$. Moreover, let $\hat{S}'$ be an abstract state space, let $\alpha : S \to \hat{S}'$ be an abstraction function for which $\alpha(S)$ is finite and let $\mathscr{R} \subseteq \alpha(S) \times S$ be the relation defined as $\{\langle \hat{s}, s \rangle \in \alpha(S) \times S \mid \alpha(s) = \hat{s}\}$. A game $\hat{G} = \langle \hat{S}, \hat{I}, \hat{T}, \hat{L}, \hat{R} \rangle$ exists for which $\hat{G} \sqsubseteq \rho(M)$ with $\mathscr{R}$ and*

- *$\hat{T}(\hat{s}) = \{\{[\hat{s}']\} \mid \hat{s}' \in \hat{S}\} \cup \{\{[\hat{s}'] \mid \hat{s}' \in \hat{S}\}\}$ for every $\hat{s} \in \hat{S}$.*

Moreover, for any such $\hat{G}$ we have that any other game $\hat{G}' = \langle \hat{S}, \hat{I}, \hat{T}', \hat{L}, \hat{R} \rangle$ such that $\hat{G}' \sqsubseteq \rho(M)$ with $\mathscr{R}$ and such that $\hat{G}'$ agrees with $\hat{G}$ on $\hat{I}, \hat{L}$ and $\hat{R}$, is abstracted by $\hat{G}$.

Proof. Let us first show a game $\hat{G}$ meeting our criteria exists. We can define $\hat{S}$, $\hat{I}$, $\hat{L}$ and $\hat{R}$ as we would in $\alpha(M)$ and use Definition 6.12 to show the conditions on initial states as well as condition (i) and (ii) of Definition 6.10 are satisfied for every $\langle \hat{s}, s \rangle \in \mathscr{R}$.

Remaining to show for our first claim is that (iii) and (iv) of Definition 6.10 are satisfied for every $\langle \hat{s}, s \rangle \in \mathscr{R}$. Let us first focus on (iii). Let $s \to \Lambda$ be an arbitrary player

A transition of $\rho(M)$. We use the player A transition $\hat{s} \to \{[\hat{s}'] \mid \hat{s}' \in \hat{S}\}$ of $\hat{G}$ to match this transition. We need to show that every player C transition $\Lambda \to \lambda$ of $\rho(M)$ has a matching player C transition from $\{[\hat{s}'] \mid \hat{s}' \in \hat{S}\}$ in $\hat{G}$.

To show this we use the fact that $\mathscr{R}$ is right-total and that, by Definition 3.2 (item (i)), the relation $\mathcal{L}(\mathscr{R})$ is right-total, also. Hence, to match player C transition $\Lambda \to \lambda$ of $\rho(M)$ we use the right-totalness of $\mathcal{L}(\mathscr{R})$ to get an $\hat{\lambda} \in \mathbb{D}(\alpha(S))$ such that $\langle \hat{\lambda}, \lambda \rangle \in \mathcal{L}(\mathscr{R})$. By the definition of combined player C transitions, the transition $\{[\hat{s}'] \mid \hat{s}' \in \hat{S}\} \xrightarrow{\text{Cmb}} \hat{\lambda}$ exists.

Remaining to show for our first claim is that condition (iv) of Definition 6.10 is satisfied for every $\langle \hat{s}, s \rangle \in \mathscr{R}$. To see this holds, let $s \to \Lambda$ be any player A transition in $\rho(M)$. We need to match this with a combined player A transition from $\hat{s}$ in $\hat{G}$. To do this let $\Lambda \to \lambda$ be an arbitrary player C transition in $\rho(M)$ and, by the right-totalness of $\mathcal{L}(\mathscr{R})$, let us take a $\hat{\lambda} \in \mathbb{D}(\alpha(S))$ such that $\langle \hat{\lambda}, \lambda \rangle \in \mathcal{L}(\mathscr{R})$. Now let us write $\hat{\lambda}$ as a *finite* sum $\sum_{i \in I} w_i \cdot [\hat{s}_i]$ of point distributions. Note this sum is finite due to the finiteness restriction on $\alpha(S)$. Now let us match the player A transition $s \to \Lambda$ in $\rho(M)$ with the combined player A transition $\hat{s} \xrightarrow{\text{Cmb}} \sum_{i \in I} w_i \cdot \{[\hat{s}_i]\}$ in $\hat{G}$. This transition satisfies (iv) as the only possible player C transition is $\sum_{i \in I} w_i \cdot \{[\hat{s}_i]\} \to \hat{\lambda}$ and the game $\rho(M)$ can match this with $\Lambda \xrightarrow{\text{Cmb}} \lambda$.

Note that in our proof of (iii) and (iv) we did not actually make any assumptions on $\rho(M)$, $\hat{S}$, $\hat{I}$, $\hat{L}$ or $\hat{R}$. In fact, $\hat{T}$ is capable of simulating *any* player A transition from any game so long that $\alpha(S)$ is finite and $\mathscr{R}$ is right-total.

This significantly simplifies the proof of our second claim. Let $\hat{G}$ be an arbitrary game satisfying the criteria and let $\hat{G}'$ be any other game for which $\hat{G}' \sqsubseteq G$ via $\mathscr{R}$ which agrees on $\hat{S}$, $\hat{I}$, $\hat{L}$ and $\hat{R}$ with $\hat{G}$. Remaining to show is that $\hat{G} \sqsubseteq \hat{G}'$. If we consider the a relation $\mathscr{R}'$ which pairs every state of $\hat{G}$ with the identical state of $\hat{G}'$ then conditions (i) and (ii) of Definition 6.10 are trivially satisfied for every $\langle \hat{s}, \hat{s}' \rangle \in \mathscr{R}$ (as well as the necessary conditions on initial states). To see (iii) and (iv) of Definition 6.10 are satisfied for every $\langle \hat{s}, \hat{s}' \rangle \in \mathscr{R}$ observe that $\mathscr{R}'$ is also right-total and $\alpha(S)$ remains finite. □

A.2.4 Proof of Lemma 6.16

In this section we prove Lemma 6.16 on page 126 holds.

Lemma 6.16. *Let $M = \langle S, I, T, L, R\rangle$ be an MDP for which there is a bound $r \in \mathbb{R}$ s.t. $R(s) \leq r$ for all $s \in S$. Moreover, let $\hat{S}'$ be an abstract state space, let $\alpha : S \to \hat{S}'$ be an abstraction function and let $\mathscr{R} \subseteq \alpha(S) \times S$ be the relation defined as $\{\langle \hat{s}, s\rangle \in \alpha(S) \times S \mid \alpha(s) = \hat{s}\}$. A game $\hat{G} = \langle \alpha(S), \hat{I}, \hat{T}, \hat{L}, \hat{R}\rangle$ exists for which $\hat{G} \sqsubseteq \rho(M)$ with $\mathscr{R}$ and*

- *$\hat{T}(\hat{s}) = \{\{\alpha(\lambda) \mid \lambda \in T(s)\} \mid s \in \alpha^{-1}(\hat{s})\}$ for all $\hat{s} \in \alpha(S)$.*

Moreover, for any such $\hat{G}$ we have that any other game $\hat{G}' = \langle \alpha(S), \hat{I}, \hat{T}', \hat{L}, \hat{R}\rangle$ such that $\hat{G}' \sqsubseteq \rho(M)$ with $\mathscr{R}$ and such that $\hat{G}'$ agrees with $\hat{G}$ on $\hat{I}, \hat{L}$ and $\hat{R}$, abstracts $\hat{G}$.

Proof. Our first claim regarding the existence of a suitable $\hat{G}$ is immediately satisfied by observing that $\alpha(M)$ (as defined by Definition 3.27) satisfies all criteria and, by Lemma 6.12, is such that $\alpha(M) \sqsubseteq \rho(M)$.

For our second claim we take an arbitrary $\hat{G}' = \langle \alpha(S), \hat{I}, \hat{T}', \hat{L}, \hat{R}\rangle$ such that $\hat{G}' \sqsubseteq \rho(M)$ via $\mathscr{R}$ and such that $\hat{G}'$ agrees with $\hat{G}$ on $\hat{I}, \hat{L}$ and $\hat{R}$. In this proof, we will use the fact that $\mathscr{R}$ is left-unique and hence, by Lemma 3.2 item (ii), $\mathcal{L}(\mathscr{R})$ is left-unique, also.

Consider the relation $\mathscr{R}'$ on $\hat{G}' \uplus \hat{G}$ which pairs every state of $\hat{G}'$ with the identical state in $\hat{G}$. As $\hat{G}'$ and $\hat{G}$ agree on $\hat{I}, \hat{L}$ and $\hat{R}$, conditions (i) and (ii) of Definition 6.10 are trivially satisfied for every $\langle \hat{s}', \hat{s}\rangle \in \mathscr{R}'$ (as well as the necessary conditions on initial states). Remaining to show is that (iii) and (iv) of Definition 6.10 are satisfied for every $\langle \hat{s}', \hat{s}\rangle \in \mathscr{R}'$.

Let us first look at condition (iii). It is sufficient to show that for every player A transition $\hat{s} \to \{\alpha(\lambda) \mid \lambda \in T(s)\}$ of $\hat{G}$, induced by some $s \in \alpha^{-1}(\hat{s})$, there exist a combined player A transitions from $\hat{s}'$ in $\hat{G}'$ that satisfies condition (iii). As $\hat{G}' \sqsubseteq \rho(M)$ via $\mathscr{R}$ and $\langle \hat{s}, s\rangle \in \mathscr{R}$ there must be some combined player A transition $\hat{s}' \xrightarrow{\text{Cmb}} \hat{\Lambda}$ in $\hat{G}'$ to match the player A transition $s \to T(s)$ in $\rho(M)$. This means that for every player C transition $T(s) \to \lambda$ in $\rho(M)$ there is a combined player C transition $\hat{\Lambda} \xrightarrow{\text{Cmb}} \hat{\lambda}$ in $\hat{G}'$ such that $\langle \hat{\lambda}, \lambda\rangle \in \mathcal{L}(\mathscr{R})$. Because of the left-uniqueness of $\mathcal{L}(\mathscr{R})$ this actually is the

transition $\hat{\Lambda} \xrightarrow{\text{Cmb}} \alpha(\lambda)$. Hence, for every transition $T(s) \to \lambda$ in $\rho(M)$ we have a transition $\hat{\Lambda} \xrightarrow{\text{Cmb}} \alpha(\lambda)$ in $\hat{G}'$.

Hence, if we match the player A transition $\hat{s} \to \{\alpha(\lambda) \mid \lambda \in T(s)\}$ in $\hat{G}$ with the player A transition $\hat{s}' \xrightarrow{\text{Cmb}} \hat{\Lambda}$ in $\hat{G}'$ then we know that for every player C transition $\{\alpha(\lambda) \mid \lambda \in T(s)\} \to \alpha(\lambda)$ of $\hat{G}$ there is a matching combined a player C transition $\hat{\Lambda} \xrightarrow{\text{Cmb}} \alpha(\lambda)$ in $\hat{G}'$. Evidently, $\langle \alpha(\lambda), \alpha(\lambda) \rangle \in \mathcal{L}(\mathscr{R}')$, meaning that (iii) of Definition 6.10 is satisfied for $\langle \hat{s}', \hat{s} \rangle \in \mathscr{R}'$.

Analogously, for condition (iv) of Definition 6.10, it must be that some combined player A transition $\hat{s}' \xrightarrow{\text{Cmb}} \hat{\Lambda}$ of $\hat{G}'$ matches the player A transition $s \to T(s)$ in $\rho(M)$ for condition (iv). That is, for every $\hat{\Lambda} \to \hat{\lambda}$ in $\hat{G}'$ there is a $T(s) \xrightarrow{\text{Cmb}} \lambda$ in $\rho(M)$ such that $\langle \hat{\lambda}, \lambda \rangle \in \mathcal{L}(\mathscr{R})$ and, because of left-uniqueness, $\hat{\lambda} = \alpha(\lambda)$.

Hence, if we match the player A transition $\hat{s} \to \{\alpha(\lambda) \mid \lambda \in T(s)\}$ in $\hat{G}$ with the player A transition $\hat{s}' \xrightarrow{\text{Cmb}} \hat{\Lambda}$ in $\hat{G}'$ then we know that for every player C transition $\hat{\Lambda} \to \hat{\lambda}$ in $\hat{G}'$ there is a distribution $\sum_{i \in I} w_i \cdot [\lambda_i] \in \mathbb{D}(T(s))$ such that $\hat{\lambda} = \alpha(\sum_{i \in I} w_i \cdot \lambda_i)$. It is not difficult to see that $\hat{\lambda} = \alpha(\sum_{i \in I} w_i \cdot \lambda_i) = \sum_{i \in I} w_i \cdot \alpha(\lambda_i)$ and hence there is a transition $\{\alpha(\lambda) \mid \lambda \in T(s)\} \xrightarrow{\text{Cmb}} \hat{\lambda}$ also. That is, for every $\hat{\Lambda} \to \hat{\lambda}$ in $\hat{G}'$ there is a transition $\{\alpha(\lambda) \mid \lambda \in T(s)\} \xrightarrow{\text{Cmb}} \hat{\lambda}$ in $\hat{G}$. This means condition (iv) of Definition 6.10 is also satisfied for $\langle \hat{s}', \hat{s} \rangle \in \mathscr{R}'$.

□

A.2.5 Proof of Theorem 6.18

In this section we set out to prove Theorem 6.18 on page 126, which we first recall:

Theorem 6.18. *Let $\hat{G}$ and G be games such that $\hat{G} \sqsubseteq G$ and $\hat{G}$ and G are finitely branching for player A. We have that*

$$\mathit{Prob}^{-}(\hat{G}) \leq \mathit{Prob}^{-}(G) \quad \text{and} \quad \mathit{Prob}^{+}(\hat{G}) \leq \mathit{Prob}^{+}(G) \,.$$

To prove Theorem 6.18 we will use abstract interpretation methods in a similar fashion to [WZ10], using the lattices in Lemma 3.10 and Corollary 3.11 and the fixpoint characterisation of game properties in Section 3.4.2. Our main problem in doing this is that the functions we use in the fixpoint characterisation take infima and suprema over normal transitions whereas our abstraction preorder uses combined transitions. This is mostly an issue for player A transitions because here we use a player C state (an element of $\overline{\mathbb{P}}\mathbb{D}S$) to represent a probabilistic combination of player C states (an element of $\mathbb{D}\overline{\mathbb{P}}\mathbb{D}S$).

Therefore, before we prove Theorem 6.18, we first prove that certain infima and suprema over combined transitions coincide with infima and suprema over normal transitions. We first show that, if we are allowed to take any weighted combination of values from a function, $v : S \to [0, \infty]$, it never pays to use non-trivial weights:

Lemma A.6. *Let S be a countable set and let $v : S \to [0, \infty]$ be some mapping from S to the non-negative reals extended with positive infinity. We have:*

$$\inf_{s' \in S} v(s') = \inf_{\lambda \in \mathbb{D}S} (\textstyle\sum_{s' \in \text{SUPP}(\lambda)} v(s') \cdot \lambda(s')) \quad \textit{and} \tag{A.8}$$

$$\sup_{s' \in S} v(s') = \sup_{\lambda \in \mathbb{D}S} (\textstyle\sum_{s' \in \text{SUPP}(\lambda)} v(s') \cdot \lambda(s')) \tag{A.9}$$

Proof. The inequality $\geq$ for (A.8) and $\leq$ for (A.9) follow from the fact that for any $s' \in S$ we have that $[s] \in \mathbb{D}S$ and the fact that $v(s') = (\sum_{s' \in \text{SUPP}([s])} v(s') \cdot [s](s'))$. It is remaining to show $\leq$ for (A.8) and $\geq$ for (A.9).

We first show $\leq$ for (A.8). If $\inf_{s' \in S} v(s') = \infty$ then it must be the case that $v(s') = \infty$ for every $s' \in S$. In this case, the sum $(\sum_{s' \in S} v(s') \cdot \lambda(s'))$ must be ∞ for *every* distribution $\lambda \in \mathbb{D}S$. Hence, $\inf_{\lambda \in \mathbb{D}S}(\sum_{s' \in S} v(s') \cdot \lambda(s')) = \infty$ and the inequality is preserved. We now consider the case when $\inf_{s' \in S} v(s') = r$ for some $r \in [0, \infty[$. To see $r \leq \inf_{\lambda \in \mathbb{D}S}(\sum_{s' \in S} v(s') \cdot \lambda(s'))$ observe that for every $\lambda \in \mathbb{D}S$ we have

$$r = (\textstyle\sum_{s' \in S} r \cdot \lambda(s')) \leq (\sum_{s' \in S} v(s') \cdot \lambda(s'))$$

because $\sum_{s' \in S} \lambda(s') = 1$ and $r \leq v(s')$ for every $s' \in S$.

It remains to show $\geq$ for (A.9). We first consider the case when $\sup_{\lambda \in \mathbb{D}S}(\sum_{s' \in S} v(s') \cdot$

$\lambda(s')) = \infty$. We will show that $\sup_{s\in S} v(s) = \infty$ by contradiction. Suppose $\sup_{s\in S} v(s) = r$ for some $r \in [0, \infty[$. Then for any $\lambda \in \mathbb{D}S$ we get we must have

$$(\textstyle\sum_{s'\in S} v(s') \cdot \lambda(s')) \leq (\sum_{s'\in S} r \cdot \lambda(s')) = r \ .$$

As for arbitrary $\lambda \in \mathbb{D}S$ the sum is bounded by r we arrive at the contradiction that $\sup_{\lambda\in\mathbb{D}S}(\sum_{s'\in S} v(s') \cdot \lambda(s')) \leq r$. Hence, it must be the case that $\sup_{s\in S} v(s) = \infty$.

Remaining to consider is the case when $\sup_{\lambda\in\mathbb{D}S}(\sum_{s'\in S} v(s') \cdot \lambda(s')) = r$ for for some $r \in [0, \infty[$. To show $\sup_{s'\in S} v(s') \geq r$, by the definition of suprema, it is sufficient to show that for every $\epsilon > 0$ we have $v(s') > r - \epsilon$ for some $s' \in S$. We again prove this by contradiction. Suppose $v(s') \leq r - \epsilon$ for all $s' \in S$. Then

$$(\textstyle\sum_{s'\in S} v(s') \cdot \lambda(s')) \leq (\sum_{s'\in S} (r - \epsilon) \cdot \lambda(s')) = (r - \epsilon) \ .$$

This is again a contradiction. □

We now extend Lemma A.6 to include quantifications over transitions. We start by extending it with a quantification over player C transitions. Then we will consider quantifications over both player A and player C transitions:

Lemma A.7. *Let $G = \langle S, I, T, L, R\rangle$ be game and let $v : S \to [0, \infty]$ be an arbitrary mapping from states of G to the non-negative real numbers extended with positive infinity. For every player C state $\Lambda \in \overline{\mathbb{P}}\mathbb{D}S$ we have*

$$\inf_{\Lambda \xrightarrow{Cmb} \lambda} \sum_{s'\in \textsc{Supp}(\lambda)} (v(s') \cdot \lambda(s')) = \inf_{\Lambda\to\lambda} \sum_{s'\in \textsc{Supp}(\lambda)} (v(s') \cdot \lambda(s')) \quad \textit{and}$$

$$\sup_{\Lambda \xrightarrow{Cmb} \lambda} \sum_{s'\in \textsc{Supp}(\lambda)} (v(s') \cdot \lambda(s')) = \sup_{\Lambda\to\lambda} \sum_{s'\in \textsc{Supp}(\lambda)} (v(s') \cdot \lambda(s'))$$

Proof. We first focus on showing the equality involving infima. Let us define a function, $\overline{v} : \Lambda \to [0, \infty]$, which yields

$$\overline{v}(\lambda) = \sum_{s'\in \textsc{Supp}(\lambda)} (v(s') \cdot \lambda(s'))$$

for every $\lambda \in \Lambda$. Using Lemma A.6 we have the following:

$$\begin{aligned}
\inf_{\Lambda \to \lambda} \sum_{s' \in \text{SUPP}(\lambda)} (v(s') \cdot \lambda(s')) &= \inf_{\lambda \in \Lambda} \ \overline{v}(\lambda) && \text{(Definition of } \overline{v}\text{)} \\
&= \inf_{\overline{\lambda} \in \mathbb{D}\Lambda} \sum_{\lambda \in \text{SUPP}(\overline{\lambda})} (\overline{\lambda}(\lambda) \cdot \overline{v}(\lambda)) && \text{(Lemma A.6)}
\end{aligned}$$

Given this equality, to show the the infimum over combined transitions equals the infimum over normal transitions, it is sufficient to show that:

$$\inf_{\Lambda \xrightarrow{\text{Cmb}} \lambda} \sum_{s' \in \text{SUPP}(\lambda)} (v(s') \cdot \lambda(s')) = \inf_{\overline{\lambda} \in \mathbb{D}\Lambda} \sum_{\lambda \in \text{SUPP}(\overline{\lambda})} (\overline{\lambda}(\lambda) \cdot \overline{v}(\lambda)) \ . \tag{A.10}$$

We first show that the inequality $\geq$ holds for (A.10) by taking an arbitrary combined transition $\Lambda \xrightarrow{\text{Cmb}} \lambda$ and showing that the sum induced by λ on the left-hand side is equal to the sum induced by some distribution $\overline{\lambda} \in \mathbb{D}\Lambda$ on the right-hand side.

We take an arbitrary combined transition $\Lambda \xrightarrow{\text{Cmb}} \lambda$. We let $\{w_i\}_{i \in I}$ be the family of weights and $\{\lambda_i\}_{i \in I}$ be the family of distributions in Λ such that $\lambda = \sum_{i \in I} w_i \cdot \lambda_i$. We pick these families such that $w_i > 0$ for every $i \in I$. For convenience, let us write $I(s')$ to denote the index set $\{i \in I \mid s' \in \text{SUPP}(\lambda_i)\}$ for arbitrary for arbitrary $s' \in S$. For $\overline{\lambda} = \sum_{i \in I} w_i \cdot [\lambda_i]$ we have the following equality:

$$\begin{aligned}
\sum_{s' \in \text{SUPP}(\lambda)} (v(s') \cdot \lambda(s')) &= \sum_{s' \in \text{SUPP}(\lambda)} \left(v(s') \cdot \left(\left(\sum_{i \in I} w_i \cdot \lambda_i \right)(s') \right) \right) && \text{(Definition of } \lambda\text{)} \\
&= \sum_{s' \in \text{SUPP}(\lambda)} \left(v(s') \cdot \left(\sum_{i \in I} w_i \cdot \lambda_i(s') \right) \right) && \text{(Rewriting)} \\
&= \sum_{s' \in \text{SUPP}(\lambda)} \left(v(s') \cdot \left(\sum_{i \in I(s')} w_i \cdot \lambda_i(s') \right) \right) && \text{(Definition of } I(s')\text{)} \\
&= \sum_{s' \in \text{SUPP}(\lambda)} \sum_{i \in I(s')} (v(s') \cdot w_i \cdot \lambda_i(s')) && \text{(Rearranging)} \\
&= \sum_{i \in I} \sum_{s' \in \text{SUPP}(\lambda_i)} (w_i \cdot (v(s') \cdot \lambda_i(s'))) && \text{(Rearranging)}
\end{aligned}$$

$$
\begin{aligned}
&= \sum_{i \in I} \left(w_i \cdot \left(\sum_{s' \in \text{SUPP}(\lambda_i)} (v(s') \cdot \lambda_i(s')) \right) \right) && \text{(Rearranging)} \\
&= \sum_{i \in I} (w_i \cdot \overline{v}(\lambda_i)) && \text{(Definition of } \overline{v}\text{)} \\
&= \sum_{\lambda \in \text{SUPP}(\overline{\lambda})} (\overline{\lambda}(\lambda) \cdot \overline{v}(\lambda)) && \text{(Definition of } \overline{\lambda}\text{)}
\end{aligned}
$$

This means that $\geq$ holds for (A.10). To see that $\leq$ holds also consider that there is a combined transition from Λ matching every $\overline{\lambda} \in \mathbb{D}\Lambda$ for which the above equality holds. The equality involving suprema follows analogously. □

We finally extend Lemma A.6 and Lemma A.7 to include player A transitions:

Lemma A.8. *Let $G = \langle S, I, T, L, R \rangle$ be game that is finitely branching for player A and let $v : S \to [0, \infty]$ be an arbitrary mapping from states of G to the non-negative real numbers extended with positive infinity. For every state $s \in S$ we have*

$$
\begin{aligned}
&\inf_{s \xrightarrow{Cmb} \Lambda} \inf_{\Lambda \xrightarrow{Cmb} \lambda} \sum_{s' \in \text{SUPP}(\lambda)} (v(s') \cdot \lambda(s')) = \inf_{s \to \Lambda} \inf_{\Lambda \to \lambda} \sum_{s' \in \text{SUPP}(\lambda)} (v(s') \cdot \lambda(s')) \quad \textit{and} \\
&\inf_{s \xrightarrow{Cmb} \Lambda} \sup_{\Lambda \xrightarrow{Cmb} \lambda} \sum_{s' \in \text{SUPP}(\lambda)} (v(s') \cdot \lambda(s')) = \inf_{s \to \Lambda} \sup_{\Lambda \to \lambda} \sum_{s' \in \text{SUPP}(\lambda)} (v(s') \cdot \lambda(s')) \quad \textit{and} \\
&\sup_{s \xrightarrow{Cmb} \Lambda} \inf_{\Lambda \xrightarrow{Cmb} \lambda} \sum_{s' \in \text{SUPP}(\lambda)} (v(s') \cdot \lambda(s')) = \sup_{s \to \Lambda} \inf_{\Lambda \to \lambda} \sum_{s' \in \text{SUPP}(\lambda)} (v(s') \cdot \lambda(s')) \quad \textit{and} \\
&\sup_{s \xrightarrow{Cmb} \Lambda} \sup_{\Lambda \xrightarrow{Cmb} \lambda} \sum_{s' \in \text{SUPP}(\lambda)} (v(s') \cdot \lambda(s')) = \sup_{s \to \Lambda} \sup_{\Lambda \to \lambda} \sum_{s' \in \text{SUPP}(\lambda)} (v(s') \cdot \lambda(s')) \,.
\end{aligned}
$$

Proof. We first focus on showing the equality

$$
\inf_{s \xrightarrow{\text{Cmb}} \Lambda} \sup_{\Lambda \xrightarrow{\text{Cmb}} \lambda} \sum_{s' \in \text{SUPP}(\lambda)} (v(s') \cdot \lambda(s')) = \inf_{s \to \Lambda} \sup_{\Lambda \to \lambda} \sum_{s' \in \text{SUPP}(\lambda)} (v(s') \cdot \lambda(s')) \,. \tag{A.11}
$$

Note that, by Lemma A.7, we can replace the supremum over combined player C transitions on the left-hand side with a supremum over normal player C transitions. Hence, it

is remaining to show that

$$\inf_{s \xrightarrow{\text{Cmb}} \Lambda} \sup_{\Lambda \to \lambda} \sum_{s' \in \text{SUPP}(\lambda)} (v(s') \cdot \lambda(s')) = \inf_{s \to \Lambda} \sup_{\Lambda \to \lambda} \sum_{s' \in \text{SUPP}(\lambda)} (v(s') \cdot \lambda(s')) \,. \tag{A.12}$$

Let us define a function, $\overline{v} : T(s) \to [0, \infty]$, which yields

$$\overline{v}(\Lambda) = \sup_{\Lambda \to \lambda} \sum_{s' \in \text{SUPP}(\lambda)} (v(s') \cdot \lambda(s'))$$

for every $\Lambda \in T(s)$.

Using Lemma A.6 we have the following:

$$\begin{aligned} \inf_{s \to \Lambda} \left(\sup_{\Lambda \to \lambda} \sum_{s' \in \text{SUPP}(\lambda)} (v(s') \cdot \lambda(s')) \right) &= \inf_{\Lambda \in T(s)} \overline{v}(\Lambda) && \text{(Definition of } \overline{v}) \\ &= \inf_{\overline{\Lambda} \in \mathbb{D}(T(s))} \left(\sum_{\Lambda \in \text{SUPP}(\overline{\Lambda})} (\overline{\Lambda}(\Lambda) \cdot \overline{v}(\Lambda)) \right) && \text{(Lemma A.6)} \end{aligned}$$

Given this equality, to show the the infimum over combined player A transitions equals the infimum over normal player A transitions, it is sufficient to show that:

$$\inf_{s \xrightarrow{\text{Cmb}} \Lambda} \left(\sup_{\Lambda \to \lambda} \sum_{s' \in \text{SUPP}(\lambda)} (v(s') \cdot \lambda(s')) \right) = \inf_{\overline{\Lambda} \in \mathbb{D}(T(s))} \sum_{\Lambda \in \text{SUPP}(\overline{\Lambda})} (\overline{\Lambda}(\Lambda) \cdot \overline{v}(\Lambda)) \,. \tag{A.13}$$

We first show that the inequality $\geq$ holds for (A.13) by taking an arbitrary combined transition $s \xrightarrow{\text{Cmb}} \Lambda$ and showing that the value for Λ on the left-hand side is equal to the value induced by some distribution on player C states $\overline{\Lambda} \in \mathbb{D}(T(s))$ on the right-hand side.

We take an arbitrary combined transition $s \xrightarrow{\text{Cmb}} \Lambda$. We let $\{w_i\}_{i \in I}$ be the finite family of weights and $\{\Lambda_i\}_{i \in I}$ be the finite family of player C states such that $\Lambda = \sum_{i \in I} w_i \cdot \Lambda_i$. We can take these families such that $w_i > 0$ for every $i \in I$. Let *Fam* be the set of all families $\{\lambda_i\}_{i \in I}$ such that $\lambda_i \in \Lambda_i$ for all $i \in I$. For convenience, for some $\{\lambda_i\}_{i \in I}$ and state $s' \in S$ let us write $I(\{\lambda_i\}_{i \in I}, s')$ to denote the index set $\{i \in I \mid s' \in \text{SUPP}(\lambda_i)\}$. For

$\overline{\Lambda} = \sum_{i\in I} w_i \cdot [\Lambda_i]$ we have the following equality:

$$
\begin{aligned}
\sup_{\Lambda\to\lambda} \sum_{s'\in \textsc{Supp}(\lambda)} (v(s')\cdot\lambda(s')) &\overset{(1)}{=} \sup_{\{\lambda_j\}_{j\in I}\in Fam} \sum_{s'\in\textsc{Supp}(\sum_{k\in I} w_k\cdot\lambda_k)} \left(v(s')\cdot\left(\sum_{i\in I} w_i\cdot\lambda_i\right)(s')\right)\\
&= \sup_{\{\lambda_j\}_{j\in I}\in Fam} \sum_{s'\in\textsc{Supp}(\sum_{k\in I} w_k\cdot\lambda_k)} \left(v(s')\cdot\left(\sum_{i\in I} w_i\cdot\lambda_i(s')\right)\right)\\
&\overset{(2)}{=} \sup_{\{\lambda_j\}_{j\in I}\in Fam} \left(\sum_{s'\in\textsc{Supp}(\sum_{k\in I} w_k\cdot\lambda_k)} \sum_{i\in I(\{\lambda_k\}_{k\in I},s')} (w_i\cdot(v(s')\cdot(\lambda_i(s'))))\right)\\
&\overset{(3)}{=} \sup_{\{\lambda_j\}_{j\in I}\in Fam} \left(\sum_{i\in I}\sum_{s'\in\textsc{Supp}(\lambda_i)} (w_i\cdot(v(s')\cdot(\lambda_i(s'))))\right)\\
&\overset{(4)}{=} \sum_{i\in I}\left(\sup_{\Lambda_i\to\lambda}\left(\sum_{s'\in\textsc{Supp}(\lambda)} (w_i\cdot(v(s')\cdot\lambda(s')))\right)\right)\\
&= \sum_{i\in I}\left(w_i\cdot\left(\sup_{\Lambda_i\to\lambda}\sum_{s'\in\textsc{Supp}(\lambda)} (v(s')\cdot\lambda(s'))\right)\right)\\
&\overset{(5)}{=} \sum_{i\in I}(w_i\cdot\overline{v}(\Lambda_i))\\
&\overset{(6)}{=} \sum_{\Lambda\in\textsc{Supp}(\overline{\Lambda})} (\overline{\Lambda}(\Lambda)\cdot\overline{v}(\Lambda))
\end{aligned}
$$

Equality (1) uses the fact that every normal player C transition from the player C state $\sum_{i\in I} w_i\cdot\Lambda_i$ is of the form $\sum_{i\in I} w_i\cdot\Lambda_i \to \sum_{i\in I} w_i\cdot\lambda_i$ for some family of distributions $\{\lambda_i\}_{i\in I}$ with $\lambda_i\in\Lambda_i$ for every $i\in I$. In (2) we use the definition of $I(\{\lambda_k\}_{k\in I}, s')$. Equality (3) rearranges the terms of the summand. For (4), observe that we take a supremum over a set which contains for each family $\{\lambda_i\}_{i\in I}$ a sum over $i\in I$ where each summand is a value that depends only λ_i and not other members of the family. As we can choose each member of the family independently, this is equivalent to taking the sum over over all $i\in I$ and where each summand is a supremum over distributions $\lambda\in\Lambda_i$. Equality (5) and (6) are due to the definition of $\overline{v}$ and $\overline{\lambda}$, respectively.

Because of this equality we have that $\geq$ holds for (A.13). To see that $\leq$ holds also consider that, for games that are finitely branching for player A, there is a combined

player A transition from s matching every $\overline{\Lambda} \in \mathbb{D}(T(s))$.

Analogous arguments show the remaining equalities also hold. □

We are now finally in a position to focus on Theorem 6.18. We will use abstract interpretation techniques to prove this theorem. Before we prove this theorem, however, we first prove that the fixpoint characterisations on games $\hat{G}, G$ such that $\hat{G} \sqsubseteq G$ are related via the lattices defined in Lemma 3.10 and Corollary 3.11.

Lemma A.9. *Let $G = \langle S, I, T, L, R\rangle$ and $\hat{G} = \langle \hat{S}, \hat{I}, \hat{T}, \hat{L}, \hat{R}\rangle$ be games that are finitely branching for player A and suppose $\hat{G} \sqsubseteq G$ via a strong probabilistic game simulation $\mathscr{R} \subseteq \hat{S} \times S$. Let*

$$\langle S \to [0,1], \leq\rangle \underset{\alpha^+}{\overset{\gamma^+}{\leftrightarrows}} \langle \hat{S} \to [0,1], \leq\rangle \quad \textit{and} \quad \langle \hat{S} \to [0,1], \leq\rangle \underset{\gamma^-}{\overset{\alpha^-}{\leftrightarrows}} \langle S \to [0,1], \leq\rangle$$

be the lattices as discussed in Lemma 3.10 and Corollary 3.11 for $\mathscr{R}$. Moreover, let

$$\mathrm{P}_G^{--}, \mathrm{P}_G^{+-}, \mathrm{P}_G^{-+}, \mathrm{P}_G^{++} : (S \to [0,1]) \to (S \to [0,1]) \textit{ and}$$
$$\mathrm{P}_{\hat{G}}^{--}, \mathrm{P}_{\hat{G}}^{+-}, \mathrm{P}_{\hat{G}}^{-+}, \mathrm{P}_{\hat{G}}^{++} : (\hat{S} \to [0,1]) \to (\hat{S} \to [0,1])$$

be the functions as defined in Section 3.4.2 for G and $\hat{G}$. We have:

$$(\gamma^- \circ \mathrm{P}_{\hat{G}}^{--} \circ \alpha^-) \leq \mathrm{P}_G^{--} \quad \textit{and} \quad (\alpha^+ \circ \mathrm{P}_G^{+-} \circ \gamma^+) \leq \mathrm{P}_{\hat{G}}^{+-} \quad \textit{and} \tag{A.14}$$
$$(\gamma^- \circ \mathrm{P}_{\hat{G}}^{-+} \circ \alpha^-) \leq \mathrm{P}_G^{-+} \quad \textit{and} \quad (\alpha^+ \circ \mathrm{P}_G^{++} \circ \gamma^+) \leq \mathrm{P}_{\hat{G}}^{++} \tag{A.15}$$

Proof. Let us first consider inequalities in (A.14) and (A.15). Following the definition of $\leq$, it is sufficient to show that, for every valuation $\hat{v} : \hat{S} \to [0,1]$ and $v : S \to [0,1]$ and for every $\hat{s} \in \hat{S}$ and $s \in S$, the following inequalities hold:

$$\gamma^-(\mathrm{P}_{\hat{G}}^{--}(\alpha^-(v)))(s) \leq \mathrm{P}_G^{--}(v)(s) \quad \text{and} \quad \alpha^+(\mathrm{P}_G^{+-}(\gamma^+(\hat{v})))(s) \leq \mathrm{P}_{\hat{G}}^{+-}(\hat{v})(\hat{s}) \text{ and}$$
$$\gamma^-(\mathrm{P}_{\hat{G}}^{-+}(\alpha^-(v)))(s) \leq \mathrm{P}_G^{-+}(v)(s) \quad \text{and} \quad \alpha^+(\mathrm{P}_G^{++}(\gamma^+(\hat{v})))(s) \leq \mathrm{P}_{\hat{G}}^{++}(\hat{v})(\hat{s})$$

To show these inequalities hold, following the definition of γ^-, α^+, it is sufficient to show

that for every $\langle \hat{s}, s \rangle \in \mathscr{R}$ we have:

$$\mathrm{P}^{--}_{\hat{G}}(\alpha^-(v))(\hat{s}) \leq \mathrm{P}^{--}_{G}(v)(s) \quad \text{and} \quad \mathrm{P}^{+-}_{G}(\gamma^+(\hat{v}))(s) \leq \mathrm{P}^{+-}_{\hat{G}}(\hat{v})(\hat{s}) \quad \text{and} \tag{A.16}$$

$$\mathrm{P}^{-+}_{\hat{G}}(\alpha^-(v))(\hat{s}) \leq \mathrm{P}^{-+}_{G}(v)(s) \quad \text{and} \quad \mathrm{P}^{++}_{G}(\gamma^+(\hat{v}))(s) \leq \mathrm{P}^{++}_{\hat{G}}(\hat{v})(\hat{s}) \tag{A.17}$$

Because $\mathscr{R}$ is a strong probabilistic game simulation we have that $\hat{L}(\hat{s}, \mathbf{F}) \leq L(s, \mathbf{F})$. Hence, by definition of $\leq$ in Definition 3.8 we have

$$\mathrm{LB}(\hat{L}(\hat{s}, \mathbf{F})) \Rightarrow \mathrm{LB}(L(s, \mathbf{F})) \quad \text{and} \quad \mathrm{UB}(\hat{L}(\hat{s}, \mathbf{F})) \Leftarrow \mathrm{UB}(L(s, \mathbf{F})) \,.$$

This means that if the left-hand side of the inequalities in (A.16) is 1 due to the propositional labelling, then so is the right-hand side. Hence, by the definition of P^{--}_{G}, $\mathrm{P}^{+-}_{\hat{G}}$, P^{-+}_{G}, P^{++}_{G} it remains to show that

$$\inf_{\hat{s}\to\hat{\Lambda}} \inf_{\hat{\Lambda}\to\hat{\lambda}} \textstyle\sum_{\hat{s}'\in \mathrm{SUPP}(\hat{\lambda})}(\alpha^-(v)(\hat{s}') \cdot \hat{\lambda}(\hat{s}')) \leq \inf_{s\to\Lambda} \inf_{\Lambda\to\lambda} \textstyle\sum_{s'\in \mathrm{SUPP}(\lambda)}(v(s') \cdot \lambda(s')) \quad \text{and} \tag{A.18}$$

$$\sup_{s\to\Lambda} \inf_{\Lambda\to\lambda} \textstyle\sum_{s'\in \mathrm{SUPP}(\lambda)}(\gamma^+(\hat{v})(s') \cdot \lambda(s')) \leq \sup_{\hat{s}\to\hat{\Lambda}} \inf_{\hat{\Lambda}\to\hat{\lambda}} \textstyle\sum_{\hat{s}'\in \mathrm{SUPP}(\hat{\lambda})}(\hat{v}(\hat{s}') \cdot \hat{\lambda}(\hat{s}')) \quad \text{and} \tag{A.19}$$

$$\inf_{\hat{s}\to\hat{\Lambda}} \sup_{\hat{\Lambda}\to\hat{\lambda}} \textstyle\sum_{\hat{s}'\in \mathrm{SUPP}(\hat{\lambda})}(\alpha^-(v)(\hat{s}') \cdot \hat{\lambda}(\hat{s}')) \leq \inf_{s\to\Lambda} \sup_{\Lambda\to\lambda} \textstyle\sum_{s'\in \mathrm{SUPP}(\lambda)}(v(s') \cdot \lambda(s')) \quad \text{and} \tag{A.20}$$

$$\sup_{s\to\Lambda} \sup_{\Lambda\to\lambda} \textstyle\sum_{s'\in \mathrm{SUPP}(\lambda)}(\gamma^+(\hat{v})(s') \cdot \lambda(s')) \leq \sup_{\hat{s}\to\hat{\Lambda}} \sup_{\hat{\Lambda}\to\hat{\lambda}} \textstyle\sum_{\hat{s}'\in \mathrm{SUPP}(\hat{\lambda})}(\hat{v}(\hat{s}') \cdot \hat{\lambda}(\hat{s}')) \,. \tag{A.21}$$

To prove (A.18), we first show that, for all $\langle \hat{\lambda}, \lambda \rangle \in \mathcal{L}(\mathscr{R})$ we have

$$\sum_{\hat{s}'\in \mathrm{SUPP}(\hat{\lambda})} (\alpha^-(v)(\hat{s}') \cdot \hat{\lambda}(\hat{s}')) \leq \sum_{s'\in \mathrm{SUPP}(\lambda)} (v(s') \cdot \lambda(s')) \quad \text{and} \tag{A.22}$$

$$\sum_{s'\in \mathrm{SUPP}(\lambda)} (\gamma^-(\hat{v})(s') \cdot \lambda(s')) \leq \sum_{\hat{s}'\in \mathrm{SUPP}(\hat{\lambda})} (\hat{v}^+(\hat{s}') \cdot \hat{\lambda}(\hat{s}')) \,. \tag{A.23}$$

The inequality in (A.22) follows easily if we consider the weight function $\delta : \hat{S} \times S \to [0, 1]$

witnessing $\langle \hat{\lambda}, \lambda \rangle \in \mathcal{L}(\mathscr{R})$ (see Definition 3.1):

$$\begin{aligned}
\sum_{\hat{s} \in \textsc{Supp}(\hat{\lambda})} \alpha^-(v)(\hat{s}) \cdot \hat{\lambda}(\hat{s}) &\overset{(1)}{=} \sum_{\hat{s} \in \textsc{Supp}(\hat{\lambda})} \left(\alpha^-(v)(\hat{s}) \cdot \left(\sum_{s \in S} \delta(\hat{s}, s) \right) \right) \\
&= \sum_{\hat{s} \in \textsc{Supp}(\hat{\lambda})} \sum_{s \in \{s' \in S \mid \delta(\hat{s}, s') > 0\}} (\alpha^-(v)(\hat{s}) \cdot \delta(\hat{s}, s)) \\
&\overset{(2)}{=} \sum_{s \in \textsc{Supp}(\lambda)} \sum_{\hat{s} \in \{\hat{s}' \in \hat{S} \mid \delta(\hat{s}', s) > 0\}} (\alpha^-(v)(\hat{s}) \cdot \delta(\hat{s}, s)) \\
&\overset{(3)}{\leq} \sum_{s \in \textsc{Supp}(\lambda)} \sum_{\hat{s} \in \{\hat{s}' \in \hat{S} \mid \delta(\hat{s}', s) > 0\}} (v(s) \cdot \delta(\hat{s}, s)) \\
&= \sum_{s \in \textsc{Supp}(\lambda)} \left(v(\hat{s}) \cdot \left(\sum_{\hat{s} \in \hat{S}} \delta(\hat{s}, s) \right) \right) \\
&= \sum_{s \in \textsc{Supp}(\lambda)} v(s) \cdot \lambda(\hat{s})
\end{aligned}$$

The equality in (1) follows from the definition of weight functions. Then, the equality in (2) is a rearrangements of terms. To see this recall that, by definition of weight functions, $\delta(\hat{s}, s)$ is 0 whenever $s \notin \textsc{Supp}(\lambda)$ or $\hat{s} \notin \textsc{Supp}(\hat{\lambda})$. Finally, to see (3), by definition of δ, we have that whenever $\delta(\hat{s}, s) > 0$, we must have that $\langle \hat{s}, s \rangle \in \mathscr{R}$. As a result, by definition of α^-, it must be the case that $\alpha^-(v)(\hat{s}) \leq v(s)$. The inequality for (A.23) follows analogously.

Now recall that $\langle \hat{s}, s \rangle$ are in a strong probabilistic game simulation, and hence condition (iv) of Definition 6.10 hold for $\hat{s}$ and s. Following these conditions and the inequalities

in (A.22) and (A.23) we get:

$$\inf_{\hat{s}\xrightarrow{\text{Cmb}}\hat{\Lambda}}\inf_{\hat{\Lambda}\xrightarrow{\text{Cmb}}\hat{\lambda}}\sum_{\hat{s}'\in\text{SUPP}(\hat{\lambda})}(\alpha^-(v)(\hat{s}')\cdot\hat{\lambda}(\hat{s}'))\leq\inf_{s\to\Lambda}\inf_{\Lambda\to\lambda}\sum_{s'\in\text{SUPP}(\lambda)}(v(s')\cdot\lambda(s'))\quad\text{and}$$

$$\sup_{s\to\Lambda}\inf_{\Lambda\xrightarrow{\text{Cmb}}\lambda}\sum_{s'\in\text{SUPP}(\lambda)}(\gamma^+(\hat{v})(s')\cdot\lambda(s'))\leq\sup_{\hat{s}\xrightarrow{\text{Cmb}}\hat{\Lambda}}\inf_{\hat{\Lambda}\to\hat{\lambda}}\sum_{\hat{s}'\in\text{SUPP}(\hat{\lambda})}(\hat{v}(\hat{s}')\cdot\hat{\lambda}(\hat{s}'))\quad\text{and}$$

$$\inf_{\hat{s}\xrightarrow{\text{Cmb}}\hat{\Lambda}}\sup_{\hat{\Lambda}\to\hat{\lambda}}\sum_{\hat{s}'\in\text{SUPP}(\hat{\lambda})}(\alpha^-(v)(\hat{s}')\cdot\hat{\lambda}(\hat{s}'))\leq\inf_{s\to\Lambda}\sup_{\Lambda\xrightarrow{\text{Cmb}}\lambda}\sum_{s'\in\text{SUPP}(\lambda)}(v(s')\cdot\lambda(s'))\quad\text{and}$$

$$\sup_{s\to\Lambda}\sup_{\Lambda\to\lambda}\sum_{s'\in\text{SUPP}(\lambda)}(\gamma^+(\hat{v})(s')\cdot\lambda(s'))\leq\sup_{\hat{s}\xrightarrow{\text{Cmb}}\hat{\Lambda}}\sup_{\hat{\Lambda}\xrightarrow{\text{Cmb}}\hat{\lambda}}\sum_{\hat{s}'\in\text{SUPP}(\hat{\lambda})}(\hat{v}(\hat{s}')\cdot\hat{\lambda}(\hat{s}'))\,.$$

These inequalities are like the inequalities in (A.18) up to (A.21), except that they sometimes quantify over combined transitions. It follows directly from Lemma A.7 and Lemma A.8 these inqualities coincide for games that are finitely branching for player A.

□

We are now finally in a position to prove Theorem 6.18:

Theorem 6.18. *Let $\hat{G}$ and G be games such that $\hat{G}\sqsubseteq G$ and $\hat{G}$ and G are finitely branching for player A. We have that*

$$\mathit{Prob}^-(\hat{G})\leq\mathit{Prob}^-(G)\quad\text{and}\quad\mathit{Prob}^+(\hat{G})\leq\mathit{Prob}^+(G)\,.$$

Proof. Suppose $\hat{G}=\langle\hat{S},\hat{I},\hat{T},\hat{L},\hat{R}\rangle$ and $G=\langle S,I,T,L,R\rangle$ are games and suppose that $\mathscr{R}\subseteq\hat{S}\times S$ is the strong probabilistic game simulation such that $\hat{G}\sqsubseteq_{\mathscr{R}}G$.[2] Because $I\subseteq\mathscr{R}(\hat{I})$ and $\hat{I}\subseteq\mathscr{R}^{-1}(I)$, by Definition 3.23 and by the definition of $\leq$ in $[0,1]\times[0,1]$, the inequalities hold if for all $\langle\hat{s},s\rangle\in\mathscr{R}$ the following inequalities hold:

$$\mathit{Prob}^{--}(\hat{G},\hat{s})\leq\mathit{Prob}^{--}(G,s)\quad\text{and}\quad\mathit{Prob}^{+-}(G,s)\leq\mathit{Prob}^{+-}(\hat{G},\hat{s})\quad\text{and}\tag{A.24}$$

$$\mathit{Prob}^{-+}(\hat{G},\hat{s})\leq\mathit{Prob}^{-+}(G,s)\quad\text{and}\quad\mathit{Prob}^{++}(G,s)\leq\mathit{Prob}^{++}(\hat{G},\hat{s})\,.\tag{A.25}$$

[2]The existence of such a $\mathscr{R}$ was discussed in the proof of Lemma 6.14.

Using Lemma 3.10 and Corollary 3.11 we can rewrite (A.24) and (A.25) to:

$$(\mathrm{LFP}(\mathrm{P}^{--}_{\hat{G}}))(\hat{s}) \leq (\mathrm{LFP}(\mathrm{P}^{--}_{G}))(s) \text{ and } (\mathrm{LFP}(\mathrm{P}^{+-}_{G}))(s) \leq (\mathrm{LFP}(\mathrm{P}^{+-}_{\hat{G}}))(\hat{s}) \tag{A.26}$$

$$(\mathrm{LFP}(\mathrm{P}^{-+}_{\hat{G}}))(\hat{s}) \leq (\mathrm{LFP}(\mathrm{P}^{-+}_{G}))(s) \text{ and } (\mathrm{LFP}(\mathrm{P}^{++}_{G}))(s) \leq (\mathrm{LFP}(\mathrm{P}^{++}_{\hat{G}}))(\hat{s})\,. \tag{A.27}$$

Now suppose we consider these functions as functions on the lattices $\langle \hat{S} \to [0,1], \leq \rangle$ and $\langle S \to [0,1], \leq \rangle$. Considering definition of γ^-, γ^+ in Lemma 3.10 and Corollary 3.11 it is sufficient to show that

$$\gamma^-(\mathrm{LFP}(\mathrm{P}^{--}_{\hat{G}})) \leq \mathrm{LFP}(\mathrm{P}^{--}_{G}) \quad \text{and} \quad \mathrm{LFP}(\mathrm{P}^{+-}_{\hat{G}}) \leq \gamma^+(\mathrm{LFP}(\mathrm{P}^{+-}_{G})) \quad \text{and} \tag{A.28}$$

$$\gamma^-(\mathrm{LFP}(\mathrm{P}^{-+}_{\hat{G}})) \leq \mathrm{LFP}(\mathrm{P}^{-+}_{G}) \quad \text{and} \quad \mathrm{LFP}(\mathrm{P}^{++}_{\hat{G}}) \leq \gamma^+(\mathrm{LFP}(\mathrm{P}^{++}_{G}))\,. \tag{A.29}$$

The right-hand inequalities in (A.28) and (A.29) follow directly from Lemma A.9, 3.10 and 3.12. Similarly, the left-hand inequalities in (A.28) and (A.29) follow directly from Lemma A.9, Corollary 3.11 and Lemma 3.12. □

A.3 Proofs of Chapter 7

A.3.1 Proof of Theorem 7.8 (Model-level Soundness)

We will devote this section to proving Theorem 7.8 — i.e. we will show that model-level instrumentations satisfy the soundness requirement. However, first we require an ordering on paths of MDPs. We have already informally discussed that transition orderings induce an order over finite paths — we now formally define this ordering:

Definition A.10. *Let $M = \langle S, I, T, L, R \rangle$ be an MDP and let $\preceq_M$ be a transition ordering on M. We define the ordering $\leq_M \subseteq \mathit{FinPath}_M \times \mathit{FinPath}_M$ on paths of M as follows. For $\pi, \pi' \in \mathit{FinPath}_M$ we let $\pi <_M \pi'$ if and only if there exists an index $i \in \mathbb{N}$ such that all of the following conditions hold:*

(i) $i \leq |\pi|$,

(ii) $i < |\pi'|$,

(iii) $\forall k < i : \textsc{Trans}(\pi, k) = \textsc{Trans}(\pi', k)$ *and*

(iv) $i = |\pi|$ *or* $\textsc{Trans}(\pi, i) \prec_M \textsc{Trans}(\pi', i)$.

Definition A.10 is based on so called *dictionary orders* or *lexographical orders*. Essentially, if $\pi <_M \pi'$, it must either be that π is a prefix of π' or that π and π' share a common prefix and that $\textsc{Trans}(\pi, i) \prec_M \textsc{Trans}(\pi', i)$ for the smallest $i \in \mathbb{N}$ for which $\textsc{Trans}(\pi, i) \neq \textsc{Trans}(\pi', i)$. Before starting the main proof of Theorem 7.8 we first show that Definition A.10 satisfies some properties.

Lemma A.11. *Let $M = \langle S, I, T, L, R\rangle$ be an MDP. The relation $\leq_M$ is a partial order. Moreover, when restricted to paths of a fixed initial state $s_i \in I$ and a pure strategy $\sigma \in \mathit{PureStrat}_M$, then $\leq_M$ is total.*

Proof. Reflexivity of $\leq_M$ follows directly from its definition. Antisymmetry also follows directly from (ii), (iii) and (iv) of Definition A.10. To show $\leq_M$ is transitive let π_1, π_2, π_3 be arbitrary finite paths of M such that $\pi_1 <_M \pi_2$ and $\pi_2 <_M \pi_3$. We will show that $\pi_1 \leq_M \pi_3$ holds. If either $\pi_1 = \pi_2$ or $\pi_2 = \pi_3$, then this follows trivially. The remaining case to consider is when $\pi_1 <_M \pi_2$ and $\pi_2 <_M \pi_3$. In this case, let $i_1^2, i_2^3 \in \mathbb{N}$ be the indices for which $\pi_1 <_M \pi_2$ and $\pi_2 <_M \pi_3$ hold in Definition A.10, respectively. We will show that $\pi_1 <_M \pi_3$ via index $\min(i_1^2, i_2^3)$. Conditions (i) up to (iii) of Definition A.10 hold trivially. We will show that (iv) of Def. A.10 also holds for π_1, π_3 and index $\min(i_1^2, i_2^3)$ — i.e. that either $|\pi_1| = \min(i_1^2, i_2^3)$ or $\textsc{Trans}(\pi_1, \min(i_1^2, i_2^3)) \prec_M \textsc{Trans}(\pi_3, \min(i_1^2, i_2^3))$ — by case splitting on $i_1^2 = i_2^3$, $i_1^2 < i_2^3$ and $i_1^2 > i_2^3$.

If $i_1^2 = i_2^3$, then $i_1^2 = i_2^3 = \min(i_1^2, i_2^3)$. We know either $|\pi_1| = i_1^2$ holds or $\textsc{Trans}(\pi_1, i_1^2) \prec_M \textsc{Trans}(\pi_2, i_1^2)$ holds (using condition (iv) on $\pi_1 <_M \pi_2$). In the former case, we are done. In the latter case, as $|\pi_2| > i_1^2 = i_2^3$ (using condition (ii) on $\pi_2 <_M \pi_3$) we obtain $\textsc{Trans}(\pi_2, i_2^3) \prec_M \textsc{Trans}(\pi_3, i_2^3)$ (using condition (iv) on $\pi_2 <_M \pi_3$). Finally, we get that $\textsc{Trans}(\pi_1, \min(i_1^2, i_2^3)) \prec_M \textsc{Trans}(\pi_3, \min(i_1^2, i_2^3))$ by transitivity of $\prec_M$.

If $i_1^2 < i_2^3$, then we must have that $\textsc{Trans}(\pi_2, \min(i_1^2, i_2^3)) = \textsc{Trans}(\pi_3, \min(i_1^2, i_2^3))$ (using condition (iii) on $\pi_2 <_M \pi_3$). That our premise holds follows directly from condition (iv) $\pi_1 <_M \pi_2$ with this equality and from the fact that $i_1^2 = \min(i_1^2, i_2^3)$.

Finally, if $i_1^2 > i_2^3$, we have $i_2^3 = \min(i_1^2, i_2^3)$ and, because $|\pi_2| \neq i_2^3$ (using condition (ii) on $\pi_2 <_M \pi_3$), we must have ($\textsc{Trans}(\pi_1, \min(i_1^2, i_2^3)) =$) $\textsc{Trans}(\pi_2, \min(i_1^2, i_2^3)) \prec_M \textsc{Trans}(\pi_3, \min(i_1^2, i_2^3))$.

As we showed all cases satisfy (iv), it must be that (iv) holds for π_1, π_3 and index $\min(i_1^2, i_2^3)$. Moreover, as all conditions of Definition A.10 are satisfied for π_1 and π_3, it must be that $\pi_1 <_M \pi_3$. As we made no assumption about the paths π_1, π_2, π_3 this concludes our proof of $<_M$'s transitivity.

To prove the final part of the lemma we take an arbitrary $s \in I$ and $\sigma \in \mathit{PureStrat}_M$ and two arbitrary paths $\pi, \pi' \in \mathit{FinPath}^s_{M,\sigma}$. We will show that either $\pi \leq_M \pi'$ or $\pi' \leq_M \pi$ holds. Evidently, if $\pi = \pi'$, then we trivially satisfy the totality requirement. Otherwise, let $i \in \mathbb{N}$ to be the greatest natural number such that $i \leq |\pi|$ and $i \leq |\pi'|$ *and* condition (iii) holds for i, π and π'. We case split on the lengths of π and π'.

Observe that the first three cases are trivial: if $|\pi| = |\pi'| = i$, then $\pi = \pi'$, and hence $\pi \leq_M \pi'$ and $\pi' \leq_M \pi$; if $i = |\pi|$ and $i < |\pi'|$, then $\pi <_M \pi'$ and, if $i < |\pi|$ and $i = |\pi'|$, then $\pi' <_M \pi$. The remaining case is more involved. Suppose $i < |\pi|$ and $i < |\pi'|$. By our choice of i, $\textsc{Trans}(\pi, i) \neq \textsc{Trans}(\pi', i)$. Recall π and π' share the same source state and, if $i > 0$, then it must be that $\textsc{Trans}(\pi, i-1) = \textsc{Trans}(\pi', i-1)$ by our assumption on i — this means transitions $\textsc{Trans}(\pi, i)$ and $\textsc{Trans}(\pi', i)$ share the same source state. Moreover, because π and π' are both consistent with the *pure* strategy σ the transitions $\textsc{Trans}(\pi, i)$ and $\textsc{Trans}(\pi', i)$ share the same same distribution, also. Hence, by Definition 7.3, either $\textsc{Trans}(\pi, i) \prec_M \textsc{Trans}(\pi', i)$ or $\textsc{Trans}(\pi', i) \prec_M \textsc{Trans}(\pi, i)$ and, depending on which case holds we get $\pi <_M \pi'$ or $\pi' <_M \pi$, respectively.

As π and π' are comparable in all cases we conclude that π and π' are comparable. As we made no assumption on s, σ, π or π' we conclude the lemma holds. □

The order in which instrumentation visit the paths of the original program is dictated by the ordering described above. The knowledge that, say, this ordering is irreflexive is essential in our main proof.

We are now in a position to prove soundness of our MDP-level instrumentation:

Theorem 7.8 (Model-level soundness). *Let M be an MDP and let $M^{\mathbf{p}}$ be an instrumentation of M for some bound, $\mathbf{p} \in [0,1[\cap \mathbb{Q}$. We have that*

$$\mathit{Reach}^{+}(M^{\mathbf{p}}) \quad \textit{if and only if} \quad \mathit{Prob}^{+}(M) > \mathbf{p} \ .$$

Proof. Suppose $M = \langle S, I, T, L, R\rangle$ and $M^{\mathbf{p}} = \langle S^{\mathbf{p}}, I^{\mathbf{p}}, T^{\mathbf{p}}, L^{\mathbf{p}}, R^{\mathbf{p}}\rangle$. We assume the transition ordering $\preceq_M$ is used and that $\leq_M$ is the order over paths defined in Definition A.10. We prove each direction of the equivalence constructively.

($\Leftarrow$) Assume $\mathit{Prob}^{+}(M) > \mathbf{p}$. We will show that $\mathit{Reach}^{+}(M^{\mathbf{p}})$ holds by showing there exists a *finite* path

$$\pi^{\mathbf{p}} = \langle \pi_0, \mathit{mass}_0\rangle \xrightarrow{\lambda_0^{\mathbf{p}}} \langle \pi_1, \mathit{mass}_1\rangle \xrightarrow{\lambda_1^{\mathbf{p}}} \ldots \xrightarrow{\lambda_{n-1}^{\mathbf{p}}} \langle \pi_n, \mathit{mass}_n\rangle \ ,$$

in $\mathit{FinPath}^{I^{\mathbf{p}}}_{M^{\mathbf{p}}}$ such that $L^{\mathbf{p}}(\pi_n, \mathbf{F})$ holds. From Section 7.2.1, we know there must exist some initial state $s_{\mathrm{i}} \in I$, pure strategy $\rho \in \mathit{PureStrat}_M$ and *finite* set $\Pi \subseteq \mathbf{F}\text{-}\mathit{FinPath}^{s_{\mathrm{i}}}_{M,\sigma}$ in the original model, M, such that

$$\sum_{\pi \in \Pi} \textsc{Prob}_M(\pi) > \mathbf{p} \ . \tag{A.30}$$

Because ρ is a *pure* strategyn and because of Lemma A.11n we obtain a total order on the paths in Π. Suppose $\overline{\pi}_0$, $\overline{\pi}_1$, ..., $\overline{\pi}_k$ are the paths in Π ordered by $<_M$. We will construct $\pi^{\mathbf{p}}$ by "visiting" each path $\overline{\pi}_0, \overline{\pi}_1, \ldots, \overline{\pi}_k$ in $<_M$-order in $\pi^{\mathbf{p}}$. We will use i to range over indices of $\pi^{\mathbf{p}}$ and j to range over paths in $\overline{\pi}_0, \ldots, \overline{\pi}_k$. The invariant of our construction of $\pi^{\mathbf{p}}$ is that if a state, $\langle \pi_i, \mathit{mass}_i\rangle$, of $\pi^{\mathbf{p}}$ is such that $\pi_i = \overline{\pi}_j$ for some j, then we have

$$\mathit{mass}_i = \sum_{l=0}^{j-1} \textsc{Prob}_M(\overline{\pi}_l) \ . \tag{A.31}$$

We let the initial state of $\pi^{\mathbf{p}}$ be $\langle s_{\mathrm{i}}, 0\rangle \in I^{\mathbf{p}}$. Now consider the path $\overline{\pi}_0$. We can use explorative transitions (see condition (i) of Definition 7.6) to construct a path in $M^{\mathbf{p}}$ from $\langle s_{\mathrm{i}}, 0\rangle$ to $\langle \overline{\pi}_0, 0\rangle$. This construction trivially preserves (A.31). Recall that during exploration we have full control over the path of M we follow. Moreover, explorative transitions are enabled because, for any $\overline{\pi}_i$ to be in $\mathbf{F}\text{-}\mathit{FinPath}^{s_{\mathrm{i}}}_{M,\sigma}$, it must be that *only* in their last state $L(\overline{\pi}_i, \mathbf{F})$ holds.

Now suppose the last state $\langle \pi_i, \mathit{mass}_i\rangle$ of the path fragment of $\pi^{\mathbf{p}}$ constructed so far is such that we have $\pi_i = \overline{\pi}_j$ for some $j < k$ such that (A.31) holds. We will show we can continue the construction of $\pi^{\mathbf{p}}$ by showing there is a path from $\langle \pi_i, \mathit{mass}_i\rangle$ to $\langle \overline{\pi}_{j+1}, \mathit{mass}_i + \textsc{Prob}_M(\pi_i)\rangle$ in $M^{\mathbf{p}}$ (preserving (A.31)).

Observe that $\overline{\pi}_j$ satisfies $\mathbf{F}$ in M and hence we must backtrack first. By definition of $\mathbf{F}\text{-}\mathit{FinPath}^{s_{\mathrm{i}}}_{M,\sigma}$ we know that $\overline{\pi}_j$ is not a prefix of $\overline{\pi}_{j+1}$. As $\overline{\pi}_j <_M \overline{\pi}_{j+1}$ it must be that the paths $\overline{\pi}_j$ and $\overline{\pi}_{j+1}$ share the prefix up to some length l and then $\textsc{Trans}(\overline{\pi}_j, l) \prec \textsc{Trans}(\overline{\pi}_{j+1}, l)$. Hence, the prefix of $\overline{\pi}_{j+1}$ of size $l+1$, $\overline{\pi}^{\mathrm{prefix}}_{j+1}$ say, satisfies all criteria of the backtracking condition (ii) of Definition 7.6. We add the backtracking transition $\langle \pi_i, \mathit{mass}_i\rangle \to \langle \overline{\pi}^{\mathrm{prefix}}_{j+1}, \mathit{mass}_i + \textsc{Prob}_M(\pi_i)\rangle$ to $\pi^{\mathbf{p}}$. We can then use the exploration condition as before to extend $\pi^{\mathbf{p}}$ to $\langle \overline{\pi}_{j+1}, \mathit{mass}_i + \textsc{Prob}_M(\pi_i)\rangle$.

We can continue this construction of $\pi^{\mathbf{p}}$ and, because $|\Pi|$ is finite, we will eventually reach $\overline{\pi}_k$ at some state $\langle \pi_i, \mathit{mass}_i\rangle$ of $\pi^{\mathbf{p}}$ satisfying (A.31). For this state, by our invariant (A.31):

$$\begin{aligned}
\textsc{Prob}_M(\pi_i) + \mathit{mass}_i &= \textsc{Prob}_M(\overline{\pi}_k) + \sum_{j=0}^{k-1} \textsc{Prob}_M(\overline{\pi}_j) && \text{(Equation A.31)}\\
&= \sum_{\pi \in \Pi} \textsc{Prob}_M(\pi) && \text{(Rewriting)}\\
&> \mathbf{p} && \text{(Equation A.30)}
\end{aligned}$$

By definition of $L^{\mathbf{p}}$ this means that $L^{\mathbf{p}}(\langle \pi_i, \mathit{mass}_i\rangle, \mathbf{F})$ holds and hence, because there is a finite path, $\pi^{\mathbf{p}}$, that reaches the target in $M^{\mathbf{p}}$ we have that $\mathit{Reach}^{+}(M^{\mathbf{p}})$.

($\Rightarrow$) Assume $\mathit{Reach}^{+}(M^{\mathbf{p}})$ holds. To show $\mathit{Prob}^{+}(M) > \mathbf{p}$ we will show there exists an initial state and a pure strategy ρ of M under which the probability mass of paths

reaching the target exceeds $\mathbf{p}$. We will construct ρ from a path

$$\pi^{\mathbf{p}} = \langle \pi_0, \mathit{mass}_0 \rangle \xrightarrow{\lambda_0} \langle \pi_1, \mathit{mass}_1 \rangle \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_{n-1}} \langle \pi_n, \mathit{mass}_n \rangle \ .$$

of $M^{\mathbf{p}}$ witnessing $\mathit{Reach}^{+}(M^{\mathbf{p}})$. W.l.o.g. we assume $\pi^{\mathbf{p}}$ is taken such that $\langle \pi_n, \mathit{mass}_n \rangle$ is the only state of $\pi^{\mathbf{p}}$ where $\mathbf{F}$ holds. That is, if $L^{\mathbf{p}}(\langle \pi_i, \mathit{mass}_i \rangle, \mathbf{F})$ for some $i < n$ then we consider $\pi^{\mathbf{p}}$'s prefix of length i instead of $\pi^{\mathbf{p}}$. By definition of $L^{\mathbf{p}}$, we have $\mathit{mass}_n + \textsc{Prob}_M(\pi_n) > \mathbf{p}$ and $L(\textsc{Last}(\pi_n), \mathbf{F})$.

By definition of $\pi^{\mathbf{p}}$, all transitions of $\pi^{\mathbf{p}}$ must be due to conditions (i) or (ii) of Definition 7.6.[3] By the definition of these conditions it easily follows that for that all transitions $\langle \pi_i, \mathit{mass}_i \rangle \xrightarrow{\lambda_i} \langle \pi_{i+1}, \mathit{mass}_{i+1} \rangle$ in $\pi^{\mathbf{p}}$ we have that $\pi_i <_M \pi_{i+1}$ where $\leq_M$ is the order introduced in Definition A.10.

From the fact that $\pi_0 <_M \pi_1 <_M \ldots <_M \pi_n$ and from the transitivity and irreflexivity of $<_M$ established in Lemma A.11, it follows that all π_i are distinct. This allows us to define ρ as any pure strategy of M with $\rho(\pi_i) = [\lambda_i]$ for all $i < n$ with $\neg L(\textsc{Last}(\pi_i), \mathbf{F})$. Clearly all paths $\pi_0, \pi_1, \ldots, \pi_n$ are consistent with ρ. To complete the proof we show that $\mathit{Prob}^{+}(M) > \mathbf{p}$. First, let $\Pi \subseteq \{\pi_0, \ldots, \pi_n\}$ be the set such that, for each $i \leq n$, we have $\pi_i \in \Pi$ if and only if $L(\textsc{Last}(\pi_i), \mathbf{F})$. Note that by the assumptions on $\pi^{\mathbf{p}}$ we have $\pi_n \in \Pi$. Simple inductive arguments on $\pi^{\mathbf{p}}$ show that $\Pi \subseteq \mathbf{F}\text{-}\mathit{FinPath}^{s}_{M,\rho}$ for some $s \in I$ and that mass_n — the mass component of

[3] A transition cannot be due to (iii) as then $\langle \pi_n, \mathit{mass}_n \rangle$ would not be the only state of $\pi^{\mathbf{p}}$ satisfying $\mathbf{F}$.

the final state of $\pi^{\mathbf{p}}$ — equals $\sum_{\pi_i \in \Pi \setminus \{\pi_n\}} \textsc{Prob}_M(\pi_i)$. We have:

$$
\begin{aligned}
\mathit{Prob}^+(M) &= \sup_{\sigma \in \mathit{Strat}_M,\, s' \in I} \sum_{\pi \in \mathbf{F}\text{-}\mathit{FinPath}^{s'}_{M,\rho}} \textsc{Prob}_M(\pi) && \text{(Def. } \mathit{Prob}^+\text{, Lem. 3.19)} \\
&\geq \sum_{\pi \in \mathbf{F}\text{-}\mathit{FinPath}^{s}_{M,\rho}} \textsc{Prob}_M(\pi) && \text{(Def. sup)} \\
&\geq \sum_{\pi_i \in \Pi} \textsc{Prob}_M(\pi_i) && \text{(As } \Pi \subseteq \mathbf{F}\text{-}\mathit{FinPath}^{s}_{M,\rho}\text{)} \\
&= \textsc{Prob}_M(\pi_n) + \sum_{\pi_i \in \Pi \setminus \{\pi_n\}} \textsc{Prob}_M(\pi_i) && \text{(As } \pi_n \in \Pi\text{)} \\
&= \textsc{Prob}_M(\pi_n) + \mathit{mass}_n && \text{(As } \mathit{mass}_n = \sum_{\pi_i \in \Pi \setminus \{\pi_n\}} \textsc{Prob}_M(\pi_i)\text{)} \\
&> \mathbf{p} && \text{(Def. } L^{\mathbf{p}},\ \pi^{\mathbf{p}}\text{)}
\end{aligned}
$$

From our premise, $\mathit{Reach}^+(M^{\mathbf{p}})$, we have deduced that $\mathit{Prob}^+(M) > \mathbf{p}$ holds — this implication is also valid.

As we have proved both implications we conclude Theorem 7.8 holds. □

A.3.2 Proof of Theorem 7.18 (Program-level Soundness)

In this section we will show our program-level instrumentations are sound. That is, we will prove Theorem 7.18 on page 162. We first recall this theorem:

Theorem 7.18 (Program-level soundness). *Let P be any probabilistic program and let $\mathbf{p} \in [0,1[\cap \mathbb{Q}$ be an arbitrary bound. Let $P^{\mathbf{p}}$ be the program-level instrumentation of P for bound $\mathbf{p}$. We have*

$$\mathit{Reach}^+([\![P^{\mathbf{p}}]\!]) \quad \textit{if and only if} \quad \mathit{Prob}^+([\![P]\!]) > \mathbf{p}\ .$$

The main lines of our proof follow the following equivalences:

$$
\begin{aligned}
\mathit{Prob}^{+}([\![P]\!]) > \mathbf{p} &\iff \mathit{Reach}^{+}([\![P]\!]^{\mathbf{p}}) && \text{(Theorem 7.8)} \\
&\iff \mathit{Reach}^{+}([\![P^{\mathbf{p}}]\!]^{S'}) && \text{(Stuttering simulations)} \\
&\iff \mathit{Reach}^{+}([\![P^{\mathbf{p}}]\!]) && \text{(Reduction)}
\end{aligned}
$$

where $[\![P]\!]^{\mathbf{p}}$ is the model-level instrumentation of $[\![P]\!]$ and $[\![P^{\mathbf{p}}]\!]^{S'}$ is a transformation of $[\![P^{\mathbf{p}}]\!]$ induced by a set of states S'. The first equivalence uses the soundness result for model-level instrumentations, i.e. Theorem 7.8. Given this theorem, we only need to show that the program-level instrumentation, $[\![P^{\mathbf{p}}]\!]$, can reach a target state if and only if the model-level instrumentation, $[\![P]\!]^{\mathbf{p}}$, can. Due to the complexity of $P^{\mathbf{p}}$ we will do this through a third MDP, $[\![P^{\mathbf{p}}]\!]^{S'}$. This MDP is a relatively simple transformation of $[\![P^{\mathbf{p}}]\!]$.

The definition of $[\![P^{\mathbf{p}}]\!]^{S'}$, the preservation between $[\![P^{\mathbf{p}}]\!]^{S'}$ and $[\![P]\!]^{\mathbf{p}}$ and the preservation between $[\![P^{\mathbf{p}}]\!]^{S'}$ and $[\![P^{\mathbf{p}}]\!]$ require some definitions and lemmas.

We start with a lemma that shows a stuttering simulation preserves Reach^{+}. Unlike [DNV95], we deliberately do not restrict to equivalence relations:

Lemma A.12. *Let $M_1 = \langle S_1, I_1, T_1, L_1, R_1\rangle$ and $M_2 = \langle S_2, I_2, T_2, L_2, R_2\rangle$ be non-probabilistic MDPs and let $\mathscr{R}$ be a stuttering relation on $S_1 \uplus S_2$ such that $I_1 \subseteq \mathscr{R}^{-1}(I_2)$. We have:*

$$\mathit{Reach}^{+}(M_1) \implies \mathit{Reach}^{+}(M_2)\ .$$

Proof. We remark our proof is related to Theorem 2.3.2 in [DNV95]. If $\mathit{Reach}^{+}(M_1)$ then, by definition of Reach^{+}, there is a finite path $s_0 \to \ldots \to s_k \in \mathit{FinPath}_{M_1}$ with $s_0 \in I_1$ and $L_1(s_k, \mathbf{F})$. As $S_1 \subseteq \mathscr{R}(S_2)$, let $t_0 \in I_2$ be an initial state of M_2 such that $\langle s_0, t_0\rangle \in \mathscr{R}$. Using the property of stuttering simulations, with a simple inductive argument, we can show there exists, for every $i \leq k$, a finite path $t_0 \to \ldots \to t_m \in \mathit{FinPath}_{M_2}$ with $\langle s_i, t_m\rangle \in \mathscr{R}$. The case when $i = k$ implies there is a finite path in M_2 that starts from an initial state and that end in a target state — it must be that $\mathit{Reach}^{+}(M_2)$ holds. □

We will apply Lemma A.12 by finding a relation $\mathscr{R}$ on $[\![P^{\mathbf{p}}]\!]^{S'}$ and $[\![P]\!]^{\mathbf{p}}$'s states such that both $\mathscr{R}$ and $\mathscr{R}^{-1}$ are stuttering simulations satisfying the necessary restrictions on

initial states.

We now define a non-probabilistic MDP $M^{S'}$ for any given non-probabilistic MDP M and set of states S'.

Definition A.13. *Let $M = \langle S, I, T, L, R\rangle$ be a* non-probabilistic *MDP and let $S' \subseteq S$ be subset of S. We let $M^{S'} = \langle S^{S'}, I^{S'}, T^{S'}, L^{S'}, R^{S'}\rangle$ be the MDP with:*

- $S^{S'} = S'$,
- $I^{S'} = I \cap S'$
- $T^{S'}$ *is the smallest function satisfying:*
  - $[s'] \in T^{S'}(s)$ *if there is some finite path $s_0 \to \ldots \to s_k \in FinPath^s_M$ in M such that $s_0 = s$, $s_k = s'$ and $\forall 0 < i < k : s_i \notin S'$, and*
  - $[s] \in T^{S'}(s)$ *if there is* no *finite path $s_0 \to \ldots \to s_k \in FinPath^s_M$ in M such that $s_0 = s$ and $s_k \in S'$,*
- $L^{S'}(s, a) = L(s, a)$ *for all $s \in S^{S'}$ and $a \in AP$ and*
- $R^{S'}$ *is defined arbitrarily.*

Informally, $M^{S'}$ is M where finite paths between states of S' are turned into transitions. We will use this transformation to turn the sequences of transitions we need to realise backtracking in $[\![P^{\mathbf{P}}]\!]$, into single backtracking transitions in $[\![P^{\mathbf{P}}]\!]^{S'}$. With some minor restrictions on the definition of the set S', the reduction in Definition A.13 preserves $Reach^+$. We formalise this in the following lemma:

Lemma A.14. *Let $M = \langle S, I, T, L, R\rangle$ be a non-probabilistic MDP and let $S' \subseteq S$ be subset of S such that $I \subseteq S'$ and $\{s \in S \mid L(s, \mathbf{F})\} \subseteq S'$. We have that:*

$$Reach^+(M) \quad \textit{if and only if} \quad Reach^+(M^{S'}) \,.$$

Proof. We show both directions separately.

($\Rightarrow$) If $Reach^+(M)$ then, by definition of $Reach^+$, there is a finite path $s_0 \to \ldots \to s_k \in FinPath_M$ with $s_0 \in I$ and $L(s_k, \mathbf{F})$. Recall that, by our constraint on S',

we have $s_0 \in S'$. By induction, for every $i \leq k$ with $s_i \in S'$ there is a finite path $t_0 \to \ldots \to t_n \in \mathit{FinPath}_{M^{S'}}$ with $t_0 = s_0$ and $t_n = s_i$. As, by our assumption on S', we also have $s_k \in S'$, there must be a finite path in $M^{S'}$ from s_0 to s_k. As s_k is a target state in $M^{S'}$ it must be that $\mathit{Reach}^{+}(M^{S'})$ holds.

($\Leftarrow$) If $\mathit{Reach}^{+}(M^{S'})$ then, by definition of Reach^{+}, there is a finite path $t_0 \to \ldots \to t_k \in \mathit{FinPath}_{M^{S'}}$ with $t_0 \in I^{S'}$ and $L^{S'}(t_k, \mathbf{F})$. By our assumption on S' we have that $t_0 \in I$. Let $n \leq k$ be the smallest index for which $L^{S'}(t_n, \mathbf{F})$ holds. By this assumption it must be that all transitions in $t_0 \to \ldots \to t_n$ are due to the first condition of $T^{S'}$. Using this condition, with a inductive argument, we can show there exists, for every $i \leq n$, a finite path $s_0 \to \ldots \to s_m \in \mathit{FinPath}_M$ with $t_0 = s_0$ and $s_m = t_i$. The case when $i = n$ implies there is a finite path in M that starts from an initial state and that end up in t_n — a target state in M — and hence it must be that $\mathit{Reach}^{+}(M)$ holds. □

Having introduced the necessary definitions and lemmas we are now in a position to present our main soundness proof:

Theorem 7.18 (Program-level soundness). *Let P be any probabilistic program and let $\mathbf{p} \in [0,1[\cap \mathbb{Q}$ be an arbitrary bound. Let $P^{\mathbf{p}}$ be the program-level instrumentation of P for bound $\mathbf{p}$. We have*

$$\mathit{Reach}^{+}([\![P^{\mathbf{p}}]\!]) \quad \textit{if and only if} \quad \mathit{Prob}^{+}([\![P]\!]) > \mathbf{p}\,.$$

Proof. From Theorem 7.8 we have that $\mathit{Prob}^{+}([\![P]\!]) > \mathbf{p}$ if and only if $\mathit{Reach}^{+}([\![P]\!]^{\mathbf{p}})$. Remaining to show is that $\mathit{Reach}^{+}([\![P^{\mathbf{p}}]\!])$ if and only if $\mathit{Reach}^{+}([\![P]\!]^{\mathbf{p}})$. To this end, suppose

$$P = \langle\langle \mathscr{L}, \mathscr{E}\rangle, \{\mathscr{L}_{\mathrm{N}}, \mathscr{L}_{\mathrm{P}}, \mathscr{L}_{\mathrm{B}}\}, \ell_i, \mathscr{L}_{\mathrm{T}}, \mathscr{L}_{\mathrm{C}}, \mathit{Var}, \mathit{Sem}\rangle, \qquad [\![P]\!]^{\mathbf{p}} = \langle S_1^{\mathbf{p}}, I_1^{\mathbf{p}}, T_1^{\mathbf{p}}, L_1^{\mathbf{p}}, R_1^{\mathbf{p}}\rangle,$$
$$P^{\mathbf{p}} = \langle\langle \mathscr{L}^{\mathbf{p}}, \mathscr{E}^{\mathbf{p}}\rangle, \{\mathscr{L}_{\mathrm{N}}^{\mathbf{p}}, \mathscr{L}_{\mathrm{P}}^{\mathbf{p}}, \mathscr{L}_{\mathrm{B}}^{\mathbf{p}}\}, \ell_i^{\mathbf{p}}, \mathscr{L}_{\mathrm{T}}^{\mathbf{p}}, \mathscr{L}_{\mathrm{C}}^{\mathbf{p}}, \mathit{Var}^{\mathbf{p}}, \mathit{Sem}^{\mathbf{p}}\rangle \text{ and} \qquad [\![P^{\mathbf{p}}]\!] = \langle S_2^{\mathbf{p}}, I_2^{\mathbf{p}}, T_2^{\mathbf{p}}, L_2^{\mathbf{p}}, R_2^{\mathbf{p}}\rangle\,.$$

For simplicity we will assume $\ell_i \notin \mathscr{L}_{\mathrm{T}}$ — a dummy initial location could easily be added to alleviate this restriction. In accordance with the proof strategy discussed at the beginning

of this section, let

$$S' = \{\langle \ell_2, u_2 \rangle \in \mathscr{L}^{\mathbf{P}} \times \mathscr{U}_{Var^{\mathbf{P}}} \mid \ell_2 \in (\mathscr{L} \setminus \mathscr{L}_{\mathrm{T}}) \cup \{\mathrm{init}, \mathrm{hit}, \mathrm{miss}\}\} .$$

By definition Definition 7.17 and 4.2 we have that

$$I_2^{\mathbf{P}} = (\{\mathrm{init}\} \times \mathscr{U}_{Var^{\mathbf{P}}}) \subseteq S' \quad \text{and}$$
$$\{s_2 \in S_2 \mid L_2(s_2, \mathbf{F})\} = (\{\mathrm{hit}\} \times \mathscr{U}_{Var^{\mathbf{P}}}) \subseteq S' .$$

Hence, all conditions to apply Lemma A.14 are satisfied and we obtain that $Reach^{+}([\![P^{\mathbf{P}}]\!])$ if and only if $Reach^{+}([\![P^{\mathbf{P}}]\!]^{S'})$. Remaining to show is that $Reach^{+}([\![P^{\mathbf{P}}]\!]^{S'})$ if and only if $Reach^{+}([\![P]\!]^{\mathbf{P}})$. We will show this via stuttering simulations. To define this simulation we first define a function, $\textsc{ProbLocs} : FinPath_{[\![P]\!]}^{\{\ell_i\} \times \mathscr{U}_{Var}} \times \mathbb{N} \to \mathbb{P}\mathbb{N}$ which is defined for every $\pi_1 = \langle \ell_1^0, u_1^0 \rangle \to \ldots \to \langle \ell_1^n, u_1^n \rangle \in FinPath_{[\![P]\!]}^{\{\ell_i\} \times \mathscr{U}_{Var}}$ and $i \in \mathbb{N}$ as

$$\textsc{ProbLocs}(\pi_1, i) = \{j \in \mathbb{N} \mid j < n,\ \ j \leq i,\ \ \ell_1^j \in \mathscr{L}_{\mathrm{P}}\} .$$

Informally, $\textsc{ProbLocs}(\pi_1, i)$ yields the set of all state-indices of π_1 less or equal to i that are probabilistic locations in π_1.

We will define $\mathscr{R} \subseteq S_1^{\mathbf{P}} \times S_2^{\mathbf{P},S'}$ such that $\mathscr{R}$ and $\mathscr{R}^{-1}$ are stuttering simulations on $[\![P]\!]^{\mathbf{P}} \uplus [\![P^{\mathbf{P}}]\!]^{S'}$ and such that $I_1^{\mathbf{P}} \subseteq \mathscr{R}^{-1}(I_2^{\mathbf{P},S'})$ and $I_2^{\mathbf{P},S'} \subseteq \mathscr{R}(I_1^{\mathbf{P}})$. The idea is then we can apply Lemma A.12 twice to show $Reach^{+}([\![P]\!]^{\mathbf{P}})$ if and only if $Reach^{+}([\![P^{\mathbf{P}}]\!]^{S'})$.

We now will define $\mathscr{R}$. Let $\langle s_1, s_2 \rangle$ be a tuple in $S_1^{\mathbf{P}} \times S_2^{\mathbf{P},S'}$. Suppose $s_1 = \langle \pi_1, mass_1 \rangle$ with suppose $\pi_1 = \langle \ell_1^0, u_1^0 \rangle \xrightarrow{\lambda_0} \langle \ell_1^1, u_1^1 \rangle \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_{n-1}} \langle \ell_1^n, u_1^n \rangle$ and suppose $s_2 = \langle \ell_2, u_2 \rangle$. The relation $\mathscr{R}$ includes $\langle s_1, s_2 \rangle$ iff one of the following conditions holds:

(R1) All of the following conditions hold:

- (a) $\langle s_1, s_2 \rangle \in I_1^{\mathbf{P}} \times I_2^{\mathbf{P},S'}$ and
- (b) $\forall var \in Var : u_1^n(var) = u_2(var)$.

(R2) All of the following conditions hold:

- (a) $\ell_1^n = \ell_2$ and $\ell_1^n, \ell_2 \in \mathscr{L} \setminus \mathscr{L}_{\mathrm{T}}$,

(b) $\forall var \in Var : u_1^n(var) = u_2(var)$,

(c) $mass_1 = u_2(\texttt{mass})$,

(d) $\textsc{Prob}_{[\![P]\!]^{\mathbf{p}}}(\pi_1) = u_2(\texttt{prob})$,

(e) $|\textsc{ProbLocs}(\pi_1, n)| = u_2(\texttt{index})$ and

(f) $\forall i \in \textsc{ProbLocs}(\pi_1, n)$ all of the following conditions hold:

(1) $u_1^{i+1}(\textsc{LValue}_{\ell_1^i}) = u_2(\texttt{res}_{\ell_1^i})\texttt{[}|\textsc{ProbLocs}(\pi_1, i)|\texttt{]}$,

(2) $\ell_1^i = u_2(\texttt{loc})\texttt{[}|\textsc{ProbLocs}(\pi_1, i)|\texttt{]}$ and

(3) $\forall var \in Var : u_1^i(var) = u_2(\texttt{copy\_}var)\texttt{[}|\textsc{ProbLocs}(\pi_1, i)|\texttt{]}$.

(4) $\textsc{Prob}_{[\![P]\!]^{\mathbf{p}}}(\langle \ell_1^0, u_1^0\rangle \xrightarrow{\lambda_0} \ldots \xrightarrow{\lambda_{i-1}} \langle \ell_1^i, u_1^i\rangle) = u_2(\texttt{copy\_prob})\texttt{[}|\textsc{ProbLocs}(\pi_1, i)|\texttt{]}$.

(R3) All of the following conditions hold:

(a) $\ell_1^n \in \mathscr{L}_{\mathrm{T}}$,

(b) $\ell_2 = \text{hit}$ if $u_2(\texttt{mass}) + u_2(\texttt{prob})(\pi_1) > \mathbf{p}$ and $\ell_2 = \text{miss}$ otherwise,

(c) conditions (R2.b) — (R2.f) hold.

Trivially, we have that $I_1^{\mathbf{p}} \subseteq \mathscr{R}^{-1}(I_2^{\mathbf{p},S'})$ and $I_2^{\mathbf{p},S'} \subseteq \mathscr{R}(I_1^{\mathbf{p}})$ due to (R1). Remaining to prove is that $\mathscr{R}$ and $\mathscr{R}^{-1}$ are stuttering simulations on $[\![P]\!]^{\mathbf{p}} \uplus [\![P^{\mathbf{p}}]\!]^{S'}$.

To show $\mathscr{R}$ and $\mathscr{R}^{-1}$ are stuttering simulations, according to Definition 3.20, we firstly need that $L(s_1, a) = L(s_2, a)$ for all atomic propositions $a \in \text{AP}$ for all $\langle s_1, s_2\rangle \in \mathscr{R}$. This trivially holds because only $\mathbf{F}$ can hold in states of instrumentations and the validity of $\mathbf{F}$ matches in $[\![P]\!]^{\mathbf{p}}$ and $[\![P^{\mathbf{p}}]\!]^{S'}$ due to condition (R3.a) and (R3.b) (and because $\ell_i \notin \mathscr{L}_{\mathrm{T}}$). Remaining to show is that the transitions of $[\![P]\!]^{\mathbf{p}}$ and $[\![P^{\mathbf{p}}]\!]^{S'}$ can be matched. That is, for any $\langle s_1, s_2\rangle \in \mathscr{R}$ we need to show that the following conditions hold:

(C1) $\forall s_1 \to s_1' : \exists s_2^0 \to \ldots \to s_2^k \in \mathit{FinPath}_{[\![P^{\mathbf{p}}]\!]^{S'}}$: such that $s_2 = s_2^0$ and

- $\langle s_1, s_2^i\rangle \in \mathscr{R}$ for all $i < k$, and
- $\langle s_1', s_2^k\rangle \in \mathscr{R}$, and

(C2) $\forall s_2 \to s_2' : \exists s_1^0 \to \ldots \to s_1^k \in \mathit{FinPath}_{[\![P]\!]^{\mathbf{p}}}$: such that $s_1 = s_1^0$ and

- $\langle s_1^i, s_2\rangle \in \mathscr{R}$ for all $i < k$, and
- $\langle s_1^k, s_2'\rangle \in \mathscr{R}$, and

We now take an arbitrary $\langle s_1, s_2\rangle \in \mathscr{R}$ and show (C1) and (C2) hold for this tuple. We again suppose $s_1 = \langle \pi_1, mass_1\rangle$ with $\pi_1 = \langle \ell_1^0, u_1^0\rangle \xrightarrow{\lambda_0} \langle \ell_1^1, u_1^1\rangle \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_{n-1}} \langle \ell_1^n, u_1^n\rangle$ and suppose $s_2 = \langle \ell_2, u_2\rangle$. Clearly, if $\langle s_1, s_2\rangle \in \mathscr{R}$, then either (R1), (R2) or (R3) applies. We consider each case separately:

(R1) We first show (C2). Observe there is only one transition from s_2 in $[\![P^{\mathbf{P}}]\!]^{S'}$, namely $s_2 \to \langle \ell_2', u_2'\rangle$, where $u_2'(\texttt{mass}) = 0$, $u_2'(\texttt{prob}) = 1$ and $u_2'(\texttt{index}) = 0$ and $u_2'(var) = u_2(var)$ for all $var \in Var$. We have that $\ell_2' = \ell_i$ unless $\ell_i \in \mathscr{L}_{\text{T}}$, in which case either $\ell_2' = \text{miss}$ or $\ell_2' = \text{hit}$. This special treatment for $\mathscr{L}_{\text{T}}$ follows from the definition of $[\![P^{\mathbf{P}}]\!]^{S'}$ — the locations in $\mathscr{L}_{\text{T}}$ are not in S' and hence, depending on the value of $u_2'(\texttt{mass}) + u_2'(\texttt{prob})$, we would transition to either "miss" or "hit" instead.

Suppose $\ell_i \notin \mathscr{L}_{\text{T}}$. The path s_1 (without any transitions) trivially satisfies (C2) as $s_1 \in I_1^{\mathbf{P},S'}$ implies $mass = 0$, $n = 0$, $\ell_1^0 = \ell_i$, $\textsc{Prob}_M[\![P]\!](\pi_1) = 1$ and $\textsc{ProbLocs}(\pi_1, n) = \emptyset$ — trivially, $\langle\langle \pi_1, mass_1\rangle, \langle \ell_2', u_2'\rangle\rangle \in \mathscr{R}$ due to (R2). The case for when $\ell_i \in \mathscr{L}_{\text{T}}$ is analogous but uses (R3), instead.

Remaining to show is (C1). Let $s_1 \to s_1'$ be any transition from s_1 in $[\![P]\!]^{\mathbf{P}}$. We have already shown that there exists a transition $s_2 \to s_2'$ such that $\langle s_1, s_2'\rangle \in \mathscr{R}$ due to condition (R2) or (R3) of $\mathscr{R}$'s definition. We depend on the proof of (C1) for condition (R2) and (R3) to give us a finite path π_2 of $[\![P^{\mathbf{P}}]\!]^{S'}$ such that the path $s_2 \to \pi_2$ satisfies (C1).

(R2) Note that, because $\ell_1^n \notin \mathscr{L}_{\text{T}}$, we *only* have to deal with "explorative" transitions. We do a case split on the location type of ℓ_1^n, ℓ_2:

- Suppose $\ell_1^n, \ell_2 \in \mathscr{L}_{\text{E}} \setminus \mathscr{L}_{\text{T}}$. By Definition 4.2 and 7.6, the *only* outgoing transition in $[\![P]\!]^{\mathbf{P}}$ is $s_1 \to \langle \pi_1 \to \langle \ell_1^n, u_1^n\rangle, mass_1\rangle$. Analogously, by Definition 4.2, the only *only* outgoing transition in s_2 is the self-loop $s_2 \to s_2$. The transition added to π_1 is non-probabilistic per definition and has no influence on the conditions of (R2) — we trivially have $\langle\langle \pi_1 \to \langle \ell_1^n, u_1^n\rangle, mass_1\rangle, s_2\rangle \in \mathscr{R}$ and (C1) and (C2) are trivally satisfied.
- Suppose $\ell_1^n, \ell_2 \in (\mathscr{L}_{\text{N}} \cup \mathscr{L}_{\text{B}} \cup \mathscr{L}_{\text{C}}) \setminus \mathscr{L}_{\text{T}}$. Let us first show (C1). Clearly, the *all* transitions from $s_1 = \langle \pi_1, mass_1\rangle$ in $[\![P]\!]^{\mathbf{P}}$ are due to the exploration

condition (i) of Definition 7.6. Let us take an arbitrary such transition $s_1 \to \langle \pi \to \langle \ell_1^{n+1}, u_1^{n+1} \rangle, mass_1 \rangle$

From Definition 7.17 we know $\langle \ell_2, \ell_1^{n+1} \rangle \in \mathscr{E}_2^{\mathbf{p}}$ and $Sem_2^{\mathbf{p}}(\langle \ell_2, \ell_1^{n+1} \rangle)$ is lifted from $Sem(\langle \ell_1^n, \ell_1^{n+1} \rangle)$. This means there exists a transition $s_2 \to \langle \ell_1^{n+1}, u_2^{n+1} \rangle$ in $[\![P^{\mathbf{p}}]\!]$ — but maybe not in $[\![P^{\mathbf{p}}]\!]^{S'}$ — such that the original variables of P match in u_1^{n+1} and u_2^{n+1} and the instrumentation variables do not change value. By definition of S', this transition is present in $[\![P^{\mathbf{p}}]\!]^{S'}$ only if $\ell_1^{n+1} \notin \mathscr{L}_{\mathrm{T}}$.

Observe that in this case we have

$$\langle \langle \pi \to \langle \ell_1^{n+1}, u_1^{n+1} \rangle, mass_1 \rangle, \langle \ell_1^{n+1}, u_2^{n+1} \rangle \rangle$$

via condition (R2) and hence transition $s_2 \to \langle \ell_1^{n+1}, u_2^{n+1} \rangle$ satisfies (C1). To deal with the case when $\ell_1^{n+1} \in \mathscr{L}_{\mathrm{T}}$, observe that we have a transition $s_2 \to \langle \mathrm{hit}, u_2^{n+1} \rangle$ in $[\![P^{\mathbf{p}}]\!]^{S'}$ if $u^n(\texttt{mass}) + u^n(\texttt{prob}) > \mathbf{p}$ and a transition $s_2 \to \langle \mathrm{miss}, u_2^{n+1} \rangle$, otherwise. In either case it is easy to see all conditions of (R3) are satisfied.

Finally, because we matched transitions to transitions (instead of longer path) and because our discussion covered *all* the transitions from s_2 in $[\![P^{\mathbf{p}}]\!]^{S'}$, condition (C2) is satisfied, also.

– Suppose $\ell_1^n, \ell_2 \in \mathscr{L}_{\mathrm{P}} \setminus \mathscr{L}_{\mathrm{T}}$. We first show (C1). Clearly, *all* transitions from $s_1 = \langle \pi_1, mass_1 \rangle$ in $[\![P]\!]^{\mathbf{p}}$ are due to the exploration condition (i) of Definition 7.6. Let us take an arbitrary such transition

$$\langle \pi_1, mass_1 \rangle \to \langle \pi_1 \xrightarrow{\lambda_n} \langle \ell_1^{n+1}, u_1^{n+1} \rangle, mass_1 \rangle$$

where, by definition, $\ell_1^{n+1} \in \mathscr{L}$ and u_1^n and u_1^{n+1} differ only in their value for $\textsc{LValue}_{\ell_1^n}$. Let us assume for now that $\ell_1^{n+1} \notin \mathscr{L}_{\mathrm{T}}$. We will match this transition with a transition $s_2 \to s_2'$ in $[\![P^{\mathbf{p}}]\!]^{S'}$. Let us take the finite path from $s_2 = \langle \ell_2, u_2 \rangle$ in $[\![P^{\mathbf{p}}]\!]$ of the form

$$\langle \ell_2, u_2 \rangle \to \langle \mathrm{expl}_\ell, u_2' \rangle \to \langle \mathrm{valid}_\ell, u_2'' \rangle \to \langle \ell_2', u_2'' \rangle$$

where $u_2'(\texttt{res}_\ell)[u_2(\texttt{index})+1] = u_1^{n+1}(\text{LVALUE}_{\ell_2})$. Evidently, considering the definition of S' and because we assumed $\ell_1^{n+1} \notin \mathscr{L}_\text{T}$, $\langle \ell_2, u_2\rangle \to \langle \ell_2', u_2''\rangle$ is a transition of $[\![P^{\mathbf{p}}]\!]^{S'}$. Moreover, it is easy to see

$$\langle\langle \pi_1 \xrightarrow{\lambda_n} \langle \ell_1^{n+1}, u_1^{n+1}\rangle, \mathit{mass}_1\rangle, \langle \ell_2', u_2''\rangle\rangle \in \mathscr{R}$$

via condition (R2) as, by following the definition of $\mathit{Sem}^{\mathbf{p}}$ in Definition 7.17, it easily follows all conditions of (R2) hold for this tuple.

To see (C1) also holds when $\ell_1^{n+1} \in \mathscr{L}_\text{T}$ we simply augment the transition $\langle \ell_2', u_2''\rangle \to \langle \text{hit}, u_2''\rangle$ or $\langle \ell_2', u_2''\rangle \to \langle \text{miss}, u_2''\rangle$ to the path of $[\![P^{\mathbf{p}}]\!]$ we consider (depending on whether $u_2''(\texttt{mass}) + u_2''(\texttt{prob}) > \mathbf{p}$) and use an analogous proof with condition (R3) instead of condition (R2).

Due to the constraint in "valid$_\ell$" our discussion covers *all* transitions of $[\![P^{\mathbf{p}}]\!]^{S'}$ and hence (C2) holds, also.

(R3) We first show (C1). Let us first consider the case when $s_1 \to s_1'$ is a bracktracking transition. An arbitrary such transition is

$$\langle\langle \ell_1^0, u_1^0\rangle \xrightarrow{\lambda_0} \ldots \xrightarrow{\lambda_{i-1}} \langle \ell_1^i, u_1^i\rangle \xrightarrow{\lambda_i} \langle \ell_1^{i+1}, u_1^{i+1}\rangle \xrightarrow{\lambda_{i+1}} \ldots \xrightarrow{\lambda_{n-1}} \langle \ell_1^n, u_1^n\rangle, \mathit{mass}_1\rangle \longrightarrow$$
$$\langle\langle \ell_1^0, u_1^0\rangle \xrightarrow{\lambda_0} \ldots \xrightarrow{\lambda_{i-1}} \langle \ell_1^i, u_1^i\rangle \xrightarrow{\lambda_i} \langle \ell_1', u_1'\rangle, \mathit{mass}_1 + \text{PROB}_M(\pi_1)\rangle$$

where $u_1^{i+1}(\text{LVALUE}_{\ell_1^i}) \neq u_1'(\text{LVALUE}_{\ell_1^i})$ as well as $\langle u_1^{i+1}(\text{LVALUE}_{\ell_1^i}), u_1'(\text{LVALUE}_{\ell_1^i})\rangle \in \text{ORDER}_\ell$ for some $i < n$.

We will match this transition with a transition $s_2 \to s_2'$ in $[\![P^{\mathbf{p}}]\!]^{S'}$. To this end, we will show there is a finite path π_2 from $s_2 = \langle \ell_2, u_2\rangle$ to $\langle \ell_1', u_2'\rangle$ in $[\![P^{\mathbf{p}}]\!]$ such that

$$\langle\langle\langle \ell_1^0, u_1^0\rangle \xrightarrow{\lambda_0} \ldots \xrightarrow{\lambda_i} \langle \ell_1', u_1'\rangle, \mathit{mass}_1 + \text{PROB}_M(\pi_1)\rangle, \langle \ell_1', u_2'\rangle\rangle \in \mathscr{R}\,. \tag{A.32}$$

We know that ℓ_2 is either "hit" or "miss". We start with a transition $\langle \ell_2, u_2\rangle \to \langle \text{backtr}, u_2^{\text{t1}}\rangle$ to π_2. Note the only difference between π_2 and π_2^{t1} is the value of $\texttt{mass}$, i.e. $u_2^{\text{t1}}(\texttt{mass}) = u_2(\texttt{mass}) + u_2(\texttt{prob})$.

We now need to iterate through probabilistic choices that come *after* the probabilistic choice we wish to backtrack to. Suppose $m_1, \ldots, m_k$ are the indices in $\textsc{ProbLocs}(\pi_1, n) - \textsc{ProbLocs}(\pi_1, i)$ in descending order. We augment π_2 with the transitions

$$\begin{aligned}
&\langle \text{backtr}, u_2^{\text{t1}} \rangle \to \langle \text{backtr}_{\ell_1^{m_1}}, \ldots \rangle \to \langle \text{choose}_{\ell_1^{m_1}}, \ldots \rangle \to \langle \text{skip}, \ldots \rangle \to \\
&\langle \text{backtr}, \ldots \rangle \to \langle \text{backtr}_{\ell_1^{m_2}}, \ldots \rangle \to \langle \text{choose}_{\ell_1^{m_2}}, \ldots \rangle \to \langle \text{skip}, \ldots \rangle \to \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \ldots \\
&\langle \text{backtr}, \ldots \rangle \to \langle \text{backtr}_{\ell_1^{m_k}}, \ldots \rangle \to \langle \text{choose}_{\ell_1^{m_k}}, \ldots \rangle \to \langle \text{skip}, u_2^{\text{t2}} \rangle \to \\
&\langle \text{backtr}, u_2^{m_k} \rangle \to \langle \text{backtr}_{\ell_1^i}, \ldots \rangle \to \langle \text{choose}_{\ell_1^i}, \ldots \rangle \to \langle \text{take}_{\ell_1^i}, \ldots \rangle \to \\
&\qquad\qquad \langle \text{order}_{\ell_1^i}, \ldots \rangle \to \langle \text{expl}_{\ell_1^i}, \ldots \rangle \to \langle \text{valid}_{\ell_1^i}, u_2' \rangle \to \langle \ell_1', u_2' \rangle
\end{aligned}$$

with, of course, $u_2'(\texttt{res}_\ell)[u_2(\texttt{index})] = u_1'(\textsc{LValue}_{\ell_1^i})$. Although u_2^{t1} and u_2^{t2} differ in their values for the variables in *Var* and `prob`, they agree on `mass` as well as the arrays in the instrumentation. We also have $u^{\text{t2}}(\texttt{index}) = u^{\text{t1}}(\texttt{index}) - k = |\textsc{ProbLocs}(\pi_1, i)|$.

Suppose $\ell_1' \notin \mathscr{L}_{\text{T}}$. Because of the definition of S' we have that $\langle \ell_2, u_2 \rangle \to \langle \ell_2', u_2' \rangle$ is a transition of $[\![P^{\texttt{P}}]\!]^{S'}$. Moreover, it turns out that (A.32) holds because all conditions in (R2) are satisfied. To see this, observe that due to (R2.f), the variables in *Var* coincide with u_1^i after the transition $\langle \text{backtr}_{\ell_1^i}, \ldots \rangle \to \langle \text{choose}_{\ell_1^i}, \ldots \rangle$. Analogously `prob` equals the probability of of π_1's prefix of length i after this transition. Both the variables in P and `prob` are updated to encorporate the final step $\langle \ell_1^i, u_1^i \rangle \xrightarrow{\lambda_i} \langle \ell_1', u_1' \rangle$ in the transition from "$\text{expl}_{\ell_1^i}$".

If $\ell_1' \in \mathscr{L}_{\text{T}}$ then we can use an analogous argument with by augmenting π_2 with a transition to $\langle \text{hit}, \ell_2' \rangle$ or $\langle \text{miss}, \ell_2' \rangle$ and considering the conditions in (R3) instead.

There is also a possibility that there is no backtracking transition available in s_1. In this case there is a self-loop in s_1 and hence we have to match the transition $\langle \pi_1, \mathit{mass}_1 \rangle \to \langle \pi_1, \mathit{mass}_1 \rangle$. This, of course, we can do with the path s_2 (without any transitions).

The conditions in "backtr", "order_ℓ", "valid_ℓ" are such that *all* transitions from

$\langle \ell_2, u_2 \rangle$ of $[\![P^{\mathbf{p}}]\!]^{S'}$ correspond with a backtracking transition of $[\![P]\!]^{\mathbf{p}}$ as described above. Hence, (C2) holds also.

This concludes that $\mathscr{R}$ and $\mathscr{R}^{-1}$ are indeed stuttering simulations on $[\![P]\!]^{\mathbf{p}} \uplus [\![P^{\mathbf{p}}]\!]^{S'}$. We already established that $I_1^{\mathbf{p}} \subseteq \mathscr{R}^{-1}(I_2^{\mathbf{p},S'})$ and $I_2^{\mathbf{p},S'} \subseteq \mathscr{R}(I_1^{\mathbf{p}})$ and hence we can use Lemma A.12 twice to obtain

$$\mathit{Reach}^{+}([\![P]\!]^{\mathbf{p}}) \Longrightarrow \mathit{Reach}^{+}([\![P^{\mathbf{p}}]\!]^{S'}) \quad \text{and} \quad \mathit{Reach}^{+}([\![P^{\mathbf{p}}]\!]^{S'}) \Longrightarrow \mathit{Reach}^{+}([\![P]\!]^{\mathbf{p}}) \,.$$

Combining this with the previous equivalences we get

$$\mathit{Reach}^{+}([\![P^{\mathbf{p}}]\!]) \Leftrightarrow \mathit{Reach}^{+}([\![P^{\mathbf{p}}]\!]^{S'}) \Leftrightarrow \mathit{Reach}^{+}([\![P]\!]^{\mathbf{p}}) \Leftrightarrow \mathit{Prob}^{+}([\![P]\!]) > \mathbf{p} \,. \qquad \square$$

Appendix B

Tools

In this appendix, we describe we developed to evaluate the verification techniques presented in this thesis. We start with QPROVER in Section B.1, an implementation of the abstraction-refinement approach described in Chapter 5. Section B.2 then concerns PROBITY, a tool implementing the instrumentation-based approach of Chapter 7. .

B.1 QPROVER

The abstraction-refinement methodology described in Chapter 5 is implemented in a tool we call QPROVER.[1] This model checker verifies ordinary ANSI-C programs which call quantitative functions such as those shown in Figure 4.3, page 56. Our implementation is an extensive adaptation of the CEGAR-based model checker SATABS [CKSY05]. To obtain a probabilistic program as defined in Definition 4.1 we use SATABS' infrastructure for preprocessing, compiling source code, removing side-effects and function pointers and for inlining function calls [Kro10b, CKSY05]. We also use a built-in constant propagation analysis to simplify programs wherever possible. Because game-based abstractions are substantially different from those used in SATABS, we have reimplemented most of the abstraction-refinement loop. We still rely on existing functionality for the abstraction procedure (generating SAT formulas from syntax) and the refinement procedure (taking weakest preconditions). To model check game abstractions we employ an adapted version

[1] QPROVER is available from http://www.prismmodelchecker.org/qprover/.

of PRISM's symbolic value iteration algorithms for MDPs.[2] For a detailed discussion we refer to [Par02].

Our tool can deal with most of ANSI-C, including bit-level operations, functions, pointers, side-effects, function pointers, type casts, arrays, structs, unions and goto statements. However, it is currently not able to deal with recursion, concurrency or dynamic memory allocation in a meaningful way.

B.2 PROBITY

The instrumentation-based verification technique described in Chapter 7 is implemented in a tool called PROBITY.[3] Like QPROVER, the software that can be verified by this model checker are ordinary ANSI-C programs that call certain quantitative functions. The program-level instrumentation function described in Chapter 7 is implemented as a source-code transformation in PROBITY. Like QPROVER we use the infrastructure of the model checker SATABS for preprocessing, compiling source code and removing side-effects and function pointers (see, e.g., [Kro10b, CKSY05]).

To verify instrumented programs we rely on an adapted version of a model checker called WOLVERINE, which implements the interpolation-based model checking algorithm in [McM06]. The interpolating decision procedure that is used in WOLVERINE is the one described in [KW09]. We adapted this model checker with the extensions and optimisations described in Section 7.4, including the analysis of proofs and counter-examples, the greedy explation heuristic and the template invariant extension.

[2] This part of the implementation is due to David Parker.
[3] PROBITY is available from http://www.prismmodelchecker.org/probity/.

Appendix C

Case Studies

In this section, we discuss the case studies that are considered in this thesis. The case studies we have considered can be categorised as follows:

- PING, TFTP, NTP (*network clients*),
- FREI, HERM (*randomised algorithms*),
- MGALE, AMP, FAC (*pGCL programs*),
- BRP, ZERO, CONS (*PRISM programs*).

Each case study is a probabilistic program written in ANSI-C.[1] To give some indication of scale, we give the number of lines of code and the number of control-flow locations for each case study in Figure C.1. Note that this is the number of locations after inlining. In Figure C.2, we show the types of properties we consider for each case study, following the notation in Section 3.3.2. .

We discuss the network clients, randomised algorithms, pGCL programs and PRISM programs in more detail in Section C.1, C.2, C.3 and C.4, respectively. For each case study, we will provide a brief description of the program as well as the parameter space and the properties that we consider.

[1] The source code for each case study is available from http://www.prismmodelchecker.org/qprover/.

	LOC	NODES
PING	1,091	291
TFTP	1,018	514
NTP	1,185	1,404
FREI	94	82
HERM	94	40
MGALE	80	32
AMP	70	32
FAC	67	28
BRP	141	94
ZERO	59	45
CONS	58	77

FIGURE C.1: The number of lines of code (LOC) and control-flow locations (NODES) for each case study.

	A	B	C	D	A+	B+
PING	$Prob^+$	$Prob^+$	$Cost^+$	$Prob^+$	–	–
TFTP	$Prob^+$	$Prob^+$	$Cost^+$	–	–	–
NTP	$Prob^+$	$Prob^+$	$Cost^+$	–	–	–
FREI	$Prob^+$	$Prob^-$	–	–	–	–
HERM	$Prob^-$	$Cost^+$	$Prob^+$	$Prob^-$	–	–
MGALE	$Prob^+$	$Prob^-$	$Cost^+$	–	–	$Prob^+$
AMP	$Prob^-$	$Prob^-$	$Prob^+$	–	$Prob^+$	$Prob^+$
FAC	$Prob^-$	$Cost^+$	–	–	–	–
BRP	$Prob^+$	$Prob^+$	$Prob^-$	–	–	–
ZERO	$Prob^-$	$Cost^+$	–	–	–	–
CONS	$Prob^-$	$Prob^-$	–	–	–	–

FIGURE C.2: We show for each case study the types of properties that we consider. Properties are marked with labels A, B, C, D (and A+, B+ in Chapter 7).

C.1 Network Programs

In this section we describe our three main case studies: PING, TFTP and NTP. All three of these programs are real network clients of approximately 1,000 lines of code. These programs feature complex ANSI-C programming constructs such as structs, functions, arrays, pointers, function pointers.

PING

This program is a familiar network utility that is often used to establish connectivity in a network. Our source code is based on the PING client found in GNU InetUtils 1.5 that is widely employed in many LINUX distributions. Essentially, the protocol broadcasts an ICMP packet to a specific host a number of times and waits for a reply packet. For the purposes of model checking we replaced the kernel call that opens a socket with a probabilistic choice such that this function fails with probability $\frac{1}{100}$ and we replaced the kernel call for sending a packet with a probabilistic choice such that sending a packet fails with probability $\frac{2}{25}$. We replaced the remaining kernel calls with non-deterministic choices.

Parameters The number of ICMP requests that is sent (0 meaning an infinite amount).

Properties We check the following properties of this program:

A "maximum probability of receiving replies to *none* of the ICMP requests."

B "maximum probability of receiving replies to *some* but not *all* ICMP requests."

C "maximum expected number of ICMP requests until one reply is received."

D "maximum probability of receiving replies to *all* ICMP requests."

A File Transfer Protocol (TFTP)

This case study is an implementation of a client of a file transfer protocol (TFTP) based on the client TFTP-HPA 0.48. The protocol starts by establishing what is called a *write request*. This involves the client and host both sending one packet. Then, the protocol sends some file data over the network. When the host receives a packet he will send back an acknowledgement. The protocol has some simple mechanisms for dealing with packet loss. For the purposes of verification we replaced the kernel functions used to send and receive packets with stubs that fail with probability $\frac{1}{5}$.

Parameters None.

Properties We check the following properties of this program:

A "maximum probability of establishing a write request."

B "maximum probability of sending *some* file data."

C "maximum expected amount of data packets that is sent before a timeout occurs."

Network Time Protocol (NTP)

Our final network client is an implementation of the network time protocol (NTP). The program is based on the client implementation "ntpclient 365". The aim of the NTP protocol is to synchronise the system time of a client with a reference time on a host system. It does this by sending a number of packets over an unreliable connection and waiting for replies from the host. For the purposes of model checking we replaced the kernel call that is used to receive packets from the host with a stub that fails with probability $\frac{2}{25}$.

Parameters The number of probes that is sent (0 meaning an infinite amount).

A "maximum probability of failing to receive a reply."

B "maximum probability of receiving a reply."

C "maximum expected number of NTP probes sent before a reply packet is received or the program terminates."

C.2 Randomized Algorithms

In this section we consider two randomised algorithms: namely Freivald's algorithm and Herman's self-stabilisation protocol [Her90].

Freivald's Algorithm (FREI)

The idea of Freivald's algorithm is to use a Monte Carlo algorithm to check whether, for three $n \times n$ matrices A, B and C, we have that $AB = C$. The algorithm probabilistically chooses an n-vector of 0's and 1's, v, and then computes whether $A(Bv) = Cv$ holds. From theoretical results we know that if $AB = C$ then $A(Bv) = Cv$ for every v. Moreover, if $AB \neq C$, then it is guaranteed that $A(Bv) \neq Cv$ with probability $\frac{1}{2}$ or more. The motivation for using randomisation is that checking $A(Bv) = Cv$ is computationally much less costly then checking whether $AB = C$. We consider an implementation of this algorithm with 2×2 matrices.

Parameters The number of bits for each data element of A, B and C.

Properties We check the following properties of this program:

A "maximum probability of not detecting $AB \neq C$."

B "minimum probability of not detecting $AB \neq C$."

Herman's Self-stabilisation Protocol (HERM)

Our second randomised protocol is Hermans self-stabilisation protocol [Her90]. The idea is that there are a number of nodes in a ring topology. Depending on the local state of these nodes a node has a token or not. A stable state in the ring is a state where only

one node has a token. The self-stabilisation protocol executes in rounds. In each round, the synchronously processes update their local state with some probabilistic choice. This update directly affects the way tokens are distributed. We use a sequentialised version of this protocol taken from the APEX tool [LMOW08]. The APEX case study is, in turn, a sequentialised version of a PRISM model.

Parameters The number of nodes.

Properties We check the following properties of this program:

A "minimum probability of terminating in a stable state."

B "maximum expected number of rounds."

C "maximum probability of terminating in an unstable state."

D "minimum probability of terminating."

C.3 pGCL Case Studies

These probabilistic programs are from [MM05] and have been translated from a probabilistic guarded command language (pGCL). The characteristics of these programs is that they are very small and use a very small subset of ANSI-C but, due to the use of probabilistic choice in these models, they are not simple to verify.

Martingale (MGALE)

This case study concerns the behaviour of a gambler in a casino and originates from [MM05, Section 2.5.1].[2] The rules of the casino are that the gambler may bet any amount of his money after which the casino will either return double his bet (with probability $\frac{1}{2}$) or nothing (with probability $\frac{1}{2}$). The gambler decides on what he believes to be an infallible strategy and first bets \$1, and, if he loses, he bets \$2, \$4, \$8, etc. This way, as soon as the gambler wins, he is sure to make a profit. Of course, the gambler has only finite resources and his chance of actually making a profit depends on his initial capital.

[2] Note that our program is, in fact, the one in Figure 2.10.1 of [MM05] and not the variant in Figure 2.5.1 — i.e. our program terminates when the gambler runs out of money.

Parameters The initial capital of the gambler.

Properties We check the following properties of this program:

A "maximum probability of eventually making a profit."

B "minimum probability of eventually finishing gambling."

C "maximum expected number of bets."

In Chapter 7, we also consider the property:

B+ "maximum probability of eventually finishing gambling."

Amplification (AMP)

This example originates from [MM05, Section 2.5.2] and describes a typical Monte Carlo algorithm. The idea is that this Monte-Carlo algorithm attempts to decide wether some hypothesis `Q` is true or false by running a test repeatedly inside a loop — whether the hypothesis `Q` holds non-deterministically decided at the beginning of the program. If `Q` is false, then, in our program, the test will detect this with probability $\frac{1}{4}$. However, if `Q` if true then we have no way of establishing this via our test.

Parameters The number of tests.

Properties We check the following properties of this program:

A "minimum probability of establishing the correct value for `Q`."

B "minimum probability of termination."

C "maximum probability of terminating with the incorrect value for `Q`."

In Chapter 7, we also consider the variants:

A+ "maximum probability of establishing the correct value for `Q`."

B+ "maximum probability of termination."

Faulty Factorial (FAC)

Our final program originates from [MM05, Section 2.5.3] and describes a procedure that computes the factorial of a natural number `n` iteratively. The idea of this case study is to

introduce a small probability of an iteration of this procedure failing to execute correctly. We let an iteration fail to execute correctly with probability $\frac{1}{100}$. We consider a slightly different failure than that of [MM05] and do not replace an decrement with a increment, but instead replace it with a "skip" command.

Parameters The number `n` to compute the factorial of.

Properties We check the following properties of this program:

A "minimum probability of termination."

B "maximum expected number of loop iterations until termination."

C.4 PRISM Case Studies

Our final set of programs are PRISM case studies. They have been translated and sequentialised into ANSI-C from a simple probabilistic guarded command language. These programs are relatively small in terms of specification, but exhibit some non-trivial probabilistic behaviour.[3]

Bounded Retransmission Protocol (BRP)

The bounded retransmission protocol (BRP) is a protocol for sending a number of chunks of data from a client to a server with some fault tolerance [DJJL01]. The client sends each chunk in sequence and waits for an acknowledgement from the server. The basic idea is that, if a chunk fails to transmit or its acknowledgement is lost, then the chunk is resent at most MAX times. We consider a sequentialised PRISM model of this protocol for which MAX=4.

Parameters The number of chunks to send.

Properties We check the following properties of this program:

A "maximum probability that the receiver does not receive *any* chunk."

B "maximum probability that eventually the sender reports an uncertainty on the

[3] For more information on these models we refer to `http://www.prismmodelchecker.org/`.

success of the transmission."

C "minimum probability that eventually the sender reports a successful transmission."

We note that BRP A and BRP B are "property 4" and "property 2" in [DJJL01].

IPv4 ZeroConf Protocol (ZERO)

This program is a sequentialised version of the IPv4 ZeroConf Protocol [CAG05]. The aim of the ZeroConf protocol is to make networks self-configuring. In particular, the focus of this model is on the selection of an IP address. The idea is that a network client picks an IP address probabilistically and then sends a number of probes into the network. It then awaits any responses from other nodes in the network that indicate the address is already in use. We consider a sequentialised version of a PRISM model of this protocol.

Parameters The number of probes that are sent.

Properties We check the following properties of this program:

A "minimum probability of eventually configuring with a fresh IP address."

B "maximum expected number of probes that are sent."

Consensus Protocol (CONS)

The idea of the consensus protocol (CONS) is for N processes to agree one out of two outcomes [AH90]. This outcome is chosen through a random walk from 0. At each point in time a process can request to flip a coin. If the coin is heads then the counter is incremented, if it is tails it is decremented. For some parameter, K, we choose one outcome if the counter reaches the value $K \cdot N$ or the other outcome if the counter reaches the value $-K \cdot N$. We consider a sequentialised version of the PRISM model of this protocol for which $N = 1$.

Parameters The value of the constant K.

Properties We check the following properties of this program:

A "minimum probability that all processes that enter the coin protocol also leave the protocol."

B "minimum probability that all processes that enter the protocol leave with all coins equal to 1."