Stony Brook University

Verification of Probabilistic Branching Time Systems

A Dissertation presented

by

Andrey Gorlin

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

December 2016

Stony Brook University

The Graduate School

Andrey Gorlin

We, the dissertation committee for the above candidate for the

Doctor of Philosophy degree, hereby recommend

acceptance of this dissertation

C. R. Ramakrishnan - Dissertation Advisor
Professor, Dept. of Computer Science

Scott D. Stoller - Chairperson of Defense
Professor, Dept. of Computer Science

Scott A. Smolka
Professor, Dept. of Computer Science

W. Rance Cleaveland
Professor, Dept. of Computer Science, University of Maryland

This dissertation is accepted by the Graduate School

Charles Taber
Dean of the Graduate School

Abstract of the Dissertation

Verification of Probabilistic Branching Time Systems

by

Andrey Gorlin

Doctor of Philosophy

in

Computer Science

Stony Brook University

2016

This thesis deals with verification of complex systems, which involve probabilistic choices. In total, we explore three interrelated problems. First, we explore probabilistic extensions of μ-calculus. GPL extends μ-calculus by having all the probabilistic choices made first; this keeps the model checking procedure decidable for any property. We extend GPL to a logic, XPL, which is undecidable, in general. We define a syntactic property of an XPL formula, separability, as a sufficient condition for model checking. Second, we can frame the problem of probabilistic model checking as query evaluation over probabilistic logic programs. We have developed an inference algorithm, PIP, using tabled logic programming, which is sufficiently powerful to verify GPL and separable XPL properties. PIP uses finite generative structures, called FEDs, to represent families of explanations. Finally, we explore an alternative paradigm for verification of temporal models: compositional or partial model checking. In particular, we employ a technique called quotienting, where we take a μ-calculus formula and a process and return another formula that must be satisfied by the remainder of the system.

Table of Contents

1 Introduction **1**
- 1.1 Transition Systems and Model Checking . . . 2
  - 1.1.1 Temporal Logics . . . 3
  - 1.1.2 Linear-time Probabilistic Logics . . . 5
  - 1.1.3 Branching Time in the Probabilistic Domain . . . 5
- 1.2 Logic Programming . . . 6
- 1.3 Compositional Model Checking . . . 7
  - 1.3.1 Quotienting . . . 7
  - 1.3.2 PRISM Model Checker . . . 8
- 1.4 Thesis Outline . . . 8

2 Probabilistic Systems **9**
- 2.1 Reactive Probabilistic LTSs . . . 9
- 2.2 Generalized Probabilistic Logic . . . 11
  - 2.2.1 GPL Syntax . . . 12
  - 2.2.2 GPL Semantics . . . 13
- 2.3 GPL Model Checking . . . 15
- 2.4 Recursive Markov Chains . . . 18
- 2.5 Probabilistic Polynomial Systems . . . 20
- 2.6 The Interpretation of Branching Time . . . 23
  - 2.6.1 Branching Processes . . . 23
  - 2.6.2 PTTL . . . 25
  - 2.6.3 Additional Interpretations . . . 25

3 Linear Nondeterminism in Probabilistic Systems **26**
- 3.1 Probabilistic Labeled Transition Systems . . . 27
- 3.2 XPL . . . 30
  - 3.2.1 XPL Syntax . . . 30
  - 3.2.2 XPL Semantics . . . 30

3.3 XPL Model Checking 31
3.3.1 Separability of Fuzzy Formulae 32
3.3.2 Dependency Graph 35
3.4 Encoding Other Model Checking Problems 40
3.4.1 Encoding PCTL* over MDPs 40
3.4.2 Encoding of RMDP Termination 41
3.4.3 PTTL and Branching Processes 44
3.5 Conclusion and Related Work 45

4 Model Checking with Logic Programming 48
4.1 Related Work 51
4.2 Preliminaries 53
4.3 The Inference Procedure PIP 53
4.3.1 Representing Explanations 54
4.3.2 Factored Explanation Diagrams 57
4.3.3 Nondeterminism and Merge 61
4.3.4 Computing Probabilities from FEDs 64
4.4 Applications 66
4.5 Experimental Results 68
4.6 Conclusions 72

5 Partial pLμ Model Checking 73
5.1 pLμ 74
5.1.1 pLμ Syntax 74
5.1.2 pLμ Semantics 75
5.1.3 Markov Branching Plays 76
5.1.4 Partial Model Checking and pLμ 76
5.2 Probabilistic Model 77
5.2.1 Process Algebra 77
5.3 Quotienting 79
5.3.1 Probability Function 79
5.3.2 Quotienting Rules 80
5.4 Read Operator 85
5.4.1 Effect on Processes 86
5.5 Case Studies 88
5.5.1 Rabin's Choice Coordination Problem Encoding 89
5.5.2 ECo-MAC Encoding 91
5.6 Conclusion 92

6 Conclusion **93**

List of Figures

2.1 RPLTS Example 11
2.2 D-Tree Example 12
2.3 Dependency graph $\Pr(s_1, \psi)$ 18
2.4 RMC Example 19

3.1 PLTS Example 29
3.2 Example PLTS with nondeterministic choice on "a" 32
3.3 Dependency graph $\mathsf{Dg}(s_1, \psi)$ 40
3.4 Example RMDP with Call, Return, and Exit edges added to A. 44

4.1 (a) Example Markov chain; (b) PRISM encoding of transitions in the chain. 49
4.2 FEDs for Example 4.2 62
4.3 Set of equations generated from the set of FEDs of Example 4.4 65
4.4 Model checker for a fragment of PCTL 67
4.5 Fragment of a model checker for fuzzy formulae in GPL . . . 69
4.6 Performance of PIP on PRISM Programs 70
4.7 Performance of PCTL model checking using PIP and the PRISM model checker for Synchronous Leader Election protocol . . . 72

List of Tables

2.1 GPL semantics: fuzzy formulae 14
2.2 GPL semantics: state formulae 14
2.3 Non-action nodes 16

3.1 XPL semantics: state formulae 31

5.1 pLμ semantics 75
5.2 Quotienting rules 81
5.3 Additional Quotienting Rules 88

List of Abbreviations

BDD Binary Decision Diagram. 7, 8, 50, 57, 60, 63, 69–72
BMDP Branching MDP. 27, 44–46, 93
BP Branching Process. 23–25, 44–46, 93
BSSG Branching Simple Stochastic Game. 44, 45, 76

CCS Calculus of Communicating Systems. 7, 48, 73, 77, 94
CSL Continuous Stochastic Logic. 5, 8
CTL Computation Tree Logic. 3, 4, 23, 25, 45

d-tree deterministic tree. 10–12, 16, 17, 25, 28, 29, 32, 41, 74, 75, 93
DAG directed acyclic graph. 57, 58, 61
DCG Definite Clause Grammar. 55–57
DNF disjunctive normal form. 34, 36, 38

ENF Explanation Normal Form. 60, 63

FED Factored Explanation Diagram. 7, 57–66, 69–71, 94

GFP greatest fixed point. 21, 23, 39, 52, 68
GPL Generalized Probabilistic Logic. 1, 2, 6–9, 11–13, 15, 19–22, 25, 26, 30–32, 35, 37–39, 45–48, 51–53, 64, 67, 68, 73–75, 93, 94

HMM Hidden Markov Model. 70, 71

ICL Independent Choice Logic. 52

LFP least fixed point. 21–23, 39, 68, 74
LHS left-hand side. 56, 57
LP logic programming. 48, 53
LPAD Logic Programs with Annotated Disjunctions. 50, 52
LTL Linear-time Temporal Logic. 3–5, 9, 20, 52
LTS Labeled Transition System. 1, 4, 7

MBP Markov Branching Play. 6, 23, 74, 76
MDP Markov Decision Process. 1–3, 27, 41, 46, 52, 73, 76
MPS Monotone Polynomial System. 21

NSM Nested State Machine. 43

PCTL Probabilistic Real-time CTL. 5, 6, 8, 9, 20, 24, 45, 51, 52, 66–69, 71, 94
PIP "Probabilistic Inference Plus". 2, 6–8, 48, 51–53, 66, 69–72
PITA Probabilistic Inference with Tabling and Answer Subsumption. 49, 50, 58, 69–71
PLC Probabilistic Left Corner. 71
PLP Probabilistic Logic Programming. 6, 7, 48, 51, 52, 54–56, 94
PLTS Probabilistic LTS. 1–3, 6–9, 26–31, 35–39, 41–43, 45, 53, 69, 73–79, 92, 93
PPS Probabilistic Polynomial System. 9, 21–23, 25, 46
PRISM Programming In Statistical Modeling. 6–8, 48–50, 52–54, 59, 61, 69–71
PTTL Probabilistic Tree Temporal Logic. 24, 25, 45

RHS right-hand side. 56, 57
RMC Recursive Markov Chain. 3, 8, 9, 18–21, 23, 24, 26, 41, 43, 46, 51–53, 74, 93
RMDP Recursive MDP. 1, 27, 41–46, 61, 62, 73, 93
RPLTS Reactive Probabilistic LTS. 3, 6, 9–13, 15, 17, 19, 20, 24–30, 43–45, 51, 53, 67, 93
RSSG Recursive Simple Stochastic Game. 1, 41, 42, 45, 76

SM stackless and memoryless. 42, 45
SNF Simple Normal Form. 21, 22
SRL Statistical Relational Learning. 52

XPL Extended Probabilistic Logic. 1, 2, 6, 8, 26, 27, 30–32, 36, 38–40, 45–48, 53, 64, 69, 73–75, 94
XPS Extended Polynomial System. 22, 23

Chapter 1

Introduction

This thesis deals with verification of complex systems. In particular, a key aspect of the systems in this thesis is that they are probabilistic. This means that at least some decisions are modeled as probabilistic choices, possibly as a result of uncertainty and lack of information, or because the underlying system is indeed random in some aspect. The presence of probability additionally means that, when formulating questions about a system, we will generally not seek to guarantee that a particular property holds; instead, we will estimate its likelihood.

Another important aspect of the systems in this thesis is that they are temporal. When analyzing temporal systems, there are two general ways to consider system executions. We can look at traces, *i.e.*, paths through the system, or trees. Trees may arise in several different ways, of which a rather interesting one is via a branching nondeterministic choice, where a tree represents a single probabilistic outcome. Generalized Probabilistic Logic (GPL) [CIN05] is a branching-time logic over such trees as the outcomes. In Chapter 2, we provide an overview of GPL as originally presented in [CIN05]. Throughout this thesis, it serves a primary role in our discussion, as we will both extend GPL directly and apply our logic programming advances to it. In total, we explore three interrelated problems.

First, probabilistic choices do not entirely replace nondeterminism in Labeled Transition Systems (LTSs). However, for model checking purposes, the nondeterministic systems typically considered have been limited to Markov Decision Processes (MDPs) [Ste09]. In this thesis, we analyze the model checking of Probabilistic LTSs (PLTSs) [Seg95, Mio12]. In part, this is motivated by the results on Recursive MDPs (RMDPs) and Recursive Simple Stochastic Games (RSSGs) [EY15]. Some of the properties in the resulting logic, which we call Extended Probabilistic Logic (XPL), become undecidable, but we describe a wide class of properties that we can verify. We

elaborate on this in Chapter 3.

Second, logic programming has previously been applied to model checking in the non-probabilistic domain. Both fields have been separately extended to support probability in some form, and it is a challenging problem to reconcile the two. In particular, the demands of model checking are different from the other problems to which probabilistic logic programming has been applied. Our inference algorithm, "Probabilistic Inference Plus" (PIP), is a first step towards this goal; we discuss it in Chapter 4. With PIP, we can model check GPL, and we explore how it handles (linear) nondeterminism, which could be used to support a decidable fragment of XPL and the model checking of PCTL* [Bai98] over MDPs.

Finally, an alternative paradigm for verification for temporal models is compositional or partial model checking [And95, BR06]. In this case, we can view the system as modeled by separate modules. While a straightforward approach would be to compose them to get the whole system, which could then be model checked, this may also be intractable. Instead, we seek to verify them separately, with a technique called *quotienting*. The probabilistic branching time paradigm of GPL is not easily amenable to quotienting, due to its semantics for the propositional connectives (*i.e.*, to handle a conjunction, we cannot necessarily handle the conjuncts separately). Instead, we apply pLμ, in which the semantics of a formula is a value in $[0, 1]$, and the value of a binary operator is some function of the values of the operand (and extensions admit binary operators other than standard conjunction and disjunction) [Mio11, Mio12]. Our model remains a PLTS, and we use a simple process algebra to encode PLTSs to facilitate composition and quotienting. We discuss our results in Chapter 5.

1.1 Transition Systems and Model Checking

Our models will generally be transition systems. Transition systems have states; there is a set of propositions, Prop, a subset of which holds at each state; and there is some notion of changing the state of the system, via the transition relation. The transition relation may be probabilistic, yielding a distribution of possible states to be reached for a particular transition. Transitions may also be labeled by actions (drawn from the set Act), allowing for composition synchronized on matching actions, or for a system to have a branching nature, where different actions correspond to independent branches. Markov chains [Ste09] are a natural basic probabilistic model.

As the transition systems outlined above, a Markov chain has states and transitions (but no action labels), and the transitions from a state form a distribution, with their probabilities summing to 1. When residency time, *i.e.*, the amount of time that a system spends at a state before moving to a different one, is important, we may also distinguish between discrete- and continuous-time Markov chains. In this thesis, we will generally view time as discrete (or unimportant).

A PLTS is a general model allowing all of the above. Additionally, probabilistic transitions may not replace *all* of the nondeterministic choices; nondeterminism may be present in probabilistic systems in two forms. The first kind, which we will call *linear* nondeterminism, is resolved, potentially by a scheduler (or a strategy), before the probabilistic choices are made. MDPs are an extension of Markov chains with linear nondeterminism. The second kind, which we will call *branching*, is resolved after the probabilistic choices. In Chapter 2, we discuss a version of a PLTS where all nondeterminism is branching (called a Reactive Probabilistic LTS (RPLTS) due to [CIN05]), and discuss Recursive Markov Chains (RMCs) [EY09], where branching nondeterminism is intrinsic to the system. We deal with full PLTSs in Chapter 3.

1.1.1 Temporal Logics

Given a transition system, we will want to state and verify a property of the system. Properties are *temporal*, describing not only what currently holds, but how the system may evolve over time. Many temporal properties can be expressed with two building blocks: a unary *next* operator, denoted by $\mathsf{X}\phi$, and a binary *until* operator, denoted by $\phi_1 \mathsf{U} \phi_2$. In particular, U is a fixed-point operator, which can describe properties about, *e.g.*, some proposition becoming true at an arbitrary point in the future or remaining true henceforth.

First, we mention three non-probabilistic logics (a more detailed description may be found in, *e.g.*, [HR04, Chapter 3]). Linear-time Temporal Logic (LTL) [IN96] is built with the X and U operators, along with the typical propositional connectives. A minimal syntax for LTL, with $A \in \mathsf{Prop}$, may be given as follows:

$$\psi ::= A \mid \neg\psi \mid \psi \wedge \psi \mid \mathsf{X}\psi \mid \psi \mathsf{U} \psi.$$

Linear-time properties are associated with paths, *i.e.*, the sequences of states that a system is in as time advances. Meanwhile, Computation Tree Logic

(CTL) [CES86] is a branching-time logic. It allows quantification over paths, either universally or existentially, with the A and E quantifiers, respectively. The syntax for CTL may be given as follows:

$$\phi ::= A \mid \neg\phi \mid \phi \wedge \phi \mid \mathsf{AX}\phi \mid \mathsf{EX}\phi \mid \mathsf{A}[\phi \mathsf{U} \phi] \mid \mathsf{E}[\phi \mathsf{U} \phi].$$

Despite the superficial similarity, the propositional connectives are quite different in LTL and CTL, highlighting the distinction between linear and branching time. For instance, $\mathsf{X}\psi_1 \vee \mathsf{X}\psi_2 \equiv \mathsf{X}(\psi_1 \vee \psi_2)$ in LTL, but, in CTL, the formula $\mathsf{AX}\phi_1 \vee \mathsf{AX}\phi_2$ is stronger than $\mathsf{AX}(\phi_1 \vee \phi_2)$.

An alternative way to write the syntax for CTL splits its formulae into two types, *state formulae* ϕ and *path formulae* ψ.

$$\begin{aligned} \phi &::= A \mid \neg\phi \mid \phi \wedge \phi \mid \mathsf{A}[\psi] \mid \mathsf{E}[\psi], \\ \psi &::= \mathsf{X}\phi \mid \phi \mathsf{U} \phi. \end{aligned}$$

Note that the syntax for the path formulae of CTL makes them a subset of possible LTL formulae. CTL* [EH86] extends this syntax by allowing any LTL formula as a path formula, at the expense of additional computational complexity for verification. The syntax for CTL* may be given as follows:

$$\begin{aligned} \phi &::= A \mid \neg\phi \mid \phi \wedge \phi \mid \mathsf{A}[\psi] \mid \mathsf{E}[\psi], \\ \psi &::= \phi \mid \neg\psi \mid \psi \wedge \psi \mid \mathsf{X}\psi \mid \psi \mathsf{U} \psi. \end{aligned}$$

While X and U are high-level building blocks, modal μ-calculus [Koz83], which we will call Lμ due to [Mio12], offers lower-level constructs, with least and greatest fixed points (U is a particular instance of a least fixed point). We define it over LTSs; additionally, we use a set of propositional *variables*, Var, and a set of action labels, Act. A minimal syntax for modal μ-calculus, with $X \in \mathsf{Var}$ and $a \in \mathsf{Act}$, may be given as follows:

$$\psi ::= X \mid \psi \wedge \psi \mid \psi \vee \psi \mid [a]\psi \mid \langle a \rangle \psi \mid \mu X.\psi \mid \nu X.\psi.$$

Note that negation ($\neg\psi$) is not part of the syntax, which is convenient for maintaining monotonicity of the fixed points; the variables can also serve the role of atomic propositions. Since each operator has its dual in the syntax, it remains straightforward to write a formula $\mathsf{neg}(\psi)$ representing the negation of a (closed) formula ψ. The modal operators $[a]\psi$ and $\langle a \rangle \psi$ are loosely analogous to $\mathsf{AX}\psi$ and $\mathsf{EX}\psi$, respectively. Additionally, any CTL* formula can be translated to μ-calculus; note that the U operator can be written as a fixed point, since $\psi_1 \mathsf{U} \psi_2 \equiv \psi_1 \vee (\psi_2 \wedge \mathsf{X}(\psi_1 \mathsf{U} \psi_2))$.

1.1.2 Linear-time Probabilistic Logics

We can also model check LTL over Markov processes. The evaluation of an LTL formula becomes the probability measure of all the paths on which it holds (if the process also has linear nondeterminism, a scheduler maximizing the measure may be assumed). Notably, a probabilistic value of 1 does not necessarily require that the property hold on all paths, as a single infinite path will typically have zero measure. To extend logics with state formulae, we replace the operators $\mathsf{A}[\psi]$ and $\mathsf{E}[\psi]$ with $\mathsf{Pr}_{\geq p}\psi$ and $\mathsf{Pr}_{>p}\psi$. Thus, the syntax of Probabilistic Real-time CTL (PCTL) [HJ94], where $t \in \mathbb{N}$ and $p \in [0, 1]$, may be written as follows:

$$\phi ::= A \mid \neg\phi \mid \phi \wedge \phi \mid \mathsf{Pr}_{\geq p}\psi \mid \mathsf{Pr}_{>p}\psi,$$
$$\psi ::= \phi \mathsf{U}^{\leq t} \phi \mid \phi \mathsf{U} \phi.$$

We note briefly that PCTL has a bounded-time U operator (which could be considered syntactic sugar in LTL) and omits the X operator. With bounded-time properties, the nature of time is more significant, and time may be discrete or continuous[1]. However, we will not focus on bounded-time properties in this thesis.

The key to model-checking PCTL is the computation of the probabilistic value of an *until* formula; thus, we can view PCTL essentially as a linear-time probabilistic logic. Similarly, we can define PCTL* [Bai98], with the PCTL syntax for state formulae and the LTL syntax for the path formulae.

1.1.3 Branching Time in the Probabilistic Domain

Throughout this thesis, we will reserve the meaning of *probabilistic branching time* for probabilistic systems with branching nondeterminism. Though we do not focus on bounded-time properties, probabilistic branching time contains other emergent properties. For instance, a qualitative query, *i.e.*, whether a property holds with probability 1 or not, may depend on the specific values in the transition relation, even in a finite system. This result may be intuitively understood with processes that either branch or terminate and considering the extinction probability: if termination is more likely than the creation of an additional process, extinction is guaranteed; however, if the process population is expected to grow over time, another possible outcome is a population growing without bound.

[1] A continuous-time analogue to PCTL is Continuous Stochastic Logic (CSL) [BHHK03]

GPL [CIN05] is a probabilistic branching-time logic over RPLTSs. It has state formulae, with syntax like PCTL's, including probabilistic operators, and essentially full Lμ as the analogue of the path formulae; they are called *fuzzy* formulae, as they are defined not over paths, but deterministic *trees* (d-trees). We discuss GPL and RPLTSs in detail in Chapter 2; we extend GPL to XPL, a logic over PLTSs, in Chapter 3; and we describe an implementation of a general inference algorithm, PIP, capable of model checking GPL and XPL in Chapter 4.

Other probabilistic extensions of μ-calculus exist, as well. PLTSs are not necessarily interpreted as probabilistic branching-time systems, and can instead be seen as having only linear nondeterminism. Then, pLμ [Mio12, Mio11] is a logic over PLTSs with syntax identical to Lμ. Additionally, it supports a probabilistic branching-time extension via additional propositional connectives of independent product and coproduct, yielding the logic pL$\mu^{\odot}$ [Mio11]; then, the systems arising from the combination of the pL$\mu^{\odot}$ formula with the PLTS have the probabilistic branching-time nature, *branching plays* being similar to d-trees and Markov Branching Plays (MBPs) to RPLTSs [Mio12]. We discuss pLμ further in Chapter 5.

1.2 Logic Programming

Our main practical contribution is a probabilistic inference algorithm implemented via logic programming, which we discuss in more detail in Chapter 4. Logic programming is a natural paradigm for model checking in the non-probabilistic case, and, like model checking, it has been extended to the probabilistic domain, with the Programming In Statistical Modeling (PRISM) system [SK97]. However, while the *distribution semantics* in PRISM appears sufficient for probabilistic model checking, the existing implementations of Probabilistic Logic Programming (PLP), including PRISM, were inadequate for this purpose.

Logic programming typically involves a version of Prolog [NM95]. Prolog is a declarative programming language, which, at its most basic, supports first-order logic and consists of rules and an engine to answer queries, with universally quantified variables. PRISM adds a special predicate called msw, which corresponds to a probabilistic transition [SK97]. This is sufficient to encode probabilistic systems and pose model checking queries. However, prior PLP implementations typically made assumptions, such as mutual exclusion or finiteness of explanations, which may be simultaneously violated

in a model checking query.

Since this was not an issue with PRISM's distribution semantics [SK97], we devised an inference algorithm, PIP, capable of probabilistic model checking [GRS12]. This involves a new data structure, a generalization of Binary Decision Diagrams (BDDs), which we called Factored Explanation Diagrams (FEDs). Additionally, the distribution semantics means that the nondeterminism in PLP is branching, which allows for a natural encoding of GPL. We can also support a limited form of linear nondeterminism.

1.3 Compositional Model Checking

In many cases, a system is made from a number of largely independent components, which synchronize in a precise manner. Model checking the system as a whole, however, may be intractable. Then, the system may be reduced to an equivalent one via, *e.g.*, symmetry reduction [CTV06] or partial order reduction [HKQ11]. Another approach is partial model checking, where we do not build the whole system at all. In Chapter 5, we consider the possibility of applying partial model checking to probabilistic systems.

1.3.1 Quotienting

One way to model LTSs compositionally is with the Calculus of Communicating Systems (CCS) [Mil89]. CCS has also been extended to the probabilistic domain [JYL01]. We use a simple process algebra to represent PLTSs and employ a verification approach called *quotienting* [And95]. If a property (for the whole system) is expressed in modal μ-calculus [Koz83], then we can quotient out a component process's contribution to satisfying the formula, yielding a new μ-calculus formula to be satisfied by the remaining components. If we may have an arbitrary number of identical processes, then quotienting may reach a fixed point, allowing *parameterized* partial model checking [BR06].

The nature of pLμ [Mio12] makes it a good candidate for probabilistic partial model checking. However, while a process choice in the non-probabilistic case could be turned into a disjunction in the μ-calculus formula, the standard disjunction of pLμ cannot support a quotiented probabilistic choice; this is modeled instead by a convex combination operator, $+_\lambda$. A feature of pLμ is that it supports as extensions additional binary operations other than standard conjunction and disjunction, including $+_\lambda$. We show that, in some cases, we may reach what would be a non-probabilistic fixed point in

the formula, where the only change from quotienting out another process is on the values of λ in the $+_\lambda$ operators.

1.3.2 PRISM Model Checker

PRISM is a probabilistic model checker, and it serves as an example of supporting a higher-level specification for probabilistic models [KNP11][2]. It has Markov processes [Ste09] as its low-level models, reactive modules [AH99] for its specifications, and PCTL/PCTL* and CSL as its logics, depending on the nature of time in the model. PRISM supports a BDD representation for the models, which produces compact structures in many case studies. In this thesis, we also analyze our own models for several of these case studies, in Sections 4.5 and 5.5.

1.4 Thesis Outline

The rest of this thesis proceeds as follows. We review GPL in Chapter 2, along with RMCs [EY09]. We extend GPL to XPL, a logic supporting PLTSs, systems with linear nondeterminism, in Chapter 3. Chapter 4 introduces a general probabilistic inference algorithm, PIP, which is based on tabled logic programming [SW+12]. It includes the ability to model check GPL and a fragment of XPL. In Chapter 5, we apply and extend pLμ [Mio12] to support partial model checking. Our conclusion is in Chapter 6.

[2]Note that the PRISM system is distinct from the PRISM model checker.

Chapter 2

Probabilistic Systems

A Probabilistic LTS (PLTS) [Seg95, Mio12] is more expressive than a Markov chain [Ste09]; PLTSs integral to this thesis. To find the probability of a path formula in PCTL [HJ94], a logic over Markov processes, we can solve a linear system of equations, which may be represented as a matrix. The models covered in this chapter result in nonlinear systems; in particular, we get Probabilistic Polynomial Systems (PPSs) [EY09, ESY12b]. The nonlinearity arises as a treatment of probabilistic branching time; a PCTL path formula may be seen as coming from a subset of LTL formulae [IN96].

In Section 2.1, we discuss RPLTSs, in which there is no linear nondeterminism. In Section 2.2, we describe GPL [CIN05], including its syntax and semantics, followed by an outline of the model checking algorithm in Section 2.3. We discuss RMCs [EY09] in Section 2.4. The polynomial systems arising in GPL and RMCs are covered in Section 2.5. Finally, Section 2.6 considers the meaning of probabilistic branching time and its relation to GPL.

2.1 Reactive Probabilistic LTSs

A Reactive Probabilistic LTS (RPLTS) has internal choices made as a reaction to external choices. External choices are represented by differently labeled actions, while internal choices are probabilistic choices labeled with the same action.

Formally, with respect to fixed sets Act and Prop of *actions* and *propositions*, respectively, an RPLTS L is a quadruple (S, δ, P, I) [CIN05, Definition 1], where

- S is a countable set of states;
- $\delta \subseteq S \times \mathsf{Act} \times S$ is the transition relation;

- $P : \delta \to (0,1]$ is the transition probability distribution satisfying:
  - $\forall s \in S. \forall a \in \mathsf{Act}. \sum_{s':(s,a,s')\in\delta} P(s,a,s') \in \{0,1\}$, and
  - $\forall s \in S. \forall a \in \mathsf{Act}. (\exists s'. (s,a,s') \in \delta) \implies \sum_{s':(s,a,s')\in\delta} P(s,a,s') = 1$;
- $I : S \to 2^{\mathsf{Prop}}$ is the *interpretation*, recording the set of propositions that are true at a state.

Given $L = (S, \delta, P, I)$, a *partial computation* is a sequence

$$\sigma = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_n \ ,$$

where for all $0 \le i < n$, $(s_i, a_{i+1}, s_{i+1}) \in \delta$. Also, $\mathsf{fst}(\sigma) = s_0$ and $\mathsf{last}(\sigma) = s_n$. Each transition of a partial computation is labeled with an action $a_i \in \mathsf{Act}$. $\mathcal{C}_L$ refers to the set of all partial computations of L, and $\mathcal{C}_L(s) = \{\sigma \in \mathcal{C}_L \mid \mathsf{fst}(\sigma) = s\}$. *Composition* of partial computations, $\sigma \xrightarrow{a} \sigma'$, represents $s_0 \xrightarrow{a_1} \cdots \xrightarrow{a_n} s_n \xrightarrow{a} s'_0 \xrightarrow{b_1} \cdots \xrightarrow{b_m} s'_m$ if $(s_n, a, s'_0) \in \delta$. σ' is a *prefix* of σ if $\sigma' = s_0 \xrightarrow{a_1} \cdots \xrightarrow{a_i} s_i$ for some $i \le n$.

From a set of partial computations, we can build deterministic trees (d-trees). We often denote a d-tree by the set of paths in the tree. Every d-tree is prefix-closed and deterministic. $T \subseteq \mathcal{C}_L$ is *prefix-closed* if, for every $\sigma \in T$ and σ' a prefix of σ, $\sigma' \in T$. T is *deterministic* if for every $\sigma, \sigma' \in T$ with $\sigma = s_o \xrightarrow{a_1} \cdots \xrightarrow{a_n} s_n \xrightarrow{a} s \cdots$ and $\sigma' = s_0 \xrightarrow{a_1} \cdots \xrightarrow{a_n} s_n \xrightarrow{a'} s' \cdots$, either $a \neq a'$ or $s = s'$, *i.e.*, if a pair of computations share a prefix, the first difference cannot involve transitions labeled by the same action. A d-tree T has a starting state, denoted $\mathsf{root}(T)$; if $s = \mathsf{root}(T)$ then $T \subseteq \mathcal{C}_L(s)$.

Further, $\mathsf{edges}(T) = \{(\sigma, a, \sigma') \mid \sigma, \sigma' \in T \wedge \exists s \in S. \sigma' = \sigma \xrightarrow{a} s\}$. Analogously to the partial computation definitions, $\mathcal{T}_L$ refers to all the d-trees of L, and $\mathcal{T}_L(s) = \{T \in \mathcal{T}_L \mid \mathsf{root}(T) = s\}$. T' is a *prefix* of T if $T' \subseteq T$. $T \xrightarrow{a} T'$ means $T' = \{\sigma \mid \mathsf{root}(T) \xrightarrow{a} \sigma \in T\}$. T is *finite* if $|T| < \infty$, and *maximal* if there exists no d-tree T' with $T \subset T'$. $\mathcal{M}_L$ and $\mathcal{M}_L(s)$ are analogous to $\mathcal{T}_L$ and $\mathcal{T}_L(s)$, but for maximal d-trees.

An *outcome* is a maximal d-tree; this gives the external nondeterminism of RPLTSs the branching nature. Intuitively, the probability of some finite prefix is the product of the probabilities of all the edges. Formally, a *basic cylindrical subset* of $\mathcal{M}_L(s)$ contains all trees sharing a given prefix. Letting

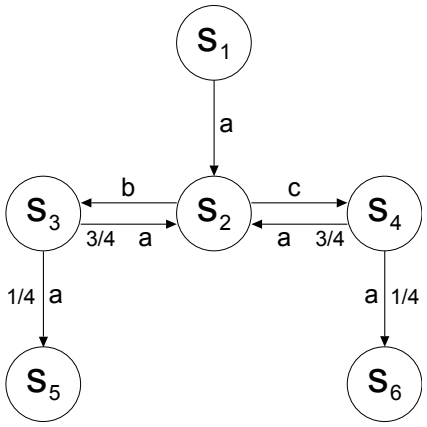


Figure 2.1: RPLTS Example

$s \in S$, and $T \in \mathcal{T}_L(s)$ to be finite, $B_T = \{T' \in \mathcal{M}_L \mid T \subseteq T'\}$. The measure of B_T is:

$$\mathsf{m}(B_T) = \prod_{(\sigma,a,\sigma') \in \mathsf{edges}(T)} P(\mathsf{last}(\sigma), a, \mathsf{last}(\sigma'))$$

From here, a probability measure $\mathsf{m}_s : \mathcal{B}_s \to [0,1]$ on the smallest field of sets $\mathcal{B}_s$ is generated from subsets B_T with $\mathsf{m}_s(B_T) = \mathsf{m}(B_T)$ [CIN05, Definition 8].

Example 2.1 (Sample RPLTS)**.** *Figure 2.1 is an example of a specification [CIN05, Figure 3], where*

- $S = \{s_1, s_2, s_3, s_4, s_5, s_6\}$,
- $\delta = \{(s_1, a, s_2), (s_2, b, s_3), (s_2, c, s_4), (s_3, a, s_2), (s_3, a, s_5),$
  $(s_4, a, s_2), (s_4, a, s_6)\}$,
- $P(s_3, a, s_5) = P(s_4, a, s_6) = \frac{1}{4}, P(s_3, a, s_2) = P(s_4, a, s_2) = \frac{3}{4}$. *For all other transitions* $t \in \delta$, $P(t) = 1$.

An example of an outcome is in Figure 2.2. This is a finite d-tree, which has a probability of $\left(\frac{1}{4}\right)^3 \cdot \frac{3}{4} = \frac{3}{256}$. Infinite outcomes are also possible, and, in this example, they have positive measure.

2.2 Generalized Probabilistic Logic

Generalized Probabilistic Logic (GPL) [CIN05] is a probabilistic branching-time extension of modal μ-calculus [Koz83].

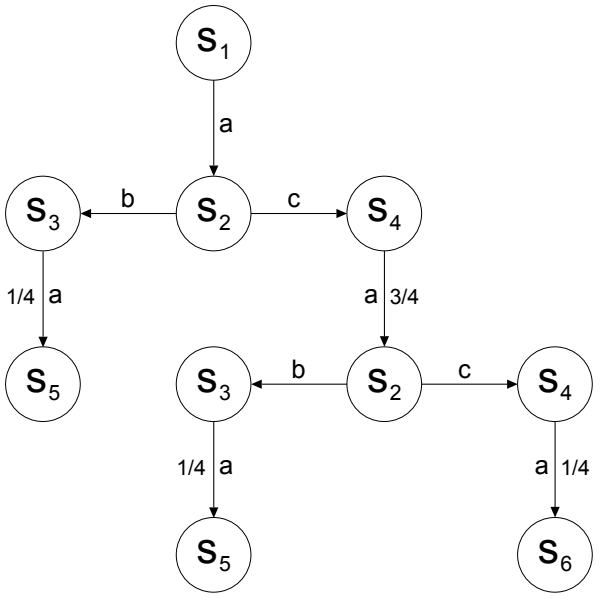


Figure 2.2: D-Tree Example

2.2.1 GPL Syntax

GPL has two different kinds of formulae. State formulae depend directly only on the given state. *Fuzzy formulae* depend on *outcomes*. Additionally, Var represents a set of variables. The syntax of GPL, with $X \in \mathsf{Var}$, $a \in \mathsf{Act}$, $A \in \mathsf{Prop}$, and $0 \leq p \leq 1$, for state formulae, ϕ, and fuzzy formulae, ψ, is:

$$\phi ::= A \mid \neg A \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathsf{Pr}_{>p}\psi \mid \mathsf{Pr}_{\geq p}\psi,$$
$$\psi ::= \phi \mid X \mid \psi \wedge \psi \mid \psi \vee \psi \mid \langle a \rangle \psi \mid [a]\psi \mid \mu X.\psi \mid \nu X.\psi.$$

Note that only atomic propositions may be negated, but every operator has its dual given in the syntax. The propositional connectives, $\wedge$ and $\vee$, can be used on both state and fuzzy formulae. Operators $\mu X.\psi$ and $\nu X.\psi$ are least and greatest fixed point operators for the "equation" $X = \psi$. Additionally, fuzzy formulae must be alternation-free, which prohibits a kind of mixing of least and greatest fixed points, and a formula ψ used to construct state formulae $\mathsf{Pr}_{>p}\psi$ and $\mathsf{Pr}_{\geq p}\psi$ may not have any free variables. These operators check the probability for a fuzzy formula ψ ($\mathsf{Pr}_{>p}$ and $\mathsf{Pr}_{\geq 1-p}$ are duals). The semantics of GPL is given in terms of RPLTS d-trees. In that interpretation, *diamond* implies *box*: $\langle a \rangle \psi$ means that there is an a-transition and it satisfies

ψ; $[a]\psi$ means that if there is an a-transition, it satisfies ψ. We also use a set $\alpha \subseteq \mathsf{Act}$ for the modalities, reading $\langle\alpha\rangle\psi$ as $\bigvee_{a\in\alpha}\langle a\rangle\psi$ and $[\alpha]\psi$ as $\bigwedge_{a\in\alpha}[a]\psi$. When we write $-\alpha$, that represents $\mathsf{Act} \setminus \alpha$.

Example 2.2 (GPL Fuzzy Formula)**.** *For the RPLTS in Example 2.1, an example of a fuzzy formula [CIN05, Example 2] is:*

$$\psi = \mu X.[a][b]X \wedge [a][c]X.$$

From states s_1, s_3, and s_4, an a-transition can go to s_2, which has a b-transition to s_3 and a c-transition to s_4. From s_3 and s_4, it is also possible to go to s_5 and s_6, respectively, where $[b]X$ and $[c]X$ are true *vacuously*. A single unfolding of the formula is:

$$([a][b]\mu X.[a][b]X \wedge [a][c]X) \wedge ([a][c]\mu X.[a][b]X \wedge [a][c]X).$$

As ψ is a least fixed point, it is, in this particular case, satisfied only by finite outcomes, and in fact all such outcomes of the model rooted at s_1 satisfy ψ.

2.2.2 GPL Semantics

We give the semantics of GPL with respect to a fixed RPLTS $L = (S, \delta, P, I)$, where Φ and Ψ are the sets of all state and fuzzy formulae, respectively. A function $\Theta_L : \Psi \to 2^{\mathcal{M}_L}$, augmented with an extra *environment* parameter $e : \mathsf{Var} \to 2^{\mathcal{M}_L}$, returns the set of outcomes satisfying a given fuzzy formula, defined inductively in Table 2.1. For a given $s \in S$, $\Theta_{L,s}(\psi) = \Theta_L(\psi) \cap \mathcal{M}_L(s)$. The relation $\models_L \subseteq S \times \Phi$ indicates when a state satisfies a state formula, and is defined inductively in Table 2.2. Note that the definitions for Θ_L and $\models_L$ are mutually recursive.

There are two properties of fuzzy formulae that are important for the completeness of the GPL model checking algorithm. First, we have distributivity on *box* and *diamond*:

Lemma 2.1 (Distributivity on modal operators ([CIN05] Lemma 1))**.** *Letting* $\oplus \in \{\wedge, \vee\}$*:*

$$\begin{aligned}
\Theta_L([a]\psi_1 \oplus [a]\psi_2) &= \Theta_L\big([a](\psi_1 \oplus \psi_2)\big),\\
\Theta_L(\langle a\rangle\psi_1 \oplus \langle a\rangle\psi_2) &= \Theta_L\big(\langle a\rangle(\psi_1 \oplus \psi_2)\big),\\
\Theta_L([a]\psi_1 \wedge \langle a\rangle\psi_2) &= \Theta_L\big(\langle a\rangle(\psi_1 \wedge \psi_2)\big),\\
\Theta_L([a]\psi_1 \vee \langle a\rangle\psi_2) &= \Theta_L\big([a](\psi_1 \vee \psi_2)\big).
\end{aligned}$$

□

Table 2.1: GPL semantics: fuzzy formulae

$$
\begin{aligned}
\Theta_L(\phi)e &= \bigcup_{s \vDash_L \phi} \mathcal{M}_L(s), \text{where } \phi \text{ is a closed formula,} \\
\Theta_L(X)e &= e(X), \\
\Theta_L(\langle a \rangle \psi)e &= \{T \in \mathcal{M}_L \mid \exists T' : T \xrightarrow{a} T' \wedge T' \in \Theta_L(\psi)e\}, \\
\Theta_L([a]\psi)e &= \{T \in \mathcal{M}_L \mid (T \xrightarrow{a} T') \implies T' \in \Theta_L(\psi)e\}, \\
\Theta_L(\psi_1 \wedge \psi_2)e &= \Theta_L(\psi_1)e \cap \Theta_L(\psi_2)e, \\
\Theta_L(\psi_1 \vee \psi_2)e &= \Theta_L(\psi_1)e \cup \Theta_L(\psi_2)e, \\
\Theta_L(\mu X.\psi)e &= \bigcup_{i=0}^{\infty} M_i, \text{where } M_0 = \varnothing \text{ and } M_{i+1} = \Theta_L(\psi)e[X \mapsto M_i], \\
\Theta_L(\nu X.\psi)e &= \bigcap_{i=0}^{\infty} N_i, \text{where } N_0 = \mathcal{M}_L \text{ and } N_{i+1} = \Theta_L(\psi)e[X \mapsto N_i].
\end{aligned}
$$

Table 2.2: GPL semantics: state formulae

$s \vDash_L A$	iff $A \in I(s)$,
$s \vDash_L \neg A$	iff $A \notin I(s)$,
$s \vDash_L \phi_1 \wedge \phi_2$	iff $s \vDash_L \phi_1$ and $s \vDash_L \phi_2$,
$s \vDash_L \phi_1 \vee \phi_2$	iff $s \vDash_L \phi_1$ or $s \vDash_L \phi_2$,
$s \vDash_L \mathsf{Pr}_{>p}\psi$	iff $\mathsf{m}_s(\Theta_{L,s}(\psi)) > p$,
$s \vDash_L \mathsf{Pr}_{\geq p}\psi$	iff $\mathsf{m}_s(\Theta_{L,s}(\psi)) \geq p$.

Second, we can relate the probability of a conjunction with that of a disjunction [CIN05, Lemma 2]:

$$\mathsf{m}_s(\Theta_{L,s}(\psi_1 \vee \psi_2)) = \mathsf{m}_s(\Theta_{L,s}(\psi_1)) + \mathsf{m}_s(\Theta_{L,s}(\psi_2)) - \mathsf{m}_s(\Theta_{L,s}(\psi_1 \wedge \psi_2)) \quad (2.1)$$

[CIN05, Lemma 2] also includes the following result for a transition:

$$\mathsf{m}_s(\Theta_{L,s}(\langle a \rangle \psi)) = \sum_{s' : (s,a,s') \in \delta} P(s,a,s') \cdot \mathsf{m}_{s'}(\Theta_{L,s'}(\psi)) \quad (2.2)$$

Additionally, a fairly standard property with respect to a μ-calculus is that the negation of a formula, regardless of the underlying model, can be expressed despite the lack of an explicit negation operator. With GPL, this entails the existence of formulae $\mathsf{neg}(\psi)$ and $\mathsf{neg}(\phi)$ for a fuzzy formula ψ and state formula ϕ, respectively, such that, for any RPLTS L and state s [CIN05, Lemma 3]:

$$\Theta_{L,s}(\mathsf{neg}(\psi)) = \mathcal{M}_L(s) - \Theta_{L,s}(\psi) \quad \text{and} \quad s \vDash_L \mathsf{neg}(\phi) \iff s \nvDash_L \phi \ .$$

The proof involves switching all the operators to their duals. Combined with the alternation-free restriction, we thus may limit ourselves to least fixed points when considering fuzzy formulae with only one fixed point.

2.3 GPL Model Checking

For model checking, we require that bound variables be guarded by a diamond or a box operator (this does not affect the expressiveness of GPL [CIN05]). We outline a model checking procedure for a fixed RPLTS $L = (S, \delta, P, I)$.

To compute $\mathsf{m}_{s_0}(\Theta_{L,s_0}(\psi))$, a *dependency graph*, $\Pr(s_0, \psi) = (N, E)$, is constructed. In the *node set* $N \subseteq S \times 2^{Cl(\psi)}$, a node (s, F) has an *associated semantics* of

$$[\![(s, F)]\!] = \Theta_{L,s}(\wedge F),$$

and $Cl(\psi)$ is the *Fisher-Ladner closure*, defined as follows:

- $\psi \in Cl(\psi)$.
- If $\psi' \in Cl(\psi)$, then:
  - if $\psi' = \psi_1 \wedge \psi_2$ or $\psi_1 \vee \psi_2$, then $\psi_1, \psi_2 \in Cl(\psi)$;
  - if $\psi' = \langle a \rangle \psi''$ or $[a]\psi''$ for some $a \in \mathsf{Act}$, then $\psi'' \in Cl(\psi)$;
  - if $\psi' = \sigma X.\psi''$, then $\psi''[\sigma X.\psi''/X] \in Cl(\psi)$, with σ either μ or ν.

The edges in the graph are labeled from $\mathsf{Act} \cup \{\varepsilon^+, \varepsilon^-\}$, so we have the *edge set* $E \subseteq N \times (\mathsf{Act} \cup \{\varepsilon^+, \varepsilon^-\}) \times N$. Finally, from the completed graph, a *system of polynomial equations* can be extracted.

Some nodes are terminal, with no outgoing edges, and many nodes have a single outgoing edge of the form $\big((s, F), \varepsilon^+, (s, F')\big)$. This is based on the existence of a formula $\psi' \in F$, and the node is classified according to the first

node	ψ'	F'	$[\![(s,F)]\!]$	$\mathsf{m}_s([\![(s,F)]\!])$
empty	$F = \varnothing$	no edges	$\mathcal{M}_L(s)$	1
false	ϕ $(s \not\models_L \phi)$ $\langle a\rangle\psi''$ $(s \overset{a}{\not\to})$		$\varnothing$	0
true	ϕ $(s \models_L \phi)$ $[a]\psi''$ $(s \overset{a}{\not\to})$	$F \setminus \{\text{all such } \psi'\}$	$[\![(s,F')]\!]$	$\mathsf{m}_s([\![(s,F')]\!])$
and	$\psi_1 \wedge \psi_2$	$F \setminus \{\psi'\} \cup \{\psi_1, \psi_2\}$		
ν	$\nu X.\psi''$	$F \setminus \{\psi'\} \cup \{\psi''[\psi'/X]\}$		
μ	$\mu X.\psi''$			
or	$\psi_1 \vee \psi_2$	$F_1 = F \setminus \{\psi'\} \cup \{\psi_1\}$, $F_2 = F \setminus \{\psi'\} \cup \{\psi_2\}$, $F_3 = F \setminus \{\psi'\} \cup \{\psi_1, \psi_2\}$	$[\![(s,F_1)]\!]$ $\cup$ $[\![(s,F_2)]\!]$	$\mathsf{m}_s([\![(s,F_1)]\!])$ $+\mathsf{m}_s([\![(s,F_2)]\!])$ $-\mathsf{m}_s([\![(s,F_3)]\!])$

Table 2.3: Non-action nodes

matching rule in the ψ' column in Table 2.3. The *or*-node has three edges, and the edge to (s,F_3) has the label ε^-, as $[\![(s,F_1)]\!] \cap [\![(s,F_2)]\!] = [\![(s,F_3)]\!]$ and, when computing the measure, we use (2.1). If every formula in F has the form $\langle a\rangle\psi'$ or $[a]\psi'$, with s having the corresponding a-transition for each formula, then (s,F) is an *action node*. We define three helper functions: $\mathsf{residue} : 2^\Psi \times \mathsf{Act} \to 2^\Psi$, $\mathsf{action} : 2^\Psi \to 2^{\mathsf{Act}}$, and $\mathsf{f}_{(s,a)} : 2^{\mathcal{M}_L} \to 2^{\mathcal{M}_L(s)}$, as follows:

$$\begin{aligned}
\mathsf{residue}(F,a) &= \{\psi \mid \langle a\rangle\psi \in F \vee [a]\psi \in F\},\\
\mathsf{action}(F) &= \{a \in \mathsf{Act} \mid \exists\psi.\langle a\rangle\psi \in F \vee [a]\psi \in F\},\\
\mathsf{f}_{(s,a)}(M) &= \{T \in \mathcal{M}_L(s) \mid \exists T' \in M.T \overset{a}{\to} T'\}.
\end{aligned}$$

The function $\mathsf{residue}(F,a)$ finds all the formulae with a particular guard, and removes the guard; $\mathsf{action}(F)$ returns all the guards found in F; and $\mathsf{f}_{(s,a)}(M)$ extends trees by adding edges $s \overset{a}{\to} s'$, where s' is a root of other maximal d-trees.

From an action node (s,F), there may be several edges of the form

$((s,F),a,(s',\mathsf{residue}(F,a)))$, and we have:

$$\begin{aligned}
[\![(s,F)]\!] &= \bigcap_{a\in\mathsf{action}(F)} \bigcup_{((s,F),a,(s',F'))\in E} \mathsf{f}_{(s,a)}([\![(s',F')]\!]),\\
\mathsf{m}_s([\![(s,F)]\!]) &= \prod_{a\in\mathsf{action}(F)} \sum_{((s,F),a,(s',F'))\in E} P(s,a,s')\cdot \mathsf{m}_{s'}([\![(s',F')]\!]). \qquad (2.3)
\end{aligned}$$

In (2.3), for the measure of an action node, the intersection naturally turns into a product, and the union into a sum, when independence and mutual exclusion, respectively, are satisfied. Meanwhile, $\mathsf{f}_{(s,a)}([\![(s',F')]\!])$ changes to $P(s,a,s')\cdot \mathsf{m}_{s'}([\![(s',F')]\!])$, which is essentially the contribution of the probabilistic extension to the model checker.

Equations are readily constructed by considering $\mathsf{m}_s([\![(s,F)]\!])$ as a variable. Note that the dependency graph treats μ and ν identically. In solving the stratified system, we start from $\mathbf{0}$ to find the least fixed point, and $\mathbf{1}$ for the greatest fixed point. The alternation-free restriction ensures that there is no cycle in the graph containing both a μ-node and a ν-node.

Example 2.3 (Model Checking)**.** *For the RPLTS in Example 2.1 and fuzzy formula ψ in Example 2.2, letting $\psi_1 = [a][b]\psi$ and $\psi_2 = [a][c]\psi$, the dependency graph is shown in Figure 2.3, and the probability measure of the d-trees satisfying ψ is $\frac{1}{9}$.*

Here, the only nodes with multiple outgoing edges are $(s_3,\{\psi_1,\psi_2\})$ and $(s_4,\{\psi_1,\psi_2\})$, and the generated equations reduce to

$$\begin{aligned}
x_1 &= x_2\cdot x_3,\\
x_2 &= \frac{3}{4}x_1+\frac{1}{4},\\
x_3 &= \frac{3}{4}x_1+\frac{1}{4},
\end{aligned}$$

with $\mathsf{m}_{s_1}([\![(s_1,\psi)]\!]) = x_1 = \left(\frac{3}{4}x_1+\frac{1}{4}\right)^2$ and solutions $x_1 = \frac{1}{9}$ and $x_1 = 1$, of which the former corresponds to the least fixed point.

Note that, in general, one way to break down the model checking algorithm is in the following two parts: writing down a polynomial system, and then finding the (approximate) solution. The first part is bounded exponentially in the size of the fuzzy formula, as we deal with subsets of the Fisher-Ladner closure. For the second part, value iteration is guaranteed to converge [CIN05, Lemma 10], but may be exponentially slow in the number

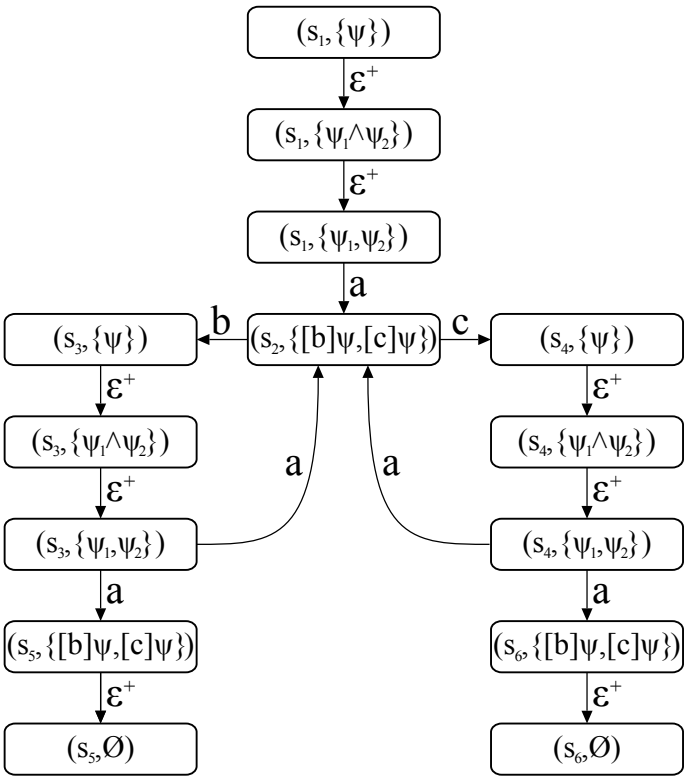


Figure 2.3: Dependency graph $\Pr(s_1, \psi)$

of digits of precision [EY09, KLE07]. When the polynomial system is of a specific form, discussed in Section 2.5, alternative approximation methods have been proven to be efficient [KLE07, ESY12b].

2.4 Recursive Markov Chains

Recursion is a powerful programming paradigm. When a function or method is called, execution is paused in the current context, to be resumed when it returns, and possibly using a return value. A Recursive Markov Chain (RMC) [EY09] is basic model for this in the probabilistic domain and a branching-time extension of Markov chains. Recursion happens via special entry and exit nodes, as well as *boxes* with call and return ports. Formally, we define an RMC A as a tuple $(A_1, \ldots, A_k)$, where each *component graph* A_i is a sextuple $(N_i, B_i, Y_i, \mathsf{En}_i, \mathsf{Ex}_i, \delta_i)$:

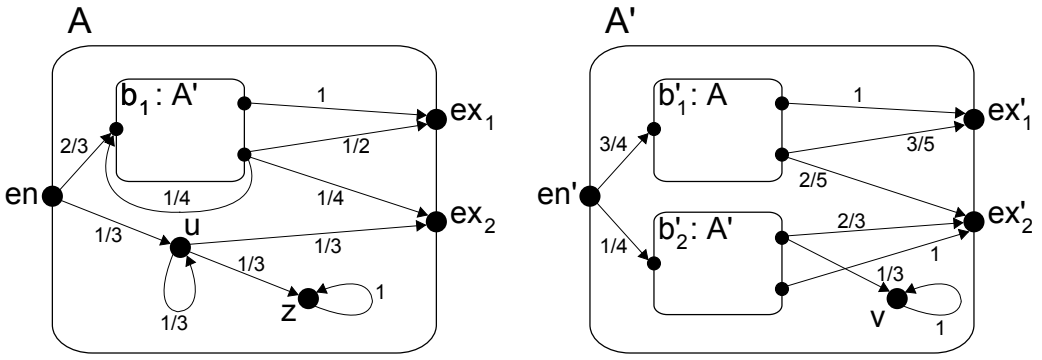


Figure 2.4: RMC Example

- N_i is a set of nodes, containing subsets En_i and Ex_i of entry and exit nodes, respectively.
- B_i is a set of boxes, with a mapping $Y_i : B_i \to \{1, \ldots, k\}$ assigning each box to a component. Each box has a set of call and return ports, corresponding to the entry and exit nodes, respectively, in the corresponding components: $\mathsf{Call}_b = \{(b, en) \mid en \in \mathsf{En}_{Y_i(b)}\}$, $\mathsf{Return}_b = \{(b, ex) \mid ex \in \mathsf{Ex}_{Y_i(b)}\}$.
- δ_i is the transition relation, with transitions of the form (u, p_{uv}, v), where u may not be an exit node or a call port, and v may not be an entry node or a return port. Additionally, $p_{uv} \in (0, 1]$ and, for each u, $\sum_{v':(u,\cdot,v')\in\delta_i} p_{uv'} = 1$.

Starting at an entry node of a component, we are typically interested in the probability that the RMC terminates at a particular (or any) exit node of the component. The problem of reachability of an RMC node, either with a particular recursive stack or ignoring it, also reduces to termination [EY09, Proposition 2.1]. In computing the termination probability of an RMC, we solve a system of polynomial equations similar to one produced from GPL model checking, which we discuss in the next section. Additionally, we can translate an RMC to an RPLTS, which we will show in Section 3.4.2.

Example 2.4 (Sample RMC)**.** *Figure 2.4 [EY09, Figure 1] shows an example of an RMC R with two components, A and A', each having a single entry node and two exit nodes. In addition, the components combine to have three boxes and three internal nodes.*

Formally, we have $R = \{A_1, A_2\}$*, with* $A_1 = A$*, where:*

- $N_1 = \{u, z, en, ex_1, ex_2\}$*, with* $\mathsf{En}_1 = \{en\}$ *and* $\mathsf{Ex}_1 = \{ex_1, ex_2\}$*;*
- $B_1 = \{b_1\}$*, with* $Y_1(b_1) = 2$*,* $\mathsf{Call}_{b_1} = \{(b_1, en')\}$*,*
  and $\mathsf{Return}_{b_1} = \{(b_1, ex'_1), (b_1, ex'_2)\}$*;*
- *and transitions such as* $(u, 1/3, z) \in \delta_1$ *and* $((b_1, ex'_2), 1/4, (b_1, en')) \in \delta_1$*;*

and $A_2 = A'$*, where:*

- $N_2 = \{en', v, ex'_1, ex'_2\}$*, with* $\mathsf{En}_2 = \{en'\}$ *and* $\mathsf{Ex}_2 = \{ex'_1, ex'_2\}$*;*
- $B_2 = \{b'_1, b'_2\}$*, with* $Y_2(b'_1) = 1$ *and* $Y_2(b'_2) = 2$*,* $\mathsf{Call}_{b'_1} = \{(b'_1, en)\}$ *and* $\mathsf{Return}_{b'_1} = \{(b'_1, ex_1), (b'_1, ex_2)\}$*, and* $\mathsf{Call}_{b'_2} = \{(b'_2, en')\}$ *and* $\mathsf{Return}_{b'_2} = \{(b'_2, ex'_1), (b'_2, ex'_2)\}$*;*
- *and transitions such as* $(en', 3/4, (b'_1, en)) \in \delta_2$ *and* $((b'_1, ex_2), 3/5, ex'_1) \in \delta_2$*.*

RMCs' components often have a single entry node, as this restriction does not affect RMC expressiveness [EY09]. Meanwhile, when all components have a single exit, an RMC is called a 1-exit RMC (or 1-RMC); otherwise, it's a multi-exit RMC.

2.5 Probabilistic Polynomial Systems

One model checking paradigm is to construct an automaton with an acceptance condition from a model and a desired property. In some cases, the automaton is essentially in the same class as the model, such as with PCTL model checking, where the Markov process is transformed into another Markov process, for which a given property becomes simple reachability [HJ94]. This applies also to RMCs and reducing properties to termination [EY09]. In other cases, the properties cannot be reduced, and instead are transformed to automata representing them; then, the product of the model with the property yields a more general automaton, as with LTL and PCTL* model checking [IN96, Bai98]. The dependency graph construction for GPL model checking, described in Section 2.3, is also essentially building a product of an RPLTS with a fuzzy formula [CIN05]. In both the RMC and GPL cases, the automata lead to polynomial systems of equations, for which

the least fixed point (LFP) or the greatest fixed point (GFP) is the desired solution [Ste15].

The systems are of the form $\mathbf{x} = P(\mathbf{x})$. Setting $\mathbf{x}^{(0)} = \mathbf{0}$ and iterating with $\mathbf{x}^{(i+1)} = P(\mathbf{x}^{(i)})$, where $\mathbf{x}^{(i+1)} \geq \mathbf{x}^{(i)}$ for all $i \geq 0$, the sequence always converges to the LFP $\mathbf{q}^*$, with $\mathbf{q}^* = P(\mathbf{q}^*)$. The GFP may be found similarly, starting with $\mathbf{x}^{(0)} = \mathbf{1}$. The alternation-free restriction of GPL may lead to solving a stratified system, where there is an acyclic relation between least and greatest fixed points, and numerical issues may lead to indeterminate results [CIN05]; given our focus, we are satisfied with the GPL model checking algorithm producing the polynomial system. Another concern is that value iteration may be exponentially slow in the number of digits of precision (*e.g.*, for the equation $x = \frac{1}{2}x^2 + \frac{1}{2}$ [KLE07]), and additional results have been reached for the systems arising from RMCs.

The polynomial systems produced from 1-RMCs and those from multi-exit RMCs were not initially distinguished [EY09], each placed as a proper subset of Monotone Polynomial Systems (MPSs) and explored with an adaptation of Newton's method to speed up computation. A polynomial-time algorithm was subsequently given for the computation of an LFP for PPSs, which correspond to the systems produced from 1-RMCs [ESY12b]. A Probabilistic Polynomial System (PPS) may always be converted to a Simple Normal Form (SNF) [ESY12a, ESY15]. In the case without linear nondeterminism, for the system $\mathbf{x} = P(\mathbf{x})$, this means two forms:

- Form L: $P_i(\mathbf{x}) = a_{i,0} + \sum_{j=1}^{n} a_{ij}x_j$, with $a_{ij} \geq 0$ for all j and $\sum_{j=0}^{n} a_{ij} \leq 1$;
- Form Q: $P_i(\mathbf{x}) = x_j \cdot x_k$ for some j, k.

For multi-exit RMCs, Form Q may be modified to a sum of products, and for the least solution $\mathbf{q}^*$, $\mathbf{q}^* \in [0,1]^n$ because the left factors correspond to reaching distinct exits and are thus mutually exclusive [EY09].

In the next chapter, we will consider systems with nondeterminism. Then, the following additional forms are in the SNF of a min/maxPPS [ESY12a, ESY15]:

- Form M: $P_i(\mathbf{x}) = \min\{x_j, x_k\}$ for some j, k;
- Form X: $P_i(\mathbf{x}) = \max\{x_j, x_k\}$ for some j, k.

We consider all the polynomial systems produced in this and the next chapter to be probabilistic, as they remain in $[0,1]^n$. However, to distinguish from PPSs, we will call them Extended Polynomial Systems (XPSs) in this thesis.

For the systems produced from GPL, if there are no disjunctions (including implicit ones), the resulting system will be a PPS; disjunctions lead to a specific kind of subtraction, which keeps the whole system in $[0,1]^n$ [CIN05]. We may thus refer to a (min/max)PPS as a *conjunctive XPS*. We may also consider a disjunctive form, Form D, that is the dual to Form Q, and define *separable* and *disjunctive* XPSs.

Definition 2.1 (Extended Polynomial System)**.** *An additional form for SNF in XPS is:*

- *Form* D*:* $P_i(\mathbf{x}) = x_j + x_k - x_j \cdot x_k$ *for some* j, k*.*

Separable XPSs in SNF may have equations in Forms L*,* Q*,* D*,* M*, and* X*; a conjunctive XPS and a disjunctive XPS are separable XPSs with no equations in Form* D *and Form* Q*, respectively.* □

Separable XPSs (without Forms M and X) have recently been considered with respect to game automata [MM15]. The systems produced by the GPL model checking algorithm may actually go beyond a separable XPS, but without any equations in Forms M or X; we will call this a *stochastic* XPS and use an *asymmetric* form for disjunction, Form D', to represent it.

Definition 2.2 (Stochastic XPS)**.** *We define the following additional form:*

- *Form* D'*:* $P_i(\mathbf{x}) = x_j + x_k - x_m$ *for some* j, k, m*.*

A stochastic XPS in SNF may have Forms L *and* Q*, as above, and Form* D'*.* □

That is, disjunctions are handled with a special form of subtraction such that the whole system may not be monotonic (*i.e.*, $\mathbf{x} \leq \mathbf{y}$ no longer implies $P(\mathbf{x}) \leq P(\mathbf{y})$), but does not violate monotonicity under iteration starting from $\mathbf{0}$ or $\mathbf{1}$, because when this form arises, x_m changes more slowly than x_j and x_k [CIN05, Lemma 10].

Note that Form Q represents the conjunction of independent outcomes; if x_m is in Form Q, such that $x_m = x_j \cdot x_k$, then Form D' reduces to Form D. Meanwhile, when x_j and x_k correspond to mutually exclusive outcomes in Form D', then, for the LFP $\mathbf{q}^*$, $q^*_m = 0$. Thus, if all disjunctions were mutually

exclusive, in the case of LFP computation, there would be no true subtraction in the resulting XPS (cf. the system formed from a multi-exit RMC [EY09]).

The problem of computing the LFP of a disjunctive XPS may be reduced to computing the GFP of a conjunctive XPS. The computation of a GFP for a (min/max)PPS, as for the LFP, has a polynomial-time algorithm unless both Forms M and X are present [ESY15].

If we are model checking an LFP disjunctive formula, we can get to the PPS GFP computation in multiple ways. First, the negation of an LFP disjunctive formula is a GFP conjunctive formula, which would then lead to a PPS, and the nature of the fixed point does not affect the PPS construction. Alternatively, we can employ the operator $R : [0,1]^n \to [0,1]^n$, with $R(\mathbf{x}) = \mathbf{1} - P(\mathbf{1} - \mathbf{x})$ (R is mentioned in the introduction of [ESY15]). Applying R to a disjunctive XPS produces a conjunctive XPS, since Form L is essentially unchanged, while all Form D equations convert to Form Q (observe that $a + b - ab = 1 - (1 - a)(1 - b)$).

2.6 The Interpretation of Branching Time

While Markov processes produce only linear equation systems, probabilistic branching-time systems may produce nonlinear ones. We have explored several probabilistic branching-time paradigms, such as the recursive calls in RMCs. The paradigm with RMCs is a sequential (or nested) one: there is a recursive call, and either exactly one or a few places to resume on return (and the possibility that a return port is not reached at all), on which the subsequent execution depends. An alternative probabilistic branching time paradigm is parallel (or independent) branching. In this case, we might be interested in what happens on all branches, or whether a branch satisfying a property exists, quite similarly to CTL [CES86]. A basic probabilistic branching-time model is a Markov Branching Play (MBP) [Mio11, Mio12, MM15]. Meanwhile, other related work has considered the more specialized Branching Processes (BPs) [EY09, ESY12b, CDK12, ESY15].

2.6.1 Branching Processes

One of the ways a BP may be used is to model populations. A process may spawn additional processes, increasing the population, or die, decreasing it. It may also evolve to a different (*e.g.*, older) type, and all of the processes

evolve in each time step. We define a BP Δ following [CDK12, Definition 1], but with notation more closely mirroring RPLTSs, as a triple (Γ, δ, P), where:

- Γ is a finite set of *types*;
- $\delta \subseteq \Gamma \times \Gamma^+$ is a finite set of transition rules; and
- $P : \delta \to (0, 1]$ is a transition probability distribution satisfying:

$$\forall X \in \Gamma. \sum_{\alpha:(X,\alpha)\in\delta} P(X, \alpha) = 1.$$

Additionally, we will write $(X, \alpha) \in \delta$ with $P(X, \alpha) = p$ as $X \xrightarrow{p} \alpha$. A BP state is a multi-set of types, and a transition applies a transition rule simultaneously to each element in the set. Deadlock is modeled by a type D with the rule $D \xrightarrow{1} D$ (alternative definitions of BPs allow a transition rule for termination, $(X, \epsilon) \in \delta$).

Example 2.5 (BP Example)**.** *An example BP [CDK12, Equation (1)] is:*

$$\begin{array}{lll} I \xrightarrow{0.9} I, & B \xrightarrow{0.2} D, & D \xrightarrow{1} D, \\ I \xrightarrow{0.1} IB, & B \xrightarrow{0.5} B, & \\ & B \xrightarrow{0.3} BB. & \end{array}$$

BP extinction refers to reaching a state that is an empty set, and the problem of finding BP extinction probability can be reduced to 1-RMC termination [EY09]. Note that, with the case of the deadlock type D instead of termination rules, reaching a state with only D types is essentially equivalent to extinction. In Example 2.5, the extinction probability when starting at state B is $\frac{2}{3}$. Meanwhile, the reachability problems are distinct for BPs and 1-RMCs (the BP reachability problem is considered in [ESY15, Section 5]). Neither extinction nor reachability properties depend on the specifics of how BPs run, and it is sufficient to know that processes evolve independent of one another. Additionally, a PCTL-like [HJ94] logic, called Probabilistic Tree Temporal Logic (PTTL), has been described over BPs [CDK12].

2.6.2 PTTL

PTTL is a logic over labeled BPs, which have an interpretation function I, as with RPLTSs. We give the syntax of PTTL following [CDK12, Definition 18], but more closely mirroring GPL. With $A \in \mathsf{Prop}$, we refer to ϕ and ψ as state and fuzzy formulae, as for GPL:

$$\phi ::= \mathsf{tt} \mid A \mid \neg\phi \mid \phi \wedge \phi \mid \mathsf{Pr}_{\geq p}\psi \mid \mathsf{Pr}_{>p}\psi,$$
$$\psi ::= \mathsf{AX}\phi \mid \mathsf{EX}\phi \mid \mathsf{A}[\phi \mathsf{U} \phi] \mid \mathsf{E}[\phi \mathsf{U} \phi] \mid \mathsf{A}[\phi \mathsf{R} \phi] \mid \mathsf{E}[\phi \mathsf{R} \phi].$$

We may view any RPLTS without terminal states as a BP, and the semantics of the X and U operators on d-trees is essentially as in CTL [CES86] (R is the dual of U); we will discuss PTTL semantics further in Section 3.4. The polynomial system of equations produced by model checking PTTL is *PPS-expressible* [CDK12].

2.6.3 Additional Interpretations

A simple formula over probabilistic branching time that goes beyond PPSs is $\mu X.[-]\langle - \rangle X$, which alternates between making the "best" and "worst" choices. Aside from GPL [CIN05], the polynomial system that could be produced from such a formula has been recently analyzed with respect to game automata [MM15].

Additionally, some GPL formulae may be called *entangled*, *e.g.*, $\psi_e = ([a]\psi_1 \wedge [b]\psi_4) \vee ([a]\psi_2 \wedge [b]\psi_3)$. Intuitively, this means that we cannot consider each branch independently, as what we need to satisfy after an a action depends on what happens after a b action (and vice versa, in this example). The ability to express and model check entangled formulae is a feature of GPL (we further discuss entanglement in Section 3.3.1), and finding applications for them is an interesting possibility.

Chapter 3

Linear Nondeterminism in Probabilistic Systems

For finite-state systems, model checking a temporal property can be cast in terms of model checking in the modal μ-calculus, the so-called "assembly language" of temporal logics. A number of temporal logics have been proposed and used for specifying properties of finite-state *probabilistic* systems. Two of the notable logics for probabilistic systems based on the μ-calculus are GPL [CIN05] and pLμ [Mio11].

GPL is defined over RPLTSs. The formal definitions of GPL and RPLTSs were given in Chapter 2, mainly following [CIN05]. In an RPLTS, each state has a set of outgoing transitions with distinct labels; each action, in turn, specifies a (probabilistic) distribution of target states. GPL categorizes formulae into *state formulae*, which have a deterministic truth value at a state, and *fuzzy formulae*, whose truth at a state is probabilistic. Operators of the form "Pr" are used to construct state formulae from fuzzy formulae. For example, the state formula $\mathsf{Pr}_{>0.5}\psi$ expresses that the fuzzy formula ψ holds with probability greater than 0.5. GPL is expressive enough to serve as an "assembly language" of a large number of probabilistic temporal logics. For instance, model checking PCTL* properties over Markov chains, as well as termination and reachability of RMCs can be cast in terms of GPL model checking [CIN05, GRS12].

In this chapter, we consider an extension to GPL to express properties of probabilistic systems with both linear and branching nondeterminism. This logic, called Extended Probabilistic Logic (XPL), is defined over PLTSs. In a PLTS, each state has a set of outgoing transitions, *possibly with common labels*; and each transition specifies a distribution of target states. PLTSs thus exhibit both probabilistic choice and nondeterministic choice, and moreover, under the XPL semantics, the internal nondeterminism is linear, while the external nondeterminism is branching. Syntactically, XPL differs from GPL by replacing the Pr operators with $\mathsf{Pr}^{\max}$ and $\mathsf{Pr}^{\min}$ to account for the max-

imal and minimal probabilistic values of fuzzy formulae (which may differ due to the internal choices in PLTSs).

Contributions and Significance: XPL is expressive enough that a wide variety of independently-studied verification problems can be cast as model checking PLTSs with XPL. In fact, undecidable problems such as termination of multi-exit RMDPs can be reduced in linear time to model checking PLTSs with XPL. We introduce a syntactically-defined subclass, called *separable XPL*, for which model checking is decidable. We describe a procedure for model checking XPL which always terminates— successfully with the model checking result, or with failure— such that it always terminates successfully for separable XPL (see Section 3.3).

A number of distinct model checking algorithms have been developed independently for decidable verification problems involving systems that have probabilistic and internal nondeterministic choice. Examples of such problems include PCTL* model checking of MDPs [Bai98], reachability in Branching MDPs (BMDPs) [ESY15], and termination of 1-exit RMDPs [EY15]. These problems can all be reduced, in linear time, to model checking *separable* XPL formulae over PLTSs (see Section 3.4).

Termination of multi-exit RMDPs, cast as a model checking problem over XPL along the same lines as our treatment of 1-exit RMDPs, yields an XPL formula that is not separable. Thus separability can be seen as a characteristic of the verification problems that are known to be decidable, when cast in terms of model checking in XPL. Consequently, XPL in general, and separable XPL in particular, form a useful formalism to study the relationships between verification problems over systems involving probabilistic and both linear- and branching-time nondeterministic choice. We discuss these issues in greater detail in Section 3.5.

3.1 Probabilistic Labeled Transition Systems

We define a Probabilistic LTS (PLTS) as an extension of an RPLTS.

Definition 3.1 (PLTS)**.** *With respect to fixed sets* Act *and* Prop *of* actions *and* propositions*, respectively, a PLTS L is a quadruple (S, δ, P, I), where*

- *S is a countable set of states;*
- *$\delta \subseteq S \times \mathsf{Act} \times S$ is the transition relation;*

- $P : \delta \times \mathbb{N} \to [0,1]$ *is the transition probability distribution satisfying:*
  - $\forall s \in S. \forall a \in \mathsf{Act}. \forall c \in \mathbb{N}. \sum_{s':(s,a,s')\in\delta} P(s,a,s',c) \in \{0,1\}$,
  - $\forall s. \forall a. \forall c \in \mathbb{N}. (\exists s'. (s,a,s') \in \delta) \implies \sum_{s':(s,a,s')\in\delta} P(s,a,s',c) = 1$, *and*
  - $\forall s. \forall a. \forall s'. (s,a,s') \in \delta \implies \exists c \in \mathbb{N}. P(s,a,s',c) > 0$*;*
- $I : S \to 2^{\mathsf{Prop}}$ *is the* interpretation*, recording the set of propositions true at a state.* □

Recall that an RPLTS does not have internal nondeterminism, *i.e.*, its transition probability distribution P is a function of δ. We defined P in this way in order to retain δ and d-trees as defined for RPLTSs. We also assume that there are finitely many distributions to choose from, *i.e.*, given s and a, there exists a c such that for all s' and $c' \geq c$, $P(s,a,s',c') = P(s,a,s',c)$. A scheduler chooses the distribution by providing a natural number.

This definition is also in line with the most general for a PLTS [Mio11, Seg95], in which, given an action, a probabilistic distribution is chosen nondeterministically. Other equally expressive models include alternating automata, in which labeled nondeterministic choices are followed by silent probabilistic ones. The difference between such models has been analyzed with respect to bisimulation [ST05].

Many properties carry over naturally from RPLTSs as given in Section 2.1. To resolve the nondeterministic transitions and give a measure for a set of outcomes, we additionally require a scheduler. Recall that $\mathcal{C}_L$ is the set of all partial computations σ of L.

Definition 3.2 (Reactive scheduler)**.** *A scheduler for an PLTS L is a function $\gamma : \mathcal{C}_L \times \mathsf{Act} \to \mathbb{N}$.* □

Note that we have defined deterministic schedulers, which are also aware of their relevant histories. Given a scheduler γ for a PLTS L, we have a (countable) RPLTS L_γ, where $S_{L,\gamma} \subseteq \mathcal{C}_L$ and so $\delta_{L,\gamma} \subseteq \mathcal{C}_L \times \mathsf{Act} \times \mathcal{C}_L$. We define a probability distribution:

Definition 3.3 (Combined probability)**.** *The probability distribution of a PLTS L with scheduler γ is a function, $P_{L,\gamma} : \delta_{L,\gamma} \to (0,1]$, where:*

$$P_{L,\gamma}(\sigma, a, \sigma') = P_L(\mathsf{last}(\sigma), a, \mathsf{last}(\sigma'), \gamma(\sigma, a)).$$

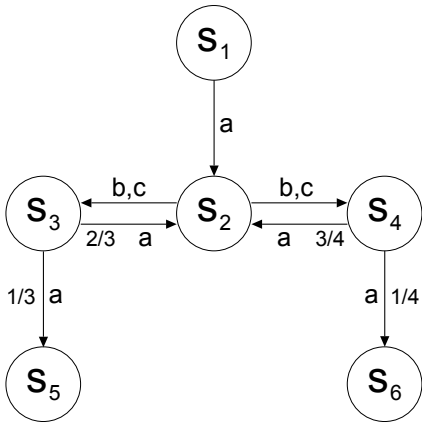


Figure 3.1: PLTS Example

□

It is convenient to extend Definition 3.3, letting $P_{L,\gamma}(\sigma, a, \sigma') = 0$ when $(\sigma, a, \sigma') \notin \delta_{L,\gamma}$.

Additionally, recall that the basic cylindrical subset B_T contains all maximal d-trees sharing the prefix tree T. For these subsets, we define the probability measure:

Definition 3.4 (Probability measure)**.** *For a PLTS L with scheduler γ, the probability measure of a basic cylindrical subset B_T is defined by a partial function $\mathsf{m}^\gamma : 2^{\mathcal{M}_L} \to [0,1]$, where:*

$$\mathsf{m}^\gamma(B_T) = \prod_{(\sigma,a,\sigma') \in \mathsf{edges}(T)} P_\gamma(\sigma, a, \sigma').$$

□

Since m^γ may be considered as defined for an RPLTS, we can extend it to a measure m^γ_s as in Section 2.1.

Example 3.1 (PLTS)**.** *Figure 3.1 is an example of a PLTS, where*

- $S = \{s_1, s_2, s_3, s_4, s_5, s_6\}, \mathsf{Act} = \{a, b, c\}$,
- $\delta = \{(s_1, a, s_2), (s_3, a, s_2), (s_3, a, s_5), (s_4, a, s_2), (s_4, a, s_6),$
  $(s_2, b, s_3), (s_2, b, s_4), (s_2, c, s_3), (s_2, c, s_4)\}$,

- $P(s_1, a, s_2, \cdot) = 1, P(s_3, a, s_5, \cdot) = \frac{1}{3}, P(s_3, a, s_2, \cdot) = \frac{2}{3}, P(s_4, a, s_6, \cdot) = \frac{1}{4}, P(s_4, a, s_2, \cdot) = \frac{3}{4}$, $P(s_2, b, s_3, 0) = P(s_2, c, s_3, 0) = 1$, *and for all* $k > 0$*:* $P(s_2, b, s_4, k) = P(s_2, c, s_4, k) = 1$.

An example of a scheduler γ for this system is $\gamma(\sigma, b) = 0$ and $\gamma(\sigma, c) = 1$, for all σ such that $\mathsf{last}(\sigma) = s_2$.

3.2 XPL

Now we define Extended Probabilistic Logic (XPL), our extension to GPL. The syntax for XPL fuzzy formulae is entirely unchanged from Section 2.2.1. Meanwhile, we replace the GPL state formula Pr operators with $\mathsf{Pr}^{\min}$ and $\mathsf{Pr}^{\max}$.

3.2.1 XPL Syntax

The XPL syntax is then as follows:

$$\phi ::= A \mid \neg A \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathsf{Pr}^{\min}_{>p}\psi \mid \mathsf{Pr}^{\min}_{\geq p}\psi \mid \mathsf{Pr}^{\max}_{>p}\psi \mid \mathsf{Pr}^{\max}_{\geq p}\psi,$$
$$\psi ::= \phi \mid X \mid \psi \wedge \psi \mid \psi \vee \psi \mid \langle a \rangle \psi \mid [a]\psi \mid \mu X.\psi \mid \nu X.\psi.$$

$\mathsf{Pr}^{\min}$ and $\mathsf{Pr}^{\max}$ compare against the minimum and maximum probabilities, respectively, over all schedulers. Note that for an RPLTS, $\mathsf{Pr}^{\min}_{>p}\psi$ and $\mathsf{Pr}^{\max}_{>p}\psi$ are equivalent to each other and to GPL's $\mathsf{Pr}_{>p}\psi$.

3.2.2 XPL Semantics

The semantics of XPL changes, from GPL as described in Section 2.2.2, only due to the measure of the PLTS outcomes. It is defined with respect to a fixed PLTS $L = (S, \delta, P, I)$. The function $\Theta_L : \Psi \to 2^{\mathcal{M}_L}$ remains essentially the same, while $\models_L \subseteq S \times \Phi$ differs for the probabilistic operators.

Definition 3.5 (XPL Semantics)**.** *The semantics for the state formulae is given in Table 3.1. For the fuzzy formulae, the semantics are as in Table 2.1.* □

Note the use of sup and inf in Table 3.1. We may refer to a value like $\sup_\gamma \mathsf{m}^\gamma_s(\Theta_{L,s}(\psi))$ as a *probabilistic value*, denoted by $\mathsf{Pr}^{\max}_{L,s}(\psi)$ ([DGJP02] uses the term *capacity*). Unlike in GPL, we may not always be able use a model checking algorithm to produce a polynomial system for which this is a solution.

Table 3.1: XPL semantics: state formulae

$s \vDash_L A$	iff $A \in I(s)$,
$s \vDash_L \neg A$	iff $A \notin I(s)$,
$s \vDash_L \phi_1 \wedge \phi_2$	iff $s \vDash_L \phi_1$ and $s \vDash_L \phi_2$,
$s \vDash_L \phi_1 \vee \phi_2$	iff $s \vDash_L \phi_1$ or $s \vDash_L \phi_2$,
$s \vDash_L \mathsf{Pr}^{\max}_{>p}\psi$	iff $\sup_\gamma \mathsf{m}^\gamma_s(\Theta_{L,s}(\psi)) > p$,
$s \vDash_L \mathsf{Pr}^{\max}_{\geq p}\psi$	iff $\sup_\gamma \mathsf{m}^\gamma_s(\Theta_{L,s}(\psi)) \geq p$,
$s \vDash_L \mathsf{Pr}^{\min}_{>p}\psi$	iff $\inf_\gamma \mathsf{m}^\gamma_s(\Theta_{L,s}(\psi)) > p$,
$s \vDash_L \mathsf{Pr}^{\min}_{\geq p}\psi$	iff $\inf_\gamma \mathsf{m}^\gamma_s(\Theta_{L,s}(\psi)) \geq p$.

3.3 XPL Model Checking

In this section, we describe an algorithm for model checking an XPL fuzzy formula [GR16]. As we will see, it does not always succeed.

With the GPL model checking algorithm, we can factor any formula by building a dependency graph. The key idea that enables this factoring is (2.1). Meanwhile, not all XPL formulae can be factored. The corresponding equality for XPL becomes (3.1) (if we are maximizing):

$$\sup_\gamma \mathsf{m}^\gamma_s\big(\Theta_{L,s}(\psi)\big) = \sup_\gamma \Big(\mathsf{m}^\gamma_s\big(\Theta_{L,s}(\psi_1)\big) + \mathsf{m}^\gamma_s\big(\Theta_{L,s}(\psi_2)\big) - \\ - \mathsf{m}^\gamma_s\big(\Theta_{L,s}(\psi_1 \wedge \psi_2)\big)\Big), \quad (3.1)$$

which is not particularly useful, because this does not yield a relationship for the probabilistic values, $\mathsf{Pr}^{\max}_{L,s}(\psi)$, as in (3.2):

$$\mathsf{Pr}^{\max}_{L,s}(\psi_1 \vee \psi_2) \stackrel{?}{=} \mathsf{Pr}^{\max}_{L,s}(\psi_1) + \mathsf{Pr}^{\max}_{L,s}(\psi_2) - \mathsf{Pr}^{\max}_{L,s}(\psi_1 \wedge \psi_2). \quad (3.2)$$

Indeed, it is easy to give an example where (3.2) does not hold. Consider the formula $\psi_a = [a]\langle b\rangle\mathsf{tt} \vee [a]\langle c\rangle\mathsf{tt}$, for a PLTS L (Figure 3.2) with $\{s_a, s_b, s_c\} \in S_L$ and nondeterministic a-transitions from s_a to s_b and s_c, such that s_b has a b-transition, but no c-transitions, and vice versa for s_c. Then, $\mathsf{Pr}^{\max}_{L,s_a}(\psi_a) = \mathsf{Pr}^{\max}_{L,s_a}([a]\langle b\rangle\mathsf{tt}) = \mathsf{Pr}^{\max}_{L,s_a}([a]\langle c\rangle\mathsf{tt}) = 1$, but $\mathsf{Pr}^{\max}_{L,s_a}([a]\langle b\rangle\mathsf{tt} \wedge [a]\langle c\rangle\mathsf{tt}) = 0$, and

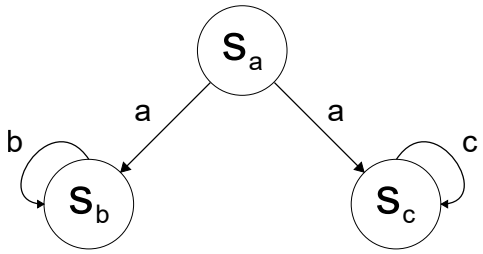


Figure 3.2: Example PLTS with nondeterministic choice on "a"

$1 + 1 - 0 \neq 1$. However, Lemma 2.1 still applies with linear nondeterminism present, because it deals only with d-trees and not measures, and we can still model check this formula, if we delay dealing with the disjunction by rewriting ψ_a as $[a](\langle b\rangle \mathsf{tt} \vee \langle c\rangle \mathsf{tt})$, as the subformula guarded by a is trivial to check at any state. We generalize this to a syntactic notion of *separability*, defined below.

3.3.1 Separability of Fuzzy Formulae

In the GPL model checking algorithm in Section 2.3, we dealt with sets of fuzzy formulae, under the semantics of conjunction. Since we cannot relate disjunctions to conjunctions with XPL, the corresponding construct becomes an *and-or tree*.

Definition 3.6 (And-or tree)**.** *The and-or tree of a fuzzy formula ψ, $AO(\psi)$ is a node labeled by $\oplus$, where $\oplus \in \{\wedge, \vee\}$, with children $AO(\psi_1)$ and $AO(\psi_2)$ when $\psi = \psi_1 \oplus \psi_2$, and a leaf ψ otherwise.*

We can flatten this tree with the straightforward flattening operator, where, *e.g.*, the tree $\wedge(\psi_1, \ldots, \wedge(\psi_2, \psi_3))$ is flattened to $\wedge(\psi_1, \ldots, \psi_2, \psi_3)$. Note that flattened trees have alternating $\wedge$ and $\vee$ nodes. A (conjunctive) set of formulae F corresponds to a flattened and-or tree with the root node labeled by $\wedge$ and having the elements of F as leaves. We will assume $AO(\psi)$ refers to the flattened tree.

A subformula of ψ of the form $\langle a\rangle\psi'$ or $[a]\psi'$ is called a *modal subformula* of ψ. We say that ψ' is an *unguarded subformula* of ψ if it is a leaf of $AO(\psi)$. We also retain the requirement for bound variables (in fixed points) to be guarded by modal operators.

Definition 3.7 (Formula transformations)**.**

- *The* fixed-point expansion *of ψ, denoted by $FPE(\psi)$, is a formula ψ' obtained by expanding any unguarded subformula of the form $\sigma X.\psi_X$ to $\psi_X[\sigma X.\psi_X/X]$ where $\sigma \in \{\mu, \nu\}$.*

- *We say that a formula is non-probabilistic if it is a state formula, or of the form $\langle a\rangle\phi$ and $[a]\phi$ for $a \in \mathsf{Act}$ and $\phi \in \{\mathsf{tt}, \mathsf{ff}\}$. The* purely probabilistic abstraction *of a fuzzy formula ψ, denoted by $PPA(\psi)$, is a formula obtained by removing unguarded non-probabilistic subformulae (*i.e.*, we replace all instances of $\psi \oplus \phi$ with ψ, where ψ is arbitrary, ϕ is non-probabilistic, and $\oplus \in \{\wedge, \vee\}$).*

- *A* grouping *of a formula ψ, denoted by $GRP(\psi)$, groups modalities in a formula using distributivity. Formally, GRP maps ψ to ψ' by applying equivalences in Lemma 2.1 left-to-right at the top level.* □

In particular, *GRP* may be viewed as repeatedly combining two leaves of a node, if they are both modal subformulae with the same action; if the node is binary, this also turns the node into a leaf node.

At a high level, a necessary condition of separability is that the actions guarding distinct conjuncts and disjuncts of a formula are distinct as well. We make this precise by first defining the *action set* of a formula (cf. $\mathsf{action}(F)$ in Section 2.3).

Definition 3.8 (Action set)**.** *The* action set *of a formula ψ, denoted by $\mathsf{action}(\psi)$ is the set of actions appearing at unguarded modal subformulae of ψ:*

- $\mathsf{action}(\phi) = \varnothing$;
- $\mathsf{action}(\langle a\rangle\psi) = \mathsf{action}([a]\psi) = \{a\}$;
- $\mathsf{action}(\psi_1 \wedge \psi_2) = \mathsf{action}(\psi_1 \vee \psi_2) = \mathsf{action}(\psi_1) \cup \mathsf{action}(\psi_2)$;
- $\mathsf{action}(\mu X.\psi) = \mathsf{action}(\nu X.\psi) = \mathsf{action}(\psi)$. □

We can now define separability based on action sets of formulae as follows, obtained by applying the fixed-point expansion, purely probabilistic abstraction, and grouping.

Definition 3.9 (Separability)**.** *The set of all separable formulae is the largest set $\mathcal{S}$ such that $\forall\psi \in \mathcal{S}$, if $\psi' = GRP(PPA(FPE(\psi)))$, then*

1. *every subformula of* ψ' *is in* $\mathcal{S}$*, and*

2. *if* $\psi' = \psi_1 \oplus \psi_2$ *where* $\oplus \in \{\wedge, \vee\}$*, then* $\mathsf{action}(\psi_1) \cap \mathsf{action}(\psi_2) = \varnothing$*.*

A formula ψ *is* separable *if* $\psi \in \mathcal{S}$*.* □

Intuitively, all the leaves of ψ' are modal subformulae, with no action guarding more than one leaf.

Below we illustrate separability of formulae. Let ψ_1-ψ_4 be all separable and distinct, and assume $\psi_1 \vee \psi_2$ and $\psi_3 \vee \psi_4$ are also separable.

First, note that *GRP* uses only distributivity of the modal operators over "$\wedge$" and "$\vee$", and not the distributivity of the boolean operators themselves. Consequently, a separable formula may be equivalent to a non-separable formula, such as when it is written in disjunctive normal form (DNF).

Example 3.2 (Separable formula with equivalent non-separable formula)**.** *The formula* ψ_s *is separable. The DNF version of* ψ_s *is not separable since action sets of disjuncts overlap.*

$$\psi_s = [a](\psi_1 \vee \psi_2) \wedge [b](\psi_3 \vee \psi_4), \tag{3.3}$$

$$\psi'_s = ([a]\psi_1 \wedge [b]\psi_3) \vee ([a]\psi_1 \wedge [b]\psi_4) \vee ([a]\psi_2 \wedge [b]\psi_3) \vee ([a]\psi_2 \wedge [b]\psi_4). \tag{3.4}$$

This is important because we need the subformulae of a separable formula to also be separable.

Example 3.3 (Non-separable formula)**.** *The formula* ψ_e *is a subformula of* ψ'_s *(3.4), is not separable, and has no equivalent separable formula:*

$$\psi_e = ([a]\psi_1 \wedge [b]\psi_4) \vee ([a]\psi_2 \wedge [b]\psi_3). \tag{3.5}$$

With ψ_e*, we need to satisfy* ψ_1 *or* ψ_2 *following an* a *action, and likewise for* ψ_3 *or* ψ_4 *following a* b *action. An equivalent separable formula would thus have to include* $[a](\psi_1 \vee \psi_2)$ *and* $[b](\psi_3 \vee \psi_4)$*, but this would also be satisfied by,* e.g.*, outcomes satisfying only* $[a]\psi_1 \wedge [b]\psi_3$*.*

We say that a formula is *entangled* at a state if it is not (equivalent to) a separable formula even after considering that state's specific characteristics. For instance, ψ_e is entangled only at states with both a and b actions present. Even when considering only states where the actions relevant to entanglement are present, a formula may be entangled at some states and not at others.

Example 3.4 (Entanglement on a and b depends on c)**.** *The formula ψ_c reduces to ψ'_s (3.4) at states that have a c-transition, and to ψ_e (3.5) otherwise.*

$$\psi_c = ([a]\psi_1 \wedge [b]\psi_3 \wedge \langle c\rangle \mathsf{tt}) \vee ([a]\psi_1 \wedge [b]\psi_4) \vee \\ \vee ([a]\psi_2 \wedge [b]\psi_3) \vee ([a]\psi_2 \wedge [b]\psi_4 \wedge \langle c\rangle \mathsf{tt}). \quad (3.6)$$

There are also non-separable formulae that nonetheless would not be entangled at any state of an arbitrary PLTS.

Example 3.5 (Never-entangled non-separable formula)**.** *For the formula ψ_d, $PPA(\psi_d) = \psi_e$, but at any state it is equivalent either to $[a]\psi_1 \wedge [b]\psi_4$ or to $[a]\psi_2 \wedge [b]\psi_3$.*

$$\psi_d = ([a]\psi_1 \wedge [b]\psi_4 \wedge [c]\mathsf{ff}) \vee ([a]\psi_2 \wedge [b]\psi_3 \wedge \langle c\rangle \mathsf{tt}). \quad (3.7)$$

Since *GRP* groups together modal operators with a common action, we have the following important consequence.

Proposition 3.1. *All conjunctive formulae and disjunctive formulae are separable.*

3.3.2 Dependency Graph

We now outline a model checking procedure for XPL's fuzzy formulae for a fixed PLTS $L = (S, \delta, P, I)$, along similar lines to the GPL model checking algorithm (Section 2.3). The model checking procedure succeeds, in the sense of producing a polynomial system of equations, whenever the given formula is separable.

The core of the model checking algorithm is the construction of a *dependency graph* $\mathsf{Dg}(s, \psi)$, to compute $\mathsf{Pr}^{\max}_{L,s}(\psi)$. When constructing a dependency graph, in order to divide a formula by actions, we transform it into *factored* form, in a similar manner to checking separability. If we are unable to transform a formula into a factored form, as can happen when a formula is non-separable, the graph construction terminates with failure.

Definition 3.10 (Factored form)**.** *A factored formula ψ can be trivial, when $\psi \in \{\mathsf{tt}, \mathsf{ff}\}$. Otherwise, every leaf of $AO(\psi)$ is in the* action *form, $\langle a\rangle\psi'$, and no action may guard more than one leaf.* □

Given a state s, a formula ψ' can be transformed into a semantically equivalent one ψ'' that is in factored form[1] as: $\psi'' = GRP\big(PE\big(s, FPE(\psi')\big)\big)$. $PE(s,\psi')$ *partially evaluates* ψ', by evaluating non-probabilistic subformulae of ψ' as well as all unguarded modal subformulae with actions absent at state s, yielding tt or ff for each, and simplifying the result.[2] Then $\big((s,\psi'),\varepsilon,(s,\psi'')\big) \in E$.

Definition 3.11 (Dependency graph)**.** *For model checking a formula ψ with respect to a state s in PLTS L, the dependency graph, denoted by $\mathsf{Dg}(s,\psi)$, is a directed graph (N,E), where* node set $N \subseteq S \times \mathcal{AO}(Cl(\psi))$*, and* edge set $E \subseteq N \times (\mathsf{Act} \cup \{\varepsilon, \varepsilon^{\wedge}, \varepsilon^{\vee}\}) \times N$*;* i.e.*, the edges are labeled from $\mathsf{Act} \cup \{\varepsilon, \varepsilon^{\wedge}, \varepsilon^{\vee}\}$. The sets N and E are the smallest such that:*

- $(s,\psi) \in N$.
- *If $(s',\psi') \in N$, ψ' is not in factored form: if equivalent ψ'' in factored form exists, then $(s',\psi'') \in N$ and $((s',\psi'),\varepsilon,(s',\psi'')) \in E$.*
- *If $(s',\psi'_1 \oplus \psi'_2) \in N$ then $(s',\psi'_i) \in N$ for $i = 1,2$. Moreover, $((s',\psi'_1 \oplus \psi'_2),\varepsilon^{\oplus},(s',\psi'_i)) \in E$ for $i = 1,2$, and $\oplus \in \{\wedge,\vee\}$.*
- *If $(s',\langle a\rangle\psi') \in N$ then $(s'',\psi') \in N$ for each s'' such that $(s',a,s'') \in \delta$. Moreover, $((s',\langle a\rangle\psi'),a,(s'',\psi')) \in E$.* □

If $(s',\psi') \in N$ and ψ' has no factored form, then the dependency graph construction fails.

When we transform ψ' to the factored form ψ'', the semantics does not change, *i.e.*, $\Theta_{L,s'}(\psi') = \Theta_{L,s'}(\psi'')$. When (s',ψ') is a node with a factored ψ', it may be a terminal node, an *action node*, an *and*-node, or an *or*-node. For the factored formulae, standard XPL semantics apply (Table 2.1). Note that we can assume *action* nodes to be of the form $(s',\langle a\rangle\psi')$, as the action a must then be present at state s'. From these semantics, we also get the relationships for the probabilistic values.

Lemma 3.2 (Probabilistic values)**.** *Fix $\mathsf{Dg}(s_0,\psi) = (N,E)$. The probabilistic value $\mathsf{Pr}^{\max}_{L,s}(\psi')$ for a node (s,ψ') is as follows:*

[1] We may use the DNF version of ψ' to check for equivalence with existing nodes, but not for finding the factored form.

[2] After applying *GRP*, we may have a leaf in action form $\langle a\rangle\psi'_a \notin \mathcal{AO}(Cl(\psi))$. Then, we may view an action a as a prefix label on the subtree $\psi'_a \in \mathcal{AO}(Cl(\psi))$.

- $\mathsf{Pr}^{\max}_{L,s}(\mathsf{ff}) = 0$ *and* $\mathsf{Pr}^{\max}_{L,s}(\mathsf{tt}) = 1$.
- *If* (s, ψ') *is an* and*-node, then:*

$$\mathsf{Pr}^{\max}_{L,s}(\psi') = \prod_{((s,\psi'),\varepsilon^{\wedge},(s,\psi'_i))\in E} \mathsf{Pr}^{\max}_{L,s}(\psi'_i)$$

- *If* (s, ψ') *is an* or*-node, then:*

$$\mathsf{Pr}^{\max}_{L,s}(\psi') = \coprod_{((s,\psi'),\varepsilon^{\vee},(s,\psi'_i))\in E} \mathsf{Pr}^{\max}_{L,s}(\psi'_i)$$

- *If* (s, ψ') *is an action node,* i.e., $\psi' = \langle a \rangle \psi'_a$*, then:*

$$\mathsf{Pr}^{\max}_{L,s}(\psi') = \max_{c\in\mathbb{N}} \sum_{((s,\psi'),a,(s',\psi'_a))\in E} P(s, a, s', c) \cdot \mathsf{Pr}^{\max}_{L,s'}(\psi'_a)$$

- *The remaining nodes* (s, ψ') *have a unique successor* (s, ψ'') *with:*

$$\mathsf{Pr}^{\max}_{L,s}(\psi') = \mathsf{Pr}^{\max}_{L,s}(\psi'')$$

Proof. Most of the cases are straightforward and similar to the GPL model checking algorithm [CIN05, Lemma 8] and a result for two-player stochastic parity games [Mio11, Theorem 4.22]. The *and*-node and *or*-node cases have the product and coproduct, respectively, due to independence. We explain the *action node* case in more detail.

The sum over the probabilistic distribution is as in GPL and (2.2); we explain the (linear) nondeterministic choice. A PLTS scheduler makes a choice for an action given the partial computation σ. Here, this choice is made based on a formula, ψ'_a, to be satisfied. When the initial formula ψ is separable, this is well-defined: given L, s, and ψ, the scheduler can deduce ψ'_a from σ. □

We note that, although a particular choice may maximize $\mathsf{Pr}^{\max}_{L,s}(\psi')$, a scheduler that makes this choice *every time* is not necessarily optimal. Indeed, no optimal scheduler may exist, in which case we would have only ϵ-optimal schedulers for any $\epsilon > 0$ [ESY15, Mio11]. The probabilistic value may be predicated on making a different choice *eventually*. The formulation in Lemma 3.2 is consistent with this possibility, and the existence of (ϵ-)optimal schedulers may be justified through a common method, called

strategy improvement or *strategy stealing* [EY15, Mio11]. The intuition is that, in case of a loop, we can add a choice to immediately succeed with the maximum probability for the state. This cannot increase the probability, and the maximizing scheduler can otherwise be the same, if this choice does not arise.

Theorem 3.3 (Model checking termination)**.** *The graph construction of* $\mathsf{Dg}(s, \psi)$ *terminates for any XPL formula* ψ *and PLTS* L*. Moreover, if* ψ *is separable, it will always complete the construction.*

Proof. $Cl(\psi)$ is finite, so $\mathcal{AO}(Cl(\psi))$ (with DNF versions) is finite. The number of actions in L and ψ is finite, so the number of factored formulae is finite. This is sufficient to guarantee termination, as we fail when we cannot construct a factored formula. Meanwhile, separability implies that we can always construct a factored formula starting from $\psi' \in \mathcal{AO}(Cl(\psi))$. □

From the completed graph, a *system of polynomial equations* can be extracted. Equations are readily constructed by considering each $\mathsf{Pr}^{\max}_{L,s}(\psi)$ as a variable. Note that the graph treats μ and ν nodes exactly the same way. In solving the stratified system, we start from $\mathbf{0}$ to find the least fixed point, and $\mathbf{1}$ for the greatest fixed point. The alternation-free restriction ensures that there is no cycle in the graph containing both a μ-node and a ν-node. However, as with GPL [CIN05], a practical implementation may suffer from numerical issues, leading to indeterminate results when the probabilistic value is sufficiently close to the threshold against which it is compared. We make a separate argument for decidability, by appealing to the first-order theory of real closed fields, which allows for addition, subtraction, and multiplication, as well as comparison operators.

Theorem 3.4 (Soundness of computation)**.** *The construction of the dependency graph* $\mathsf{Dg}(s, \psi)$ *leads to the equation system with* $\mathsf{Pr}^{\max}_{L,s}(\psi)$ *as a solution. Comparing this probabilistic value against a threshold* p *is decidable.*

Proof. We write the min/max polynomial system, $\mathbf{x} = P(\mathbf{x})$, as a sentence in the first-order theory of real closed fields, similar to [MM15]. The additional comparison will be $x_0 > p$ or $x_0 \geq p$. Along with the equation system, we need to encode fixed points and min/max.

We can encode $x_i = \max(x_j, x_k)$ as (3.8) (cf. [EY15, Section 5]):

$$x_i \geq x_j \wedge x_i \geq x_k \wedge (x_i \leq x_j \vee x_i \leq x_k) \ . \tag{3.8}$$

Meanwhile, letting N be the set of all variables and I a subset falling mutually under an LFP, we can encode the LFP as (3.9):

$$\forall \mathbf{x}'_I . \left(\bigwedge_{i \in I} x'_i = P_i(\mathbf{x}'_I, \mathbf{x}_{N \setminus I}) \implies \bigwedge_{i \in I} x_i \leq x'_i \right) . \tag{3.9}$$

The alternation-free restriction precludes a cyclical dependency between an LFP and a GFP; a GFP can be encoded similarly.

The encoding of fixed points makes the desired solution unique, and it is compared against the threshold. □

We note that decidability does not appear to break if we had both min and max in a single system.

Example 3.6 (Model Checking XPL Formula)**.** *For the fuzzy formula* $\psi = \mu X.[a][b]X \wedge [a][c]X$ *in Example 2.2 and the PLTS in Example 3.1, we have* $\mathsf{Pr}^{\max}_{L,s_1}(\psi) = \frac{1}{4}$ *and* $\mathsf{Pr}^{\min}_{L,s_1}(\psi) = \frac{1}{9}$.

The dependency graph, in Figure 3.3, remains quite similar to our example in Section 2.3. We find $\mathsf{Pr}^{\max}_{L,s_1}(\psi)$ as the LFP from the following equations:

$$\begin{aligned}
x_1^a &= x_2^{bc}, \\
x_2^{bc} &= x_2^b \cdot x_2^c, \\
x_2^b &= \max(x_3^a, x_4^a), \\
x_2^c &= \max(x_3^a, x_4^a), \\
x_3^a &= \frac{1}{3} x_5^{bc} + \frac{2}{3} x_2^{bc}, \\
x_4^a &= \frac{1}{4} x_6^{bc} + \frac{3}{4} x_2^{bc}, \\
x_5^{bc} &= 1, \\
x_6^{bc} &= 1.
\end{aligned}$$

Solving the equations, $\mathsf{Pr}^{\max}_{L,s_1}(\psi) = x_1^a = \frac{1}{4}$. We find $\mathsf{Pr}^{\min}_{L,s_1}(\psi)$ in a similar way; note that it has the same value as in Example 2.3.

The complexity analysis is also similar to the GPL model checking case (Section 2.3), although the dependency graph is now bounded double exponentially in the size of the fuzzy formula. In many practical cases, though, it may lead to a smaller graph for the same formula, as fewer intermediate nodes are constructed and disjunctions are handled directly, rather than as *or*-nodes with three outgoing edges.

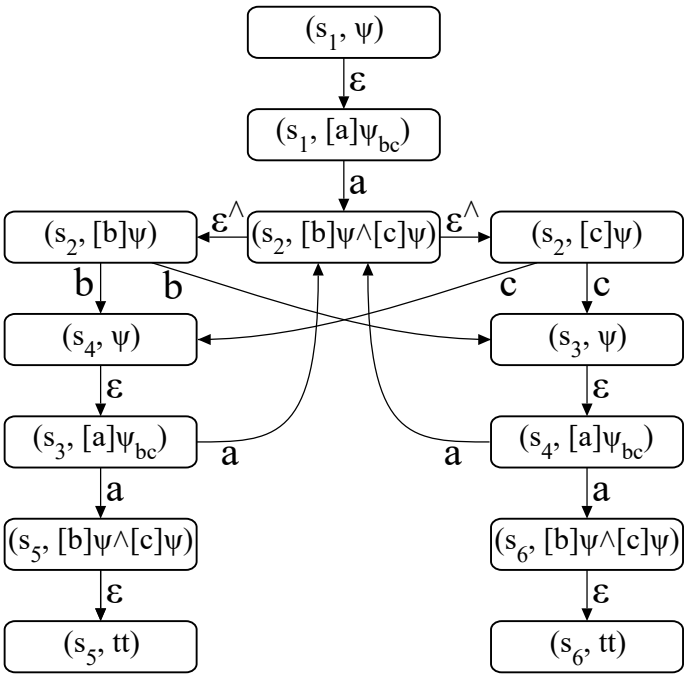


Figure 3.3: Dependency graph $\mathsf{Dg}(s_1, \psi)$

3.4 Encoding Other Model Checking Problems

In this section, we show the encoding of several model checking problems, which demonstrate various aspects of separability.

3.4.1 Encoding PCTL* over MDPs

We can encode PCTL* [Bai98] to XPL, following [CIN05, Section 3.2]. The syntax of PCTL* is then as follows, where $A \in \mathsf{Prop}$, and ϕ and ψ represent state formulae and path formulae, respectively:

$$\phi ::= A \mid \neg\phi \mid \phi \wedge \phi \mid \mathsf{Pr}_{\geq p}\psi \mid \mathsf{Pr}_{>p}\psi,$$
$$\psi ::= \phi \mid \neg\psi \mid \psi \wedge \psi \mid \mathsf{X}\psi \mid \psi \mathsf{U} \psi.$$

This is similar to the syntax given by [Bai98, Chapter 9], except omitting the bounded *until* operator. The encoding of PCTL*, $E_{PCTL^*}(\gamma)$, where γ

is either a state or path formula, is as follows:

$$E_{PCTL^*}(\gamma) = \begin{cases} \gamma, & \gamma \in Prop, \\ \mathsf{neg}(E_{PCTL^*}(\gamma')), & \gamma = \neg\gamma', \\ E_{PCTL^*}(\gamma_1) \wedge E_{PCTL^*}(\gamma_2), & \gamma = \gamma_1 \wedge \gamma_2, \\ \mathsf{Pr}^{\max}_{>p} E_{PCTL^*}(\psi), & \gamma = \mathsf{Pr}_{>p}\psi, \\ \mathsf{Pr}^{\max}_{\geq p} E_{PCTL^*}(\psi), & \gamma = \mathsf{Pr}_{\geq p}\psi, \\ \langle a \rangle E_{PCTL^*}(\psi), & \gamma = \mathsf{X}\psi, \\ \mu X. E_{PCTL^*}(\psi_2) \vee (E_{PCTL^*}(\psi_1) \wedge \langle a \rangle X), & \gamma = \psi_1 \mathsf{U} \psi_2. \end{cases}$$

Since PCTL* does not distinguish between action labels, we translate MDPs to PLTSs by retaining the same transition structure, but renaming all labels to a. Note that formulae are trivially separable if they have only one action. Additionally, the d-trees become paths in this case. Consequently, we can model check PCTL* formulae over MDPs using our encoding and model checking algorithm.

3.4.2 Encoding of RMDP Termination

We consider a Recursive MDP (RMDP) [EY15] as a nondeterministic extension of RMCs [EY09], which we described in Section 2.4. We discuss a more general model, called a Recursive Simple Stochastic Game (RSSG); formally, an RSSG A is a tuple $(A_1, \ldots, A_k)$, where each *component graph* A_i is a septuple $(N_i, B_i, Y_i, \mathsf{En}_i, \mathsf{Ex}_i, \mathsf{pl}_i, \delta_i)$:

- N_i is a set of nodes, containing subsets En_i and Ex_i of entry and exit nodes, respectively.
- B_i is a set of boxes, with a mapping $Y_i : B_i \to \{1, \ldots, k\}$ assigning each box to a component. Each box has a set of call and return ports, corresponding to the entry and exit nodes, respectively, in the corresponding components: $\mathsf{Call}_b = \{(b, en) \mid en \in \mathsf{En}_{Y_i(b)}\}$, $\mathsf{Return}_b = \{(b, ex) \mid ex \in \mathsf{Ex}_{Y_i(b)}\}$. Additionally, we have:

$$\mathsf{Call}^i = \bigcup_{b \in B_i} \mathsf{Call}_b,$$
$$\mathsf{Return}^i = \bigcup_{b \in B_i} \mathsf{Return}_b,$$
$$Q_i = N_i \cup \mathsf{Call}^i \cup \mathsf{Return}^i.$$

- $\mathsf{pl}_i : Q_i \to \{0, 1, 2\}$ is a mapping that specifies whether, at each state, the choice is probabilistic (*i.e.*, *player* 0), or nondeterministic (player 1: maximizing, player 2: minimizing). As any $u \in \mathsf{Call}^i \cup \mathsf{Ex}_i$ has no outgoing transitions, let $\mathsf{pl}_i(u) = 0$ for these states.

- δ_i is the transition relation, with transitions of the form (u, p_{uv}, v), when $\mathsf{pl}_i(u) = 0$ and u is not an exit node or a call port, and v may not be an entry node or a return port. Additionally, $p_{uv} \in (0, 1]$ and, for each u, $\sum_{v':(u,\cdot,v')\in\delta_i} p_{uv'} = 1$. Meanwhile, the nondeterministic extension yields transitions of the form $(u, \bot, v)$ when $\mathsf{pl}_i(u) > 0$. □

RMDPs only have a player 1 or player 2, depending on whether they are maximizing or minimizing, respectively. Termination probabilities can be computed for 1-RSSGs, and are always achieved, for both players, with a strategy limited to a class called stackless and memoryless (SM) [EY15]. The essence of SM strategies is that in each nondeterministic choice, the selection is fixed to a single state from its distribution, which makes the resolution of the nondeterministic choices substantially simpler than in the general case. For multi-exit RSSGs, the termination probability is *determined* [Mar98], although an optimal strategy may not exist, and the problem of computing or approximating the probability is undecidable, in general. SM strategies are inadequate even for 2-exit RMDPs [EY15].

Figure 3.4 shows an RMDP with two components, A and B. Any call to A nondeterministically results in either a call to B (via box b_1) or a transition to u.

Translating RMDPs to PLTSs

Given an RMDP A, we can define a translated PLTS L that models A, with $\mathsf{Act} = \{p, n, c, r_i, e_i\}$ and states of the PLTS corresponding to nodes of the RMDP. We retain the RMDP's transitions, labeling them as n for actions from a nondeterministic choice and p for probabilistic choice. To this basic structure we add three new kinds of edges:

- e_i for the ith exit node of a component,

- c edges from a call port to the called component's entry node, and

- r_i edges from a call port to each return port in the box.

While c edges denote control transfer due to a call, r edges summarize returns from the called procedure. Figure 3.4 shows the result of the translation for one component of the RMDP. Formally, we define the PLTS L as follows:

Definition 3.12 (Translated RMDP)**.** *The translated RMDP A is a PLTS $L = (S, \delta, P, I)$:*

- *The set of states S is the set of all the nodes, as well as the call and return ports of the boxes,* i.e., *$S = \bigcup_i Q_i$. Additionally, we associate a consistent index with each state corresponding to an exit node or a return port.*

- *The transition relation δ has all the transitions of the components, labeled by action p for the probabilistic transitions and n for the nondeterministic ones. Thus, when $(u, p_{uv}, v) \in \delta_i$ for any i, then $(u, p, v) \in \delta$, and when $(u, \bot, v) \in \delta_i$, $(u, n, v) \in \delta$. Additionally,$((b, en), c, en) \in \delta$ and $((b, en), r_i, (b, ex_i)) \in \delta$ for every box b, and $(ex_i, e_i, ex_i) \in \delta$ for every exit node. Note the indices used.*

- *The transition probability distribution P is defined as $P(u, p, v, \cdot) = p_{uv}$ as given for the RMDP A, $P(u, n, v, c(v)) = 1$, where $c : S \to \mathbb{N}$ is a one-to-one function (when $c \neq c(v)$ for any v with $(u, n, v) \in \delta$, $P(u, n, v, c) = 1$ for an arbitrary v with $(u, n, v) \in \delta$), and $P(\cdot) = 1$ if the action is not p or n.*

- *We do not use the interpretation in the translation,* i.e., *$I(s) = \varnothing$ for any state s, unless additional relevant information about the RMDP A is available.* □

For RMCs, the translation yields an RPLTS L (no n actions).

Intuitively, L preserves all the non-recursive transition structure of A via the actions labeled by p and n. There are additional c actions to model call transitions. Note that each call port will have a single outgoing c transition, while the entry nodes may have multiple incoming c transitions. Meanwhile, we need a different design to associate exit nodes with return ports, as an exit node may be associated with multiple return ports. Thus, we have indexed e and r actions and require a standard formula to model termination. We note that the resulting structure is similar to the Nested State Machines (NSMs) [ACM11], with the p/n, c, r_i, and e_i edges corresponding to the loc (local), call, *jump*, and ret edges, respectively, in the NSM model.

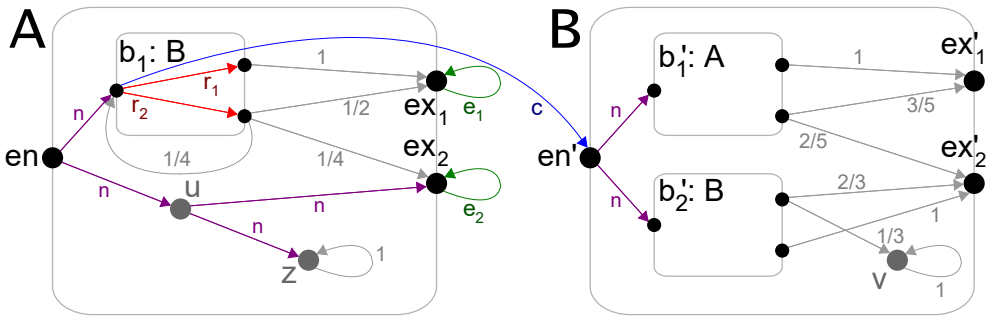


Figure 3.4: Example RMDP with Call, Return, and Exit edges added to A.

Termination of 1-RMDPs can be encoded as the following separable formula:

$$\psi_1 = \mu X.\langle e_1\rangle \mathsf{tt} \vee \langle p\rangle X \vee \langle n\rangle X \vee (\langle c\rangle X \wedge \langle r_1\rangle X). \tag{3.10}$$

Note that the probabilistic branching time appears on c and r actions, used to model the recursive RMDP call.

The corresponding formula for a 2-exit RMDP, is not separable:

$$\begin{aligned} \psi_2^1 &=_\mu \langle e_1\rangle \mathsf{tt} \vee \langle p\rangle \psi_2^1 \vee \langle n\rangle \psi_2^1 \vee (\langle c\rangle \psi_2^1 \wedge \langle r_1\rangle \psi_2^1) \vee (\langle c\rangle \psi_2^2 \wedge \langle r_2\rangle \psi_2^1), \\ \psi_2^2 &=_\mu \langle e_2\rangle \mathsf{tt} \vee \langle p\rangle \psi_2^2 \vee \langle n\rangle \psi_2^2 \vee (\langle c\rangle \psi_2^1 \wedge \langle r_1\rangle \psi_2^2) \vee (\langle c\rangle \psi_2^2 \wedge \langle r_2\rangle \psi_2^2). \end{aligned} \tag{3.11}$$

3.4.3 PTTL and Branching Processes

A Branching MDP (BMDP) is an extension of BPs, which we discussed in Section 2.6 (there are also the more general Branching Simple Stochastic Games (BSSGs) [EY15]), and we will define BMDPs with notation more closely mirroring our definitions for RPLTSs and BPs. Formally, with respect to a set Act of actions, a BMDP Δ is a triple (Γ, δ, P), where:

- Γ is a finite set of types;
- $\delta \subseteq \Gamma \times \mathsf{Act} \times \Gamma^*$ is a finite set of transition rules; and
- $P : \delta \to (0, 1]$ is a transition probability distribution satisfying:

$$\forall X \in \Gamma. \forall a \in \mathsf{Act}. \sum_{\alpha:(X,a,\alpha)\in\delta} P(X, a, \alpha) = 1 \ .$$

A maximizing (or minimizing) player is assumed to be choosing the actions for each type (in a BSSG, the types are partitioned into two sets, one for each player). The problem of BSSG extinction is reducible to 1-RSSG termination, and also admits optimal *static* strategies (the analogue to SM strategies for RSSGs) [EY15].

Recall that we described PTTL in Section 2.6.2. We can extend PTTL model checking [CDK12] to support BMDPs analogously to our PCTL* extension. As with RMDPs, we can readily translate BPs and BMDPs to PLTSs. We give the semantics for PTTL (assuming maximizing schedulers) over PLTSs without terminal states by encoding it in XPL, as $E_{PTTL}(\gamma)$.

$$E_{PTTL}(\gamma) = \begin{cases} \gamma, & \gamma \in Prop \cup \{\mathsf{tt}\}, \\ \mathsf{neg}(E_{PTTL}(\gamma')), & \gamma = \neg\gamma', \\ E_{PTTL}(\gamma_1) \wedge E_{PTTL}(\gamma_2), & \gamma = \gamma_1 \wedge \gamma_2, \\ \mathsf{Pr}^{\max}_{>p} E_{PTTL}(\psi), & \gamma = \mathsf{Pr}_{>p}\psi, \\ \mathsf{Pr}^{\max}_{\geq p} E_{PTTL}(\psi), & \gamma = \mathsf{Pr}_{\geq p}\psi, \\ [-]E_{PTTL}(\phi), & \gamma = \mathsf{AX}\phi, \\ \mu X.E_{PTTL}(\phi_2) \vee (E_{PTTL}(\phi_1) \wedge [-]X), & \gamma = \mathsf{A}[\phi_1 \mathsf{U} \phi_2], \\ \nu X.E_{PTTL}(\phi_2) \wedge (E_{PTTL}(\phi_1) \vee [-]X), & \gamma = \mathsf{A}[\phi_1 \mathsf{R} \phi_2]. \end{cases}$$

For the $\mathsf{E}[\psi]$ versions of X, U, and R, we replace the *boxes* with *diamonds*.

In this sense, PTTL is a natural extension of PCTL over PLTSs for the case $|\mathsf{Act}| > 1$, and all the formulae in the encoding are separable. Additionally, although a limitation of GPL is that it cannot encode CTL's $\mathsf{EG}\phi$ when $|\mathsf{Act}| = 1$ [CKP15], there is a PTTL formula of the form $\mathsf{EG}\phi \equiv \mathsf{E}[\mathsf{ff}\,\mathsf{R}\phi]$, which we can encode with XPL (and GPL over RPLTSs).

3.5 Conclusion and Related Work

Previous attempts to extend GPL included allowing systems with internal nondeterminism that was still branching [CI00], and EGPL, which had similar syntax and semantics to XPL, but limited the model checking to non-recursive formulae [Son04].

Following GPL, XPL treats conjunction in a traditional manner, retaining the properties that $\psi \wedge \neg\psi = \mathsf{ff}$, and $\psi \wedge \psi = \psi$ for any formula ψ. However, the probability value of $\psi_1 \wedge \psi_2$ cannot be computed based on the probability

values of the conjuncts ψ_1 and ψ_2. This makes model checking in XPL more complex, but also contributes to its expressiveness.

Another probabilistic extension of μ-calculus is pLμ. In contrast to XPL, the most expressive version of pLμ, denoted pL$\mu^{\odot}_{\oplus}$ [Mio11, Mio12], defines *three* conjunction operators and their duals such that their probability values can be computed from the probabilities of the conjuncts. The three conjunctions are defined as minimum, independent product, and truncated co-sum. The logic pL$\mu^{\odot}$ is able to support branching time and an intuitive game semantics [Mio11]. Along the same lines as our XPL encoding, we can encode termination of 1-exit RMDPs as model checking in pL$\mu^{\odot}$, and RMC termination in pL$\mu^{\odot}_{\oplus}$. However, attempting to encode multi-exit RMDP termination in pL$\mu^{\odot}_{\oplus}$ similarly to multi-exit RMC termination would lead to an incorrect, rather than undecidable, encoding. The scope of pLμ includes infinite-state systems as well. Determining the relationship between XPL and pL$\mu^{\odot}$ in branching time is an important problem. Other recent probabilistic extensions of μ-calculus include the Lukasiewicz μ-calculus [MS13a] and μ^p-calculus [CKP15], which can encode PCTL* over MDPs, and PμTL [LSWZ15], but all these limit nondeterminism to the linear-time semantics.

Although these systems are closely related, algorithms to check properties of RMCs (and probabilistic pushdown systems [EKM04]) were developed independently [EY09]. These were related to algorithms for computing properties of systems such as BP extinction and the language probability of stochastic context-free grammars, which were also phrased in terms of solving a set of polynomial equations. The relationship between GPL and these systems was mentioned briefly in [GRS12], but has remained largely unexplored.

There has been significant interest in the study of expressive systems with linear and branching nondeterminism, such as RMDPs and BMDPs [EY15]. At the same time, the understanding of the polynomial systems has expanded. In [ESY12a], the Probabilistic Polynomial System (PPS) class is introduced, which characterizes when efficient solutions to polynomial equation systems are possible even in the worst case [ESY12b]. While the systems arising from 1-RMCs were not initially distinguished from those from multi-exit RMCs [EY09], the PPS class is limited to 1-RMCs. It was also extended for RMDP termination, and later BMDP reachability, both having polynomial-time complexity for min/maxPPSs [ESY12a, ESY15].

Systems producing equations in PPS form show an interesting charac-

teristic: that the properties are expressible as purely conjunctive or purely disjunctive formulae. Recall that such formulae are trivially separable. Polynomial systems equivalent to those arising from separable GPL have recently been considered in the context of *game automata* [MM15], suggesting that we may be able to lift the alternation-free restriction from separable XPL. Characterizing equation systems that arise from separable formulae and investigating their efficient solution is an interesting open problem.

Chapter 4

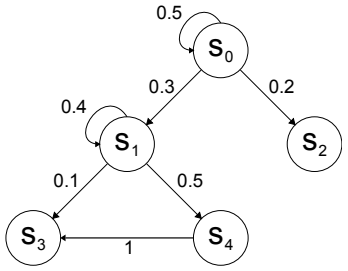
Model Checking with Logic Programming

While we described a GPL model checking algorithm in Section 2.3 and further extended it to separable XPL in Section 3.3, there remains a question of a practical implementation. The idea may actually be flipped in intent, so, while XPL is the featured application, the focus in this chapter is to build on the idea of performing model checking via query evaluation over logic programs [RRR+97].

The attractiveness of this approach is that the operational semantics of complex process languages (originally CCS [Mil89], followed by value-passing calculi [Ram01], the π-calculus [MPW92], and mobile calculi with local broadcast [SRS08]), as well as the semantics of complex temporal logics (*e.g.*, the modal μ-calculus [Koz83]), can be expressed naturally and at a high level as clauses in a logic program. Model checking over these languages and logics then becomes query evaluation over the logic programs that directly encode their semantics.

Starting in the 1990s, there have been a number of important developments in Probabilistic Logic Programming (PLP), combining logical and statistical inference, and leading to a number of increasingly mature PLP implementations. A natural question is whether the advances in PLP enable the development of model checkers for *probabilistic systems*, the same way traditional logic programming (LP) methods such as tabled evaluation and constraint handling enabled us to formulate model checkers for a variety of non-probabilistic systems.

Prior to PIP [GRS12], existing PLP inference methods were not sufficiently powerful to be used as a basis for probabilistic model checking. One of the earliest PLP inference procedures, used in PRISM [SK97], was formulated in terms of the set of *explanations* of answers. PRISM put in place three restrictions to make its inference work: (a) *independence*: random variables used in any single explanation are all independent; (b) *mutual ex-*



(a)

```
% 3 "switches" (random processes) for transitions
% from states s0, s1 and s4, respectively.
values(t(s0), [s0, s1, s2]).
values(t(s1), [s1, s3, s4]).
values(t(s4), [s3]).

% Distribution parameters of the random variables.
set_sw(t(s0), [.5, .3, .2]).
set_sw(t(s1), [.4, .1, .5]).
set_sw(t(s4), [1]).

% Transition from S at instance I goes to T, as
% determined by the corresponding random process.
trans(S, I, T) :-  msw(t(S), I, T).

% Starting at S at instance I, T is reachable.
reach(S, I, T) :-
    trans(S, I, U),
    reach(U, next(I), T).
reach(S, _, S).
```

(b)

Figure 4.1: (a) Example Markov chain; (b) PRISM encoding of transitions in the chain.

clusion: two distinct explanations of a single answer are mutually exclusive; and (c) *finiteness*: the number of possible explanations of an answer is finite.[1] Subsequent systems, notably ProbLog [DRKT07] and Probabilistic Inference with Tabling and Answer Subsumption (PITA) [RS10a] eliminated the independence and mutual exclusion restrictions of PRISM (but retained the finiteness assumption). This, however, was still insufficient for model checking, as the following example shows.

Motivating Example: Figure 4.1 shows a Markov chain and its representation in PRISM. In any execution of the chain, a transition from a state, say s, is independent of any previous transitions (including those from the same state). The definition of the trans predicate has an explicit instance parameter I, which is also used in msw, the *switch* used to encode probabilistic facts. PRISM treats different instances of the same random variable as independent. Thus trans correctly encodes the semantics of the Markov chain.

We first consider simple reachability questions of the form: What is the

[1]The finiteness assumption has since been lifted in PRISM "at no extra cost" via *cyclic explanation graphs* [SM14].

likelihood that on an execution of the chain from a start state s, a final state t will be reached? The reachability question using the reach predicate is defined in Figure 4.1(b). Consider the likelihood of reaching state s_3 from s_0. This query can be posed as the predicate $\mathsf{prob}(\mathsf{reach}(s_0, 0, s_3), P)$, where $\mathsf{prob}/2$ finds the probability of answers (P) to a given query $\mathsf{reach}(s_0, 0, s_3)$.

The query $\mathsf{prob}(\mathsf{reach}(s_0, 0, s_3), P)$ could not be evaluated in PRISM with the finiteness assumption. We illustrate this by first describing PRISM's inference at a high level. In PRISM, inference of probabilities proceeded in the same way as logical inference, except when the selected literal is an msw. In this case, the inference procedure enumerated the values of the random variable and continues the inference for each value (by backtracking). The probability of a derivation is simply the product of the probabilities of the random variables (msw outcomes) used in that derivation (under the independence assumption). The probability of a query answer is the sum of probabilities of the set of all derivations for that answer (using the mutual-exclusion and finiteness assumptions). Note that, due to the presence of cycles in this Markov chain, $\mathsf{reach}(s_0, 0, s_3)$ has infinitely many derivations, and hence, under the finiteness assumption, PRISM could not infer its probability.

Markov chains can be encoded in ProbLog and Logic Programs with Annotated Disjunctions (LPAD) [VVB04] in a similar manner. The sequence of random-variable valuations used in the derivation of an answer is called an *explanation*. In contrast to PRISM, ProbLog [DRKT07] and PITA [RS10a], which is an implementation of LPAD, materialize the set of explanations of an answer in the form of a Binary Decision Diagram (BDD). Probabilities are subsequently computed based on the BDD. This approach permits these systems to correctly infer probabilities even when the independence and mutual-exclusion assumptions are violated. However, the *set of explanations* of $\mathsf{reach}(s_0, 0, s_3)$ is infinite.[2] Since BDDs can only represent finite sets, the probability of $\mathsf{reach}(s_0, 0, s_3)$ cannot be computed in ProbLog or LPAD.

To correctly infer the probability of $\mathsf{reach}(s_0, 0, s_3)$, we need an algorithm that works even when the set of explanations is infinite. Moreover, it is easy to construct queries where the independence and mutual exclusion properties do not hold. For example, consider the problem of inferring the prob-

[2]This is in contrast to the link-analysis examples used in ProbLog and PITA [RS10b], where, even though the number of derivations for an answer may be infinite, the number of explanations is finite.

ability of reaching s_3 or s_4 (*i.e.*, the query $\mathsf{reach}(s_0, 0, s_3)$; $\mathsf{reach}(s_0, 0, s_4)$). Since some paths to s_3 pass through s_4, explanations for $\mathsf{reach}(s_0, 0, s_3)$ and $\mathsf{reach}(s_0, 0, s_4)$ are not mutually exclusive. The example of Figure 4.1 illustrates that, to build model checkers based on PLP, we need an inference algorithm that works even when the finiteness, mutual-exclusion, and independence assumptions are simultaneously violated.

Summary of Contributions: In this chapter, we discuss "Probabilistic Inference Plus" (PIP) [GRS12], an algorithm for inferring probabilities of queries in a probabilistic logic program. PIP is applicable even when explanations are not necessarily mutually exclusive or independent and the number of explanations is infinite. We demonstrate the utility of this inference algorithm by constructing model checkers for a rich class of probabilistic models and temporal logics (see Section 4.4), including those for PCTL properties of Markov chains; reachability properties in RMCs; and GPL properties of RPLTSs. Our model checkers are based on high-level, logical encodings of the semantics of the process languages and temporal logics, thus retaining the highly declarative nature of the prior work on model checking non-probabilistic systems on which we build.

We have implemented our inference algorithm in XSB Prolog [SW+12]. Based on this implementation, we have encoded probabilistic model checkers for a variety of temporal logics, including PCTL and GPL (which subsumes PCTL*), and for process languages of varying complexity (from Reactive Modules to RMCs). Our experimental results show that, despite the highly declarative nature of our encodings of the model checkers, their performance is competitive with their native implementations.

The rest of this chapter develops along the following lines. Section 4.2 provides requisite background on PLP. Section 4.3 presents our PIP algorithm. Section 4.4 describes our PLP encodings of probabilistic model checkers, while Section 4.5 contains our experimental evaluation. Section 4.6 offers some concluding remarks.

4.1 Related Work

There is a substantial body of prior work on encoding complex model checkers as logic programs. These approaches range from using constraint handling to represent sets of states such as those that arise in timed systems [GP97,

DRS00, MP00, PRR02], data-independent systems [SSR03], as well as other infinite-state systems [DP99, MP99]; tabling to handle fixed point computation [RRS+00, FL04]; and procedural aspects of proof search to handle names [YRS04] and GFPs [GBM+07]. However, all these works deal only with non-probabilistic systems.

Moreover, most of these works exploited existing logic programming techniques to implement model checkers for novel systems. In contrast, we find that existing techniques for probabilistic inference in PLP are not sufficient for the model checking of probabilistic systems. This chapter discusses PIP [GRS12], which is applicable to PLPs in general, and also enables model checkers for probabilistic systems to be constructed at the same high level as those for non-probabilistic systems.

With regard to related work on probabilistic inference, Statistical Relational Learning (SRL) has emerged as a rich area of research into languages and techniques for supporting modeling, inference and learning using a combination of logical and statistical methods [GT07]. Some SRL techniques, including Bayesian Logic Programs [KDR01b], Probabilistic Relational Models [FGKP99] and Markov Logic Networks [RD06], use logic to compactly represent statistical models. Others, such as Stochastic Logic Programs [Mug96], PRISM [SK97], CLP(BN) [SCPQC03], ProbLog [DRKT07], LPAD [VVB04], Independent Choice Logic [Poo08], and CP-Logic [VDB09], define inference primarily in logical terms, subsequently assigning statistical properties to the proofs. Motivated primarily by knowledge representation problems, these works have been naturally restricted to cases where the models and the inference proofs are finite. Recently, a number of techniques have generalized these frameworks to handle random variables that range over continuous domains (*e.g.*, [KDR01a, NBKJ10, WD08, GJD10, GTK+11, IRR12]), but they still restrict proof structures to be finite.

Modeling and analysis of probabilistic systems, both in discrete and continuous time, has been an actively researched area. PCTL [HJ94] is a widely used temporal logic for specifying properties of discrete-time probabilistic systems. PCTL* [Bai98] is a probabilistic extension of LTL and is more expressive than PCTL. GPL [CIN05] is a probabilistic branching-time variant of the modal μ-calculus. The PRISM model checker [KNP11] is a leading tool for modeling and verifying a wide variety of probabilistic systems: Markov chains and MDPs. There is also prior work on techniques for verifying more expressive probabilistic systems, including RMCs [EY09] and probabilistic pushdown systems [KEM06], both of which exhibit context-free behavior.

The probability of reachability properties in such systems is computed as the least solution to a corresponding set of monotone polynomial equations. PReMo [WE07] is a model checker for RMCs. RPLTSs [CIN05] generalize Markov chains by adding branching nondeterminism, and PLTSs [Seg95, Mio12] also have linear nondeterminism. GPL and XPL properties of such systems are also computed as a solution to a set of probabilistic polynomial equations. This chapter describes a practical implementation of a GPLXPL model checker [GRS12].

4.2 Preliminaries

Notations: The root symbol of a term t is denoted by $\pi(t)$ and its ith subterm by $\arg_i(t)$. Following traditional LP notation, a term with a predicate symbol as root is called an *atom*. The set of variables in a term t is denoted by $\mathsf{vars}(t)$. A term t is *ground* if $\mathsf{vars}(t) = \varnothing$. We also use the standard notions from LP such as derivation and substitution [NM95]. We denote the language of a grammar G by $\mathcal{L}_G$. For any string s in $\mathcal{L}_G$, the set of symbols in s is denoted by $\mathsf{sym}(s)$.

Following PRISM, a *probabilistic logic program* is of the form $P = P_F \cup P_R$, where P_R is a definite logic program and P_F is the set of all possible $\mathsf{msw}/3$ atoms. The set of possible msw atoms and the distribution of their subsets is given by values and $\mathsf{set\_sw}$ directives, respectively. For example, clauses trans and reach in Figure 4.1(b) are in P_R. The set P_F of that program contains msw atoms such as:

$$\mathsf{msw}(t(s_0), 0, s_0), \qquad \mathsf{msw}(t(s_0), 0.s_0, s_0),$$
$$\mathsf{msw}(t(s_0), 0, s_1), \qquad \mathsf{msw}(t(s_1), 0.s_1, s_1).$$

In an atom of the form $\mathsf{msw}(t_1, t_2, t_3)$, t_1 is a term representing a random process (*switch* in PRISM terminology), t_2 is an instance, and t_3 is the outcome of the process at that instance. According to PRISM semantics, two msw atoms with distinct processes or distinct instances are independent. Two msw atoms with the same process and instance but different outcomes are mutually exclusive.

4.3 The Inference Procedure PIP

A key idea behind the PIP inference algorithm [GRS12] is to represent the (possibly infinite) set of explanations in a symbolic form. Observe from

the example in Figure 4.1 that, even though the set of paths (each with its own distinct probability) from state s_0 to state s_3 is infinite, the regular expression ${s_0}^+{s_1}^+{s_4}^?s_3$ captures this set exactly. Following this analogy, we devise a grammar-based notation that can succinctly represent infinite sets of finite sequences.

Definition 4.1 (Explanation)**.** *An* explanation *of an atom A with respect to a program $P = P_F \cup P_R$ is a set $\xi \subseteq P_F$ of* msw *atoms such that (i) $\xi, P_R \vdash A$ and (ii) ξ is consistent,* i.e.*, it contains no pair of mutually exclusive* msw *atoms.*

The set of all explanations of A w.r.t. P is denoted by $\mathcal{E}_P(A)$. □

Example 4.1 (Set of explanations)**.** *Consider the program of Figure 4.1(b). The set of explanations for* $\mathsf{reach}(s_0, 0, s_3)$ *is:*

$$\mathsf{msw}(t(s_0), 0, s_1),\ \mathsf{msw}(t(s_1), 0.s_1, s_3).$$
$$\mathsf{msw}(t(s_0), 0, s_0),\ \mathsf{msw}(t(s_0), 0.s_0, s_1),\ \mathsf{msw}(t(s_1), 0.s_0.s_1, s_3).$$
$$\vdots$$
$$\mathsf{msw}(t(s_0), 0, s_1),\ \mathsf{msw}(t(s_1), 0.s_1, s_1),\ \mathsf{msw}(t(s_1), 0.s_1.s_1, s_3).$$
$$\vdots$$

4.3.1 Representing Explanations

As Example 4.1 illustrates, a representation in which instance identifiers are explicitly captured will not be nearly as compact as the corresponding regular expression (shown earlier). On the other hand, a representation (like the regular expression) that completely ignores instance identifiers will not be able to identify identical instances of a random process or properly distinguish distinct ones.

We solve this problem by observing that, in PRISM's semantics, different instances of the same random process are *independent and identically distributed* (i.i.d.). Consequently, the probability of $\mathsf{reach}(s_0, 0, s_3)$ (reaching s_3 from s_0 starting at instance 0) is the same as that of $\mathsf{reach}(s_0, H, s_3)$ for any instance H. Hence, it is sufficient to infer probabilities for a single parameterized instance. Below, we formalize the set of PLP programs for which such an abstraction is possible.

Definition 4.2 (Temporal PLP)**.** *A* temporal *probabilistic logic program is a probabilistic logic program P with declarations of the form* $\mathsf{temporal}(\mathsf{p}/n\text{–}i)$*, where* p/n *is an* n*-ary predicate, and* i *is an argument position (between* 1 *and* n*) called the* instance argument *of* p/n*. Predicates* p/n *in such declarations are called* temporal predicates. □

The set of temporal predicates in a temporal program P is denoted by $\mathsf{temporal}(P)$; the set of all predicates in P is denoted by $\mathsf{preds}(P)$. By convention, every program contains an implicit declaration $\mathsf{temporal}(\mathsf{msw}/3\text{–}2)$, indicating that $\mathsf{msw}/3$ is a temporal predicate, and its second argument is its instance argument. The instance argument of a predicate p/n is denoted by $\chi(p/n)$. For example, the program of Figure 4.1(b) becomes a temporal program when $\mathsf{temporal}(\mathsf{trans}/3\text{–}2)$ and $\mathsf{temporal}(\mathsf{reach}/3\text{–}2)$ are added. For this program,

$$\mathsf{temporal}(P) = \{\mathsf{reach}/3, \mathsf{trans}/3, \mathsf{msw}/3\}, \text{ and}$$
$$\chi(\mathsf{reach}/3) = \chi(\mathsf{trans}/3) = \chi(\mathsf{msw}/3) = 2.$$

Let α be an atom in a temporal program such that its root symbol is a temporal predicate, *i.e.*, $\pi(\alpha) \in \mathsf{temporal}(P)$. Then the *instance* of α, denoted by $\chi(\alpha)$ by overloading the symbol χ, is $\arg_{\chi(\pi(\alpha))}(\alpha)$. We also denote, by $\overline{\chi}(\alpha)$, a term constructed by *omitting* the instance of α; *i.e.*, if $\alpha = f(t_1, \ldots, t_{i-1}, t_i, t_{i+1}, \ldots, t_n)$ and $\chi(\alpha) = t_i$, $\overline{\chi}(\alpha) = f(t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n)$.

Explanations of a temporal program can be represented by a notation similar to Definite Clause Grammars (DCGs).

Example 4.2 (Set of explanations using DCG notation)**.** *Considering again the program of Figure 4.1(b), the set of explanations for* $\mathsf{reach}(s_0, H, s_3)$ *can be succinctly represented by the following DCG:*

$$\mathsf{expl}(\mathsf{reach}(s_0, s_3), H) \longrightarrow [\mathsf{msw}(t(s_0), H, s_0)], \mathsf{expl}(\mathsf{reach}(s_0, s_3), H.s_0).$$
$$\mathsf{expl}(\mathsf{reach}(s_0, s_3), H) \longrightarrow [\mathsf{msw}(t(s_0), H, s_1)], \mathsf{expl}(\mathsf{reach}(s_1, s_3), H.s_1).$$
$$\mathsf{expl}(\mathsf{reach}(s_1, s_3), H) \longrightarrow [\mathsf{msw}(t(s_1), H, s_1)], \mathsf{expl}(\mathsf{reach}(s_1, s_3), H.s_1).$$
$$\mathsf{expl}(\mathsf{reach}(s_1, s_3), H) \longrightarrow [\mathsf{msw}(t(s_1), H, s_3)], \mathsf{expl}(\mathsf{reach}(s_3, s_3), H.s_3).$$
$$\mathsf{expl}(\mathsf{reach}(s_1, s_3), H) \longrightarrow [\mathsf{msw}(t(s_1), H, s_4)], \mathsf{expl}(\mathsf{reach}(s_4, s_3), H.s_4).$$
$$\mathsf{expl}(\mathsf{reach}(s_3, s_3), H) \longrightarrow [].$$
$$\mathsf{expl}(\mathsf{reach}(s_4, s_3), H) \longrightarrow [\mathsf{msw}(t(s_4), H, s_3)], \mathsf{expl}(\mathsf{reach}(s_3, s_3), H.s_3).$$

Note that each expl generates a sequence of msws. For this example, it is also the case that in a string generated from $\mathsf{expl}(\mathsf{reach}(s_0, s_3), H)$, the msws all have instances equal to or later than H. It is then immediate that $\mathsf{msw}(t(s_0), H, s_0)$ is independent of *any* msw from $\mathsf{expl}(\mathsf{reach}(s_0, s_3), H.s_0)$. This property holds for an important subclass called *temporally well-formed programs*, defined as follows.

Definition 4.3 (Temporally Well-Formed PLP)**.** *A temporal program P is* temporally well formed *when, for each clause $(\alpha \coloneq \beta_1, \ldots, \beta_n) \in P$:*

1. *If $\pi(\alpha) \in \mathsf{temporal}(P)$, then $\forall i, 1 \leq i \leq n$, s.t. $\pi(\beta_i) \in \mathsf{temporal}(P)$, $\chi(\beta_i) = \chi(\alpha)$, or $\chi(\alpha)$ is a subterm of $\chi(\beta_i)$.*
2. *If $\pi(\alpha) \notin \mathsf{temporal}(P)$, then there is at most one i, $1 \leq i \leq n$, s.t. $\pi(\beta_i) \in \mathsf{temporal}(P)$.*
3. *Instance arguments $\chi(\alpha)$ or $\chi(\beta_i)$ or their subterms are unified only with other instance arguments, their subterms, or ground terms.* □

The first condition ensures that instances of predicates on the right-hand side (RHS) of a clause are no earlier than those of the left-hand side (LHS). The second condition ensures that a common temporal instance is created for related temporal predicates. The final condition ensures that the effects of first two are not undone by tainting the temporal arguments.

We represent the set of explanations of a temporal atom α by a special term of the form $\mathsf{expl}(\overline{\chi}(\alpha), \chi(\alpha))$. The sets of explanations for an atom can be represented succinctly by DCGs. Such DCGs are called explanation generators.

An atom β is said to be *probabilistic* if it depends on an msw atom; *i.e.*, an msw atom is probabilistic, and an atom β is probabilistic if some clause whose head unifies with β has a probabilistic atom on the RHS. For a probabilistic atom β, let $e_\beta = [\beta]$ if $\beta = \mathsf{msw}(r, t, v)$, $e_\beta = \mathsf{expl}(\overline{\chi}(\beta), \chi(\beta))$ if β is a temporal atom, and $e_\beta = \mathsf{expl}(\overline{\chi}(\beta), \bot)$ if β is not temporal. A time-abstracted derivation of a query Q is a derivation constructed by ignoring bindings to temporal arguments.

Definition 4.4 (Explanation Generator)**.** *Let P be a temporally well-formed program and let Q be a query. Then the explanation generator for Q w.r.t. P, denoted by Γ, is a DCG with non-terminals of the form $\mathsf{expl}(t_1, t_2)$ and*

terminals of the form $\mathsf{msw}(t_1, t_2, t_3)$ *where* t_1, t_2, t_3 *are terms, if* Γ *is the smallest set such that the following holds:*

Let β_0 *be the selected literal in some step of a time-abstracted derivation of* Q*. Let* c *be an instance of a clause in* P *with* β_0 *as the LHS atom and* $\beta_1, \ldots, \beta_l$ *be the probabilistic atoms on the RHS. Let* θ *be the computed answer substitution for the RHS of* c *in a time-abstracted derivation. Then*

$$e_{\beta_0}\theta \to e_{\beta_1}\theta, \ldots, e_{\beta_l}\theta \quad \in \Gamma.$$

□

An explanation generator is said to be *ground* if, for every non-terminal symbol $\mathsf{expl}(t_1, t_2)$, t_1 is ground and, for every terminal symbol $\mathsf{msw}(t_1, t_2, t_3)$, t_1 and t_3 are ground.

Example 4.3. *The DCG in Example 4.2 is the explanation generator for the query* $\mathsf{reach}(s_0, H, s_3)$ *over the program given in Figure 4.1(b).*

Proposition 4.1. *Let* P *be a temporally well-formed program,* Q *be a ground query, and* Γ *be the explanation generator for* Q *w.r.t.* P *such that* Γ *is ground. Then, the language of* Γ *corresponds to the set of explanations of* Q*,* i.e.*,* $\mathcal{E}_P(Q) = \{sym(s) \mid s \in \mathcal{L}_\Gamma\}$.

Note that the generator in Example 4.2 can be treated as a stochastic grammar (with the probability of the msws representing the probability of each production), and hence the probability of the query can be computed directly. However, this does not hold in general. For instance, the query $\mathsf{reach}(s_0, H, s_3)$; $\mathsf{reach}(s_0, H, s_4)$ considered in the introduction will result in an explanation generator where the productions are not mutually exclusive. To treat such generators, we define the *factoring* algorithm described below.

4.3.2 Factored Explanation Diagrams

The structure of a Factored Explanation Diagram (FED) closely follows that of a Binary Decision Diagram (BDD). Similar to a BDD, a FED is a labeled directed acyclic graph (DAG) with two distinguished leaf nodes: tt, representing *true*, and ff, representing *false*. While the internal nodes of a BDD are Boolean variables, a FED contains two kinds of internal nodes: one representing terminal symbols of explanations (msws), and the other representing non-terminal symbols of explanations (expls). Thus, each path in a FED can

be viewed as a *production* in a context free grammar. We will ensure, by construction, that distinct paths in a FED are mutually exclusive and that the set of msws used within a path are all mutually independent. Hence, we can view a FED as a *stochastic grammar*, where each production's probability is given by the (product of) probabilities of msws in that production. We use a partial order among nodes, denoted by "<", to construct a FED. The order is used to ensure the mutual exclusion and independence properties stated above.

Definition 4.5 (Factored Explanation Diagram)**.** *A FED is a labeled DAG with four kinds of nodes:*

- tt *and* ff *are terminal nodes;*
- $\mathsf{msw}(r, h)$ *is an* n*-ary node when* r *is a random process with* n *outcomes, and the edges to the* n *children are labeled with the possible outcomes of* r*;*
- $\mathsf{expl}(t, h)$ *is a binary node, and the edges to the children are labeled* 0 *and* 1*.*

In the above, t *is a ground term and* h *is an instance term (for an* expl *node, it represents a range of instances). If there is an edge from node* x_1 *to node* x_2*, then* $x_1 < x_2$*.* □

Note that the multi-valued decision diagrams used in the implementation of PITA [RS10b] are a special case of FEDs with only tt, ff, and $\mathsf{msw}(r, h)$ nodes, where r and h are ground.

We represent non-terminal FEDs by $x?Alts$, where x is the node and $Alts$ is the list of edge-label/child pairs. A FED F whose root is an msw node is written as $\mathsf{msw}(r, h)?[v_1{:}F_1, v_2{:}F_2, \ldots, v_n{:}F_n]$, where $F_1, F_2, \ldots, F_n$ are children FEDs (not all necessarily distinct) and $v_1, v_2, \ldots, v_n$ are possible outcomes of the random process r such that v_i is the label on the edge from F to F_i.

A FED F whose root is an expl node is written as $\mathsf{expl}(t, h)?[0{:}F_0, 1{:}F_1]$, where F_0 and F_1 are the children of F with edge labels 0 and 1, respectively.

We now define the ordering relation "<" among nodes. Let "$\prec$" be a partial order among instances such that $h_1 \prec h_2$ if h_1 represents an earlier time instance than h_2. Let $\sqsubset$ be a total order among instances, along the lines of a lexicographic ordering, such that $h_1 \prec h_2 \Rightarrow h_1 \sqsubset h_2$. If $h_1 \not\prec h_2$ and

$h_2 \not\sqsubset h_1$, then h_1 and h_2 are incomparable, denoted as $h_1 \not\sqsubset h_2$. We assume an arbitrary order $<$ among terms.

Definition 4.6 (Node order)**.** *Let x_1 and x_2 be nodes in a FED. Then $x_1 < x_2$ if it matches one of the following:*

- $\mathsf{msw}(r_1, h_1) < \mathsf{msw}(r_2, h_2)$ *if* $h_1 \sqsubset h_2$ *or* $(r_1 < r_2$ *and* $h_1 = h_2)$*;*
- $\mathsf{msw}(r_1, h_1) < \mathsf{expl}(t_2, h_2)$ *if* $h_1 \prec h_2$ *or* $h_1 \not\sqsubset h_2$*;*
- $\mathsf{expl}(t_1, h_1) < \mathsf{expl}(t_2, h_2)$ *if* $h_1 \sqsubset h_2$ *and* $h_1 \not\sqsubset h_2$. □

A node of the form $\mathsf{msw}(r, h, v)$ denotes a valuation of random process r at instance h. From PRISM semantics, two random variables are independent if they differ in their process or instance. Thus two msw nodes related by "$<$" are independent. A node n of the form $\mathsf{expl}(t, h)$ may represent a set of msws at instance h or later. It follows from the above definition that if m is an msw node and $m < n$, then m is independent of every msw represented by n. Finally, we can order two expl nodes with instances h_1 and h_2 only if they have incomparable instances, since they represent sets of msws at or later than h_1 and h_2, respectively.

Definition 4.7 (Binary Operations on FEDs)**.** *$F_1 \oplus F_2$, where $\oplus \in \{\wedge, \vee\}$, is a FED F derived as follows:*

- *F_1 is* tt*, and $\oplus = \vee$, then $F =$* tt*.*
- *F_1 is* tt*, and $\oplus = \wedge$, then $F = F_2$.*
- *F_1 is* ff*, and $\oplus = \vee$, then $F = F_2$.*
- *F_1 is* ff*, and $\oplus = \wedge$, then $F =$* ff*.*
- *$F_1 = x_1?\left[v_1^1{:}F_1^1, \ldots, v_1^{n_1}{:}F_1^{n_1}\right]$, $F_2 = x_2?\left[v_2^1{:}F_2^1, \ldots, v_2^{n_2}{:}F_2^{n_2}\right]$:*
  1. *$x_1 < x_2$: $F = x_1?\left[v_1^1{:}(F_1^1 \oplus F_2), \ldots, v_1^{n_1}{:}\left(F_1^{n_1} \oplus F_2\right)\right]$;*
  2. *$x_1 = x_2$: $F = x_1?\left[v_1^1{:}(F_1^1 \oplus F_2^1), \ldots, v_1^{n_1}{:}\left(F_1^{n_1} \oplus F_2^{n_1}\right)\right]$;*
  3. *$x_1 > x_2$: $F = x_2?\left[v_2^1{:}(F_1 \oplus F_2^1), \ldots, v_2^{n_2}{:}\left(F_1 \oplus F_2^{n_2}\right)\right]$;*
  4. *$x_1 \not< x_2, x_2 \not< x_1$: $F = \mathsf{merge}(\oplus, F_1, F_2)$.* □

Note that Definition 4.7 is a generalization of the corresponding operations on BDDs. Also, when x_1 and x_2 are both msw nodes, since $<$ defines a total order between them, case 4 will not apply. When the operand nodes cannot be ordered (case 4), then we can try to merge them, but this can potentially cause the FED generation to fail. We show how we may try to resolve the merge cases.

Definition 4.8 (Merge of FEDs)**.** *We can merge FEDs F_1 and F_2 when $F_1 = \mathsf{expl}(t_1, h_1)?[0{:}F_1^0, 1{:}F_1^1]$, $F_2 = \mathsf{expl}(t_2, h_2)?[0{:}F_2^0, 1{:}F_2^1]$, and $h_1 = h_2 = h$.*

Then, $\mathsf{merge}(\oplus, F_1, F_2) = \mathsf{expl}(t_1 \oplus t_2, h)?[0{:}(F_1^0 \oplus F_2^0), 1{:}(F_1^1 \oplus F_2^1)]$. □

When we need to merge, having the FEDs rooted at expl nodes with identical contexts may be achieved via an Explanation Normal Form (ENF).

Definition 4.9 (Explanation Normal Form)**.** *A FED is in ENF if, for some context h:*

- msw *nodes have context h;*
- expl *nodes have a context of the form $h.r.v_i$.* □

Meanwhile, we have in general the following relationships for $i \in \{1, 2\}$: $F_i = F_i^0 \vee (\mathsf{expl}(t_i, h_i) \wedge F_i^1)$, with the invariant that F_i^0 implies F_i^1. Then, when $h_1 = h_2 = h$, our construction corresponds to the following relationship:

$$F_1 \wedge F_2 \stackrel{?}{=} \left(F_1^0 \wedge F_2^0\right) \vee \left(\mathsf{expl}(t_1 \wedge t_2, h) \wedge \left(F_1^1 \wedge F_2^1\right)\right). \tag{4.1}$$

The relationship in (4.1) is not universal, though; *e.g.*, when t_1 holds, but t_2 does not, we just need $F_1^1 \wedge F_2^0$. A sufficient condition for (4.1) is $F_1^0 = F_2^0$ (for $F_1 \vee F_2$, the respective condition is $F_1^1 = F_2^1$). We will show in Section 4.3.3 some particular cases where this condition is met.

We now give a procedure for constructing FEDs from an explanation generator for query Q with respect to program P.

Definition 4.10 (FED Construction)**.** *Given an explanation generator Γ, the FED corresponding to goal G, denoted by $\mathsf{fed}(G)$, is constructed by mutually recursive functions fed and expand defined as follows:*

- $\mathsf{fed}(G)$*:*

$= \mathsf{msw}(r,h)?[v_1{:}F_1,\ldots,v_n{:}F_n]$ *if* $G = \mathsf{msw}(r,h,v)$*, where for all* i*,* $F_i = \mathsf{tt}$ *if* $v_i = v$ *and* $F_i = \mathsf{ff}$ *otherwise;*

$= \mathsf{expand}(G)$ *if* $G = \mathsf{expl}(t,h)$*,* h *is either ground or a variable;*

$= G?[0{:}\mathsf{ff}, 1{:}\mathsf{tt}]$ *otherwise.*

- $\mathsf{expand}(\beta_0) = F$ *where* $\left\{ (\beta_0 \to \beta_1^1,\ldots,\beta_1^{n_1}),\ \ldots,\ \left(\beta_0 \to \beta_k^1,\ldots,\beta_k^{n_k}\right) \right\}$ *is the set of all clauses in* Γ *with* β_0 *on the left hand side, and*

$$F = \bigvee_{i=1}^{k} \bigwedge_{j=1}^{n_k} \mathsf{fed}\left(\beta_i^j\right).$$

- $\mathsf{expand}(\mathsf{expl}(t_1 \oplus t_2, h)) = \mathsf{expand}(\mathsf{expl}(t_1,h)) \oplus \mathsf{expand}(\mathsf{expl}(t_2,h))$.

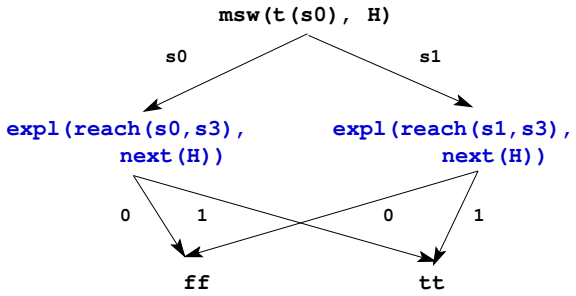
The definition can be understood as follows. The key function is expand, which constructs a FED from a set of clauses in Γ. Function fed (i) defines FEDs for msw nodes, and (ii) controls the expansion by stopping expansion of expl nodes when the instance is partially specified. Expansion of a compound expl node is achieved by expanding the component FEDs and then applying the operation to them. In our implementation, the above definition is turned into a tabled logic program. Furthermore, FEDs are maintained using a dictionary to ensure that they have a DAG structure.

Example 4.4. *Three of the four FEDs for the explanation generator in Example 4.2 are shown in Figure 4.2. The FED for* $\mathsf{expl}(\mathsf{reach}(s_3,s_3),H)$*, not shown in the figure, is* tt.

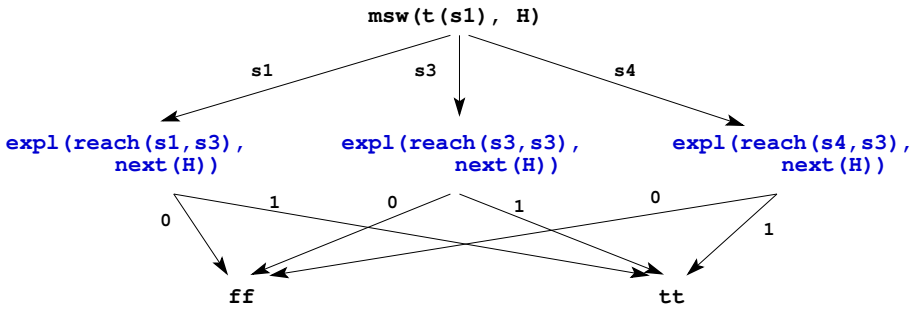
4.3.3 Nondeterminism and Merge

While we focused on systems without linear nondeterminism, in [GRS12] and so far in this chapter, we note that the FED construction is agnostic with respect to the distributions of the probabilistic choices. Thus, if we allow PRISM's [SK97] msws to be backed by linear nondeterminism, essentially following the paradigm of RMDPs [EY15] in that respect, we can still perform the FED construction. In this case, we replace the set_sw predicate giving the distribution with an indicator that the msw atom is nondeterministic.

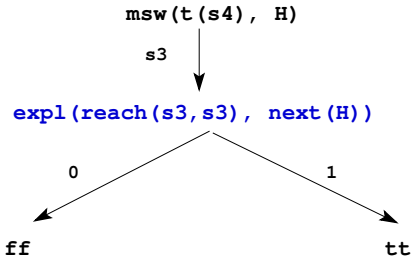
Meanwhile, the FED construction in Definition 4.10 may not be sound, if we perform the merge when the children are in a particular form. It



(a) FED for $\mathsf{expl}(\mathsf{reach}(s_0, s_3), H)$



(b) FED for $\mathsf{expl}(\mathsf{reach}(s_1, s_3), H)$



(c) FED for $\mathsf{expl}(\mathsf{reach}(s_4, s_3), H)$

Figure 4.2: FEDs for Example 4.2

works in linear systems, where $F_0 = \mathsf{ff}$ and $F_1 = \mathsf{tt}$ for any F rooted at an expl node. In probabilistic branching time, though, the results for multi-exit RMDPs [EY15] and Chapter 3 indicate that a FED construction algorithm cannot be compatible with linear nondeterminism and, at the same time, be sound and complete. In the absence of linear nondeterminism, we can change the construction algorithm with the analogue of (2.1), relating the disjunction operation on two FEDs to their conjunction. For this, we define

an additional FED type, a triple.

Definition 4.11 (Triple)**.** *A* triple *is an additional type of terminal node for a FED, representing a FED by referencing three other FEDs, and denoted by* $F = \mathsf{triple}(F_1, F_2, F_3)$*, such that its* value *may be computed from their values.*

In addition to Definition 4.7, $F_1 \oplus F_2$*, where* $F_1 = \mathsf{triple}\,(F_1^1, F_1^2, F_1^3)$*, is a FED* $F = \mathsf{triple}\,(F_1^1 \oplus F_2, F_1^2 \oplus F_2, F_1^3 \oplus F_2)$*.* □

The alternative FED construction replaces the binary children system for expl nodes with the triples (equivalently, we maintain the invariant that all 0 children are ff). This requires the following adjustment to Definition 4.7:

Definition 4.12 (Triple for disjunction)**.** *When FEDs* F_1 *and* F_2 *are rooted at distinct* expl *nodes (*i.e.*, excluding case 2 in Definition 4.7), then* $F_1 \vee F_2 = \mathsf{triple}(F_1, F_2, F_1 \wedge F_2)$*.*

Thus, for systems without linear nondeterminism, we can build FEDs from any set of explanations.

Theorem 4.2 (Sound FED Construction)**.** *For systems where all branching is separable, the FED construction algorithm via Definitions 4.7 and 4.10 is sound and complete.*

For systems without linear nondeterminism, the FED construction algorithm including Definition 4.12 is sound and complete.

Proof. We will assume that the FEDs are in ENF. Then, FED expansion entails exactly one time step, in branching time, and the merge case happens only on two expl nodes with identical contexts.

Also, we start with the input being a set of expls, for which we need to construct FEDs. Any additional FED will be a result of a merge. With the standard algorithm, the number of additional FEDs is bounded double-exponentially, and with triples, exponentially, in the number of initial expls. Thus, completeness is guaranteed.

Potential issues with soundness arise primarily in the merge case. In linear-time systems, the expl nodes have trivial children. With triples, all merges are on conjunction and the 0 children are all ff, so the merge operation in Definition 4.8 is correct. With separable systems, the results of independent contexts are assumed to be themselves independent. This means that, in combining two FEDs, we ultimately just have a tt or ff for a particular context and cannot utilize the details of how it is so. Then, the 0 and 1 children can be combined as with BDDs. □

Thus, in the GPL model checking case, we are able to complete the model checking algorithm by handling conjunctions and disjunctions in an asymmetric way, with triples. For separable XPL model checking, the standard algorithm produces the FEDs yielding the system with the correct result as a solution.

4.3.4 Computing Probabilities from FEDs

Recall that a factored explanation diagram can be viewed as a *stochastic grammar*. Following [EY09], we can generate a set of simultaneous equations from the stochastic grammar and find the probability of the language from the least solution of the equations. The generation of equations from the factored representation of explanations is formalized below, where we assume that a least solution is to be computed.

Definition 4.13 (Temporal Abstraction)**.** *Given a temporal program P, the temporal abstraction of a term t, denoted by $\mathsf{abs}(t)$, is $\overline{\chi}(t)$ if $\pi(t) \in \mathsf{temporal}(P)$ and $\chi(t)$ is not ground, and it is t otherwise. That is, for a term t with a temporal predicate as root, $\mathsf{abs}(t)$ omits its instance argument if that argument is not ground.* □

Definition 4.14 (Distribution)**.** *Let ρ be a random process specified in a temporal program P. The set of values produced by ρ is denoted by $\mathsf{values}_P(\rho)$. The distribution of ρ, denoted by $\mathsf{distr}_P(\rho)$, is a function from the set of all terms over the Herbrand Universe of P to $[0,1]$ such that*

$$\sum_{v \in \mathsf{values}_P(\rho)} \mathsf{distr}_P(\rho)(v) = 1.$$

□

Definition 4.15 (System of Equations for PLP)**.** *Let Γ be an explanation generator, fed be the relation defined in Definition 4.10, V be a countable set of variables, and f be a one-to-one function from terms to V. The system of polynomial equations $E_{(\Gamma,V,f)} = \{\big(f(\mathsf{abs}(G)) = \mathcal{P}(F)\big) \mid \mathsf{fed}(G,F)$ holds$\}$,*

// x_i: $\mathsf{prob}(\mathsf{expl}(\mathsf{reach}(s_i, \_, s_3)))$;	t_{ij}: $\mathsf{prob}(\mathsf{msw}(t(s_i), \_, s_j))$	
$x_0 = t_{00} \cdot x_0 + t_{01} \cdot x_1$	$t_{00} = .5$	$t_{14} = .5$
$x_1 = t_{11} \cdot x_1 + t_{13} \cdot x_3 + t_{14} \cdot x_4$	$t_{01} = .3$	$t_{43} = 1$
$x_3 = 1$	$t_{11} = .4$	
$x_4 = t_{43} \cdot x_3$	$t_{13} = .1$	

Figure 4.3: Set of equations generated from the set of FEDs of Example 4.4

where $\mathcal{P}$ is a function that maps FEDs to polynomials, is defined as follows:

$$\begin{aligned}
\mathcal{P}(\mathsf{ff}) &= 0,\\
\mathcal{P}(\mathsf{tt}) &= 1,\\
\mathcal{P}(\mathsf{msw}(r,h)?[v_1{:}F_1,\ldots,v_n{:}F_n]) &= \begin{cases} \sum_{i=1}^{n} \mathsf{distr}(r)(v_i)\cdot\mathcal{P}(F_i), & r \textit{ is probabilistic}\\ \max_i \mathcal{P}(F_i), & \textit{otherwise}\end{cases}\\
\mathcal{P}(\mathsf{expl}(t,h)?[0{:}F_0,1{:}F_1]) &= f\big(\mathsf{abs}(\mathsf{expl}(t,h))\big)\cdot\mathcal{P}(F_1) +\\
&\quad + \big(1 - f\big(\mathsf{abs}(\mathsf{expl}(t,h))\big)\big)\cdot\mathcal{P}(F_0),\\
\mathcal{P}(\mathsf{triple}(F_1,F_2,F_3)) &= \mathcal{P}(F_1) + \mathcal{P}(F_2) - \mathcal{P}(F_3).
\end{aligned}$$

□

The set of equations for Example 4.4 is shown in Figure 4.3.

Note that subtraction may be present either as a result of triples or non-ff 0 children, but not both.

The implementation of the above definition is such that shared FEDs result in shared variables in the equation system. The correspondence between a temporal program in factored form and the set of polynomial equations permits us to compute the probability of query answers in terms of the least solution to the system of equations.

Theorem 4.3 (Factored Forms and Probability)**.** *Let Γ be an explanation generator for query Q w.r.t. program P. Let V be a set of variables and let f be a one-to-one function from terms to V. Then, X is the probability of a query answer Q evaluated over P, denoted by $\mathsf{prob}(Q, X)$, if X is the value of the variable $f(\mathsf{expl}(\overline{\chi}(Q), \chi(Q)))$ in the least solution of the corresponding set of equations, $E_{(\Gamma,V,f)}$.*

The following properties show that the algorithm for finding probabilities of a query answer is well defined.

Proposition 4.4 (Monotonicity)**.** *If Γ is an explanation generator in factored form, V is a set of variables and f is a one-to-one function as required by Definition 4.15, then the system of equations $E_{(\Gamma,V,f)}$ is monotone in $[0,1]$.*

Monotone systems have the following important property:

Proposition 4.5 (Least Solution [EY09])**.** *Let E be a set of polynomial equations which is monotone in $[0,1]$. Then E has a least solution in $[0,1]$. Furthermore, a least solution can be computed to within an arbitrary approximation bound by an iterative procedure.*

Note that FEDs may not be regular since expl nodes may have other expl nodes as children, and hence the resulting equations may be nonlinear. Proposition 4.5 establishes that the probability of query answers can be effectively computed even when the set of equations is nonlinear.

Example 4.5. *The probability of the language of explanations in Example 4.2 (via the equations in Figure 4.3) is given by the value of x_0 in the least solution, which is* 0.6*.*

4.4 Applications

We now present two model checkers that demonstrate the utility of PIP.

PCTL: The syntax of an illustrative fragment of PCTL is given by:

$$SF ::= \mathsf{prop}(A) \mid \mathsf{neg}(SF) \mid \mathsf{and}(SF_1, SF_2) \mid \mathsf{pr}(PF, \mathsf{gt}, B) \mid \mathsf{pr}(PF, \mathsf{geq}, B),$$
$$PF ::= \mathsf{until}(SF_1, SF_2) \mid \mathsf{next}(SF).$$

Here, A is a proposition and B is a real number in $[0,1]$. The logic partitions formulae into *state* formulae (denoted by SF) and *path* formulae (denoted by PF). State formulae are given a non-probabilistic semantics: a state formula is either true or false at a state. For example, formula $\mathsf{prop}(a)$ is true at state s if proposition a holds at s; a formula $\mathsf{and}(SF_1, SF_2)$ holds at s if both SF_1 and SF_2 hold at s. The formula $\mathsf{pr}(PF, \mathsf{gt}, B)$ holds at a state s if the probability p of the set of all paths on which the path formula PF holds is such that $p > B$ (similarly, $p \geq B$ for geq).

```
% State Formulae
models(S, prop(A)) :-
  holds(S, A).
models(S, neg(A)) :-
  not models(S, A).
models(S, and(SF1,SF2)) :-
  models(S, SF1),
  models(S, SF2).
models(S, pr(PF,gt,B)) :-
  prob(pmodels(S, PF), P),
  P > B.
models(S, pr(PF,geq,B)) :-
  prob(pmodels(S, PF), P),
  P >= B.
```

```
% Path Formulae
pmodels(S, PF) :-
  pmodels(S, PF, _).

:- table pmodels/3.
pmodels(S, until(SF1, SF2), H) :-
  models(S, SF2).
pmodels(S, until(SF1, SF2), H) :-
  models(S, SF1),
  trans(S, H, T),
  pmodels(T,until(SF1,SF2),next(H)).
pmodels(S, next(SF), H) :-
  trans(S, H, T),
  models(T, SF).

temporal(pmodels/3-3).
```

Figure 4.4: Model checker for a fragment of PCTL

The formula $\mathsf{until}(SF_1, SF_2)$ holds on a given path $(s_0, s_1, s_2, \dots)$ if SF_2 holds on state s_k for some $k \geq 0$ and SF_1 holds for all s_i, $0 \leq i < k$. Full PCTL has a *bounded* until operator, which imposes a fixed upper bound on k; we omit its treatment since it has a straightforward non-fixed-point semantics. The *probability* of a path formula PF at a state s is the sum of probabilities of all paths starting at s on which PF holds. This semantics is directly encoded as the probabilistic logic program given in Figure 4.4. In this encoding, trans/3 encodes the transition relation of a Markov chain. Observe the use of an abstract instance argument "_" in the invocation of pmodels/3 from pmodels/2. This ensures that an explanation generator can be effectively computed for any query to pmodels/2.

GPL: GPL is an expressive logic based on the modal μ-calculus for probabilistic systems [CIN05], which we discussed in Chapter 2 – we repeat the basics here. GPL subsumes PCTL and PCTL* in expressiveness. GPL is designed for model checking RPLTSs, which are a generalization of Markov chains. In an RPLTS, a state may have zero or more outgoing transitions, each labeled by a distinct action symbol. Each action has a distribution on destination states.

Syntactically, GPL has *state* and *fuzzy* formulae, where the state formulae are similar to those of PCTL. The fuzzy formulae are, however, significantly more expressive. While we defined the syntax of GPL in Section 2.2, we give it here in equational form:

$$
\begin{aligned}
SF &::= \mathsf{prop}(A) \mid \mathsf{neg}(\mathsf{prop}(A)) \mid \mathsf{and}(SF, SF) \mid \mathsf{or}(SF, SF) \mid \\
&\quad \mid \mathsf{pr}(PF, \mathsf{gt}, B) \mid \mathsf{pr}(PF, \mathsf{lt}, B) \mid \mathsf{pr}(PF, \mathsf{geq}, B) \mid \mathsf{pr}(PF, \mathsf{leq}, B) \\
PF &::= \mathsf{sf}(SF) \mid \mathsf{form}(X) \mid \mathsf{and}(PF, PF) \mid \mathsf{or}(PF, PF) \mid \\
&\quad \mid \mathsf{diam}(A, PF) \mid \mathsf{box}(A, PF) \\
D &::= \mathsf{def}(X, \mathsf{lfp}(PF)) \mid \mathsf{def}(X, \mathsf{gfp}(PF))
\end{aligned}
$$

Formula $\mathsf{diam}(A, PF)$ holds at a state if there is a transition labeled A after which PF holds; $\mathsf{box}(A, PF)$ may also hold if there are no transitions labeled A. In the syntax, X denotes a formula variable defined using LFP and GFP equations in D using lfp and gfp, respectively. Formulae are specified as a set of definitions. GPL admits only alternation-free fixed-point formulae, and hence treating them as a set of recursive definitions suffices to ensure the completeness of the equational representation.

A part of the model checker for GPL that deals with fuzzy formulae is shown in Figure 4.5. Note that fuzzy formulae have probabilistic semantics, and, at the same time, may involve conjunctions or disjunctions of other fuzzy formulae. Thus, for example, when evaluating $\mathsf{models}(s, \mathsf{and}(PF_1, PF_2), H)$, the explanations of $\mathsf{models}(s, PF_1, H)$ and $\mathsf{models}(s, PF_2, H)$ may not be pairwise independent. Thus, recursion-free fuzzy formulae cannot be evaluated in PRISM, but can be evaluated using the BDD-based algorithms of ProbLog and PITA. In contrast, *recursive* fuzzy formulae can be evaluated using PIP. Separable XPL formulae can also be evaluated, directly on PLTSs in which any given transition is either nondeterministic or probabilistic, and otherwise following a basic transformation of the PLTS and the formula.

4.5 Experimental Results

PIP has been implemented using the XSB tabled logic programming system [SW+12]. An explanation generator is constructed by using query evaluation under the well-founded semantics by redefining msws to backtrack through their potential values and to have the *undefined* truth value. This generates a *residual* program in XSB that captures the dependencies between the original goal and the msws (now treated as undefined values). In

%% pmodels(S, PF, H): S in model of fuzzy formula PF at or after instant H

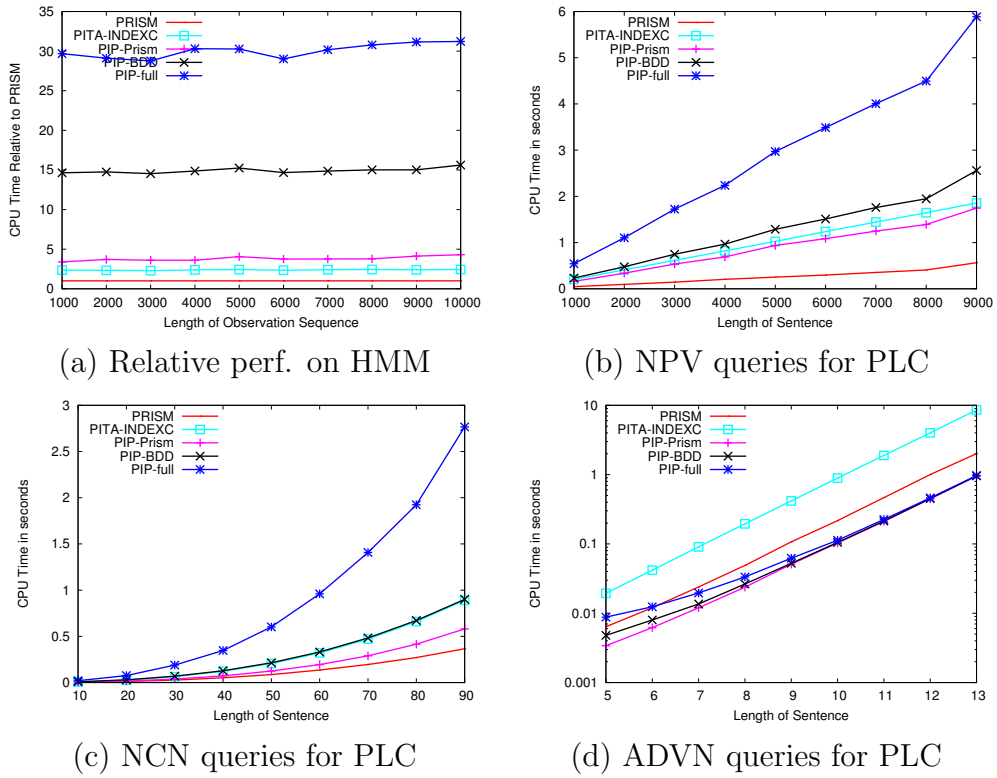
%% smodels(S, SF): S in model of state formula SF

```
pmodels(S, sf(SF), H) :-
  smodels(S, SF).
pmodels(S, and(F1,F2), H) :-
  pmodels(S, F1, H),
  pmodels(S, F2, H).
pmodels(S, or(F1,F2), H) :-
  pmodels(S, F1, H);
  pmodels(S, F2, H).
pmodels(S, diam(A, F), H) :-
  trans(S, A, SW),
  msw(SW, H, T),
  pmodels(T, F, [T,SW|H]).
pmodels(S, box(A, F), H) :-
  findall(SW,trans(S,A,SW),L),
  all_pmodels(L, S, F, H).

pmodels(S, form(X), H) :-
  tabled_pmodels(S,X,H1), H=H1.

all_pmodels([], _, _, _H).
all_pmodels([SW|Rest],S,F,H) :-
  msw(SW, H, T),
  pmodels(T,F,[T,SW|H]),
  all_pmodels(Rest, S, F, H).

:- table tabled_pmodels/3.
tabled_pmodels(S,X,H) :-
  fdef(X, lfp(F)),
  pmodels(S, F, H).
```

Figure 4.5: Fragment of a model checker for fuzzy formulae in GPL

the first partial implementation, called **PIP-Prism**, the probabilities are computed directly from the residual program. Note that such a computation will be correct if PRISM's restrictions are satisfied. In general, however, we materialize the explanation generator. The second partial implementation, called **PIP-BDD**, constructs BDDs from the explanation generator and computes probabilities from the BDD. Note that **PIP-BDD** will be correct when the finiteness restriction holds. The full implementation of PIP, called **PIP-full**, is obtained by constructing a set of FEDs from the explanation generator (Definition 4.10), generating polynomial equations from the set of FEDs (Definition 4.15), and finally finding the least solution to the set of equations. The final equation solver is implemented in C. All other parts of the three implementations, including the BDD and FED structures, are implemented entirely in tabled Prolog.

We present two sets of experimental results, evaluating the performance of PIP on (1) programs satisfying PRISM's restrictions; and (2) a program for model checking PCTL formulae. The results were collected on a machine running Mac OS X 10.6.8, with a 4-core 2.5 GHz Intel Core i5 processor and 4 GB of memory. For the first set of results, we compare PIP with PRISM



(a) Relative perf. on HMM

(b) NPV queries for PLC

(c) NCN queries for PLC

(d) ADVN queries for PLC

Figure 4.6: Performance of PIP on PRISM Programs

v2.0.3 and PITA in XSB v3.3.6. The model checking results were compared with that of PRISM Model Checker v4.0.3.

Performance on PRISM Programs: Note that all three implementations (PIP-Prism, PIP-BDD and PIP-full) may be used to evaluate PRISM programs.

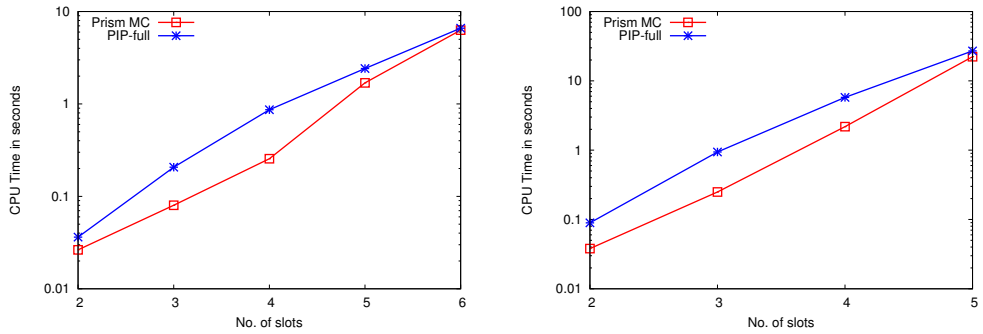
Hidden Markov Model (HMM): We used the simple 2-state gene sequence HMM from [CG09] (also used in [RS11]) for our evaluation. We measured the CPU time taken by the versions of PIP, PRISM and PITA-INDEXC [RS11] (a version of PITA that uses not BDDs, but PRISM's assumptions) to evaluate the probability of a given observation sequence, for varying sequence lengths. The observation sequence itself was embedded as a set of facts (instead of an argument list). This makes table accesses fast even when shallow indices are used. The number of nodes in the explanation

graph (*e.g.*, FED) is linear in the size of the argument list. The performance of the three PIP versions and PITA-INDEXC, relative to PRISM, is shown in Figure 4.6(a). CPU times are normalized using PRISM's time as the baseline. Observe that the PIP-Prism and PITA-INDEXC perform similarly: about 3.5 to 4 times slower than PRISM. Construction of BDDs (done in PIP-BDD, but not in PIP-Prism) increases this overhead by a factor of 4. Construction of full-fledged FEDs, generating polynomial equations and solving them (done only in PIP-full) further doubles the overhead. We find that the equation-solving time is generally negligible.

Probabilistic Left Corner (PLC) Parsing: This example was adapted from PRISM's example suite, parameterizing the length of the input sequence to be parsed. We measured the CPU time taken by the three versions of PIP, PRISM, and PITA. The performance on three queries (NPV, NCN, and $ADVN$, each encoding a different class of strings) is shown in Figure 4.6(b)–(d). As in the HMM example, the sequences are represented as facts instead of lists. The number of nodes in the explanation graph is linear in string length for NPV queries; and quadratic for NCN and $ADVN$ queries. The processing time needed for generating explanations is linear for NPV queries; quadratic for NCN queries; and exponential in $ADVN$ queries. Note that the y-axis is logarithmic in Figure 4.6(d), and linear in the rest. The performance on NPV is similar to that on HMM, with one exception: PIP-Prism marginally outperforms PITA-INDEXC. The differences are more noticeable on NCN and become significant on $ADVN$, indicating the relative efficiency of computing explanations in PIP. The performance of PIP relative to PRISM shows smaller overheads on NPV and NCN (compared to HMM) and exhibits a reversal on $ADVN$. The differences between PIP-Prism, PIP-BDD and PIP-full narrow in NPV and become negligible in $ADVN$ due to the small size of explanations.

The PLC experiments show that the overhead of BDD or FED construction is not necessarily constant across different queries or within a class of similar queries on different scales, but may depend on factors such as how compact the explanation generator is with respect to the set of explanations.

Performance of the PCTL Model Checker: We evaluated the performance of PIP-full for supporting a PCTL model checker (encoded as shown in Figure 4.4). We compared the performance of PIP-based model checker with that of the widely-used PRISM model checker [KNP11]. We



(a) Number of processes $N = 5$ (b) Number of processes $N = 6$

Figure 4.7: Performance of PCTL model checking using PIP and the PRISM model checker for Synchronous Leader Election protocol

show the performance of PIP and the PRISM model checker on the Synchronous Leader Election Protocol [IR81] for computing the probability that eventually a leader will be elected. Figure 4.7 shows the CPU time used to compute the probabilities of this property on systems with different numbers of processes (N) and number of slots used by the protocol (*slots*). Observe that our high-level implementation of a model checker based on PIP performs within a factor of 3 of the PRISM model checker (note: the y-axis on these graphs is logarithmic). Moreover, the two model checkers show similar performance trends with increasing problem instances. However, it should be noted that the PRISM model checker uses a BDD-based representation of reachable states, which can, in principle, scale better to large state spaces compared to the explicit state representation used in our PIP-based model checker.

4.6 Conclusions

In this chapter, we have shown that in order to formulate the problem of probabilistic model checking in probabilistic logic programming, one needs an inference algorithm that functions correctly even when finiteness, mutual-exclusion, and independence assumptions are simultaneously violated. We have discussed such an inference algorithm, PIP [GRS12], implemented it in XSB Prolog, and demonstrated its practical utility as the basis for encoding model checkers for a rich class of probabilistic models and temporal logics.

Chapter 5

Partial pLμ Model Checking

In Chapter 3, we saw that we could specify a whole system as a PLTS and then verify a property in a probabilistic μ-calculus, such as GPL or XPL [CIN05, GR16]. However, systems are frequently made up of smaller components; instead of expressing the whole system, we would like to specify the components under a compositional framework. The PRISM Model Checker [KNP11] achieves this in the probabilistic domain, for Markov chains and MDPs, via reactive modules [AH99]. However, we seek to allow quotienting, as defined in [And95], but in the probabilistic domain, and moreover attempt to define, in particular cases, a form of parameterized partial model checking a la [BR06]. Additionally, there are some similarities to our translation of RMDPs [EY15] in Section 3.4.2, which can be seen as sequential composition, as compared to the parallel composition we will describe in this chapter.

The suitability of PLTSs, expressed via a process algebra, and pLμ for compositional model checking has recently been explored by Mio and Simpson [MS13b]. As we depart from GPL and XPL in this chapter, we first discuss pLμ in Section 5.1. Then, we concentrate on the partial model checking aspect. We aim for our framework to facilitate quotienting, and our decisions are designed to require a minimum of extensions. We retain a visible action on every step and preserve action labels on synchronization. In Section 5.2, we define a process algebra for PLTSs, which is loosely based on CCS [Mil89, JYL01]. We describe how quotienting works with our system in Section 5.3. Additionally, in some advanced cases, we want to be able to vary the response based on the number of actions synchronized in the step, *e.g.*, to detect collisions; we analyze the effect of supporting this in Section 5.4. Finally, we discuss a pair of case studies in Section 5.5, and conclude in Section 5.6.

5.1 pLμ

The most relevant aspect of pLμ, for this thesis, is that, with the independent product extension (pL$\mu^{\odot}$) [Mio11], it also supports probabilistic branching time. However, it arises as a result of a logical operator, which enforces independence, and is not viewed as inherent to the probabilistic system. As PLTSs are not viewed as probabilistic branching time systems by pLμ, the combination of a pLμ formula with a PLTS can lead to such a system instead. This is called a Markov Branching Play (MBP). The analogue to d-trees (Section 2.1), where all the probabilistic choices of an MBP have been resolved, is called a branching play.

The key operator present in pLμ and absent in XPL is the conjunction as minimum; XPL does allow for linear nondeterminism, but can refer to it only via the schedulers. Meanwhile, the independent product extension of pLμ is quite straightforward to model with XPL and a PLTS interpreted as a probabilistic branching time system. Indeed, the model checking of separable XPL involves grouping by action, where all remaining top-level conjunctions and disjunctions acquire the semantics of independent product and coproduct, respectively.

Another extension of pLμ involves the truncated sum operator. We mention it since it allows for the modeling of multi-exit RMC termination: the truncated sum can represent a disjunction of two mutually exclusive properties, when it is under an LFP not affected by a linear-nondeterministic choice.

Finally, for partial model checking, the extension of pLμ requires, at the very least, convex combinations, with the logic denoted as pL$\mu \cup \{+_\lambda\}$ [Mio12].

5.1.1 pLμ Syntax

With $X \in \mathsf{Var}$, $a \in \mathsf{Act}$, and $0 \leq \lambda \leq 1$, the syntax of extended pL$\mu \cup \{+_\lambda\}$ is:

$$\psi ::= X \mid \psi \wedge \psi \mid \phi \vee \phi \mid \langle a \rangle \psi \mid [a]\psi \mid \mu X.\psi \mid \nu X.\psi \mid \psi +_\lambda \psi.$$

The syntax of pLμ is similar to GPL/XPL fuzzy formulae, although the propositional connectives and the modal operators represent a linear nondeterministic choice (*i.e.*, a choice made before the future probabilistic choices are resolved); additionally, the $+_\lambda$ operator is a weighted sum, which will be introduced into a formula only as a result of quotienting out a $\mathsf{pchoice}$. We may also write $\sum_i \lambda_i \psi_i = \lambda_1 \cdot \psi_1 + \cdots + \lambda_n \cdot \psi_n$ for a larger distribution.

Table 5.1: pLμ semantics

$$
\begin{aligned}
[\![X]\!]_e(s) &= e(X)(s),\\
[\![\psi_1 \vee \psi_2]\!]_e(s) &= [\![\psi_1]\!]_e(s) \sqcup [\![\psi_2]\!]_e(s),\\
[\![\psi_1 \wedge \psi_2]\!]_e(s) &= [\![\psi_1]\!]_e(s) \sqcap [\![\psi_2]\!]_e(s),\\
[\![\psi_1 +_\lambda \psi_2]\!]_e(s) &= [\![\psi_1]\!]_e(s) +_\lambda [\![\psi_2]\!]_e(s),\\
[\![\langle a\rangle\psi]\!]_e(s) &= \bigsqcup_{c\in\mathbb{N}} \left(\sum_{s':(s,a,s')\in\delta} P(s,a,s',c)\cdot [\![\psi]\!]_e(s') \right),\\
[\![[a]\psi]\!]_e(s) &= \bigsqcap_{c\in\mathbb{N}} \left(\sum_{s':(s,a,s')\in\delta} P(s,a,s',c)\cdot [\![\psi]\!]_e(s') \right),\\
[\![\mu X.\psi]\!]_e(s) &= \mathrm{lfp}\left(\lambda f.[\![\psi]\!]_{e[f/X]}\right)(s),\\
[\![\nu X.\psi]\!]_e(s) &= \mathrm{gfp}\left(\lambda f.[\![\psi]\!]_{e[f/X]}\right)(s).
\end{aligned}
$$

5.1.2 pLμ Semantics

The semantics of pLμ with respect to a fixed PLTS $L = (S, \delta, P, I)$ is similar to that of the fuzzy formulae in Table 2.1 [CIN05, GR16], but is a value $p \in [0,1]$, rather than a set of d-trees. Additionally, the environment here is a function $e : \mathsf{Var} \to [0,1]^S$. The semantics of a pL$\mu$ formula is in Table 5.1. Note that disjunction, conjunction, and the modal operators use $\sqcup$ and $\sqcap$ here, which, for finitely branching systems, correspond to max and min, respectively. Additionally, we explain "lfp" and "gfp", as GPL and XPL allowed for simpler fixed points, due to the alternation-free restriction. For a complete lattice $(L, \leq)$ (which makes $\sqcup$ and $\sqcap$ well-defined for any subset) and monotone function $f : L \to L$, we have the equalities [Mio11, Theorem 2.3]:

$$\mathrm{lfp}(f) = \bigsqcup_\alpha f^\alpha, \qquad f^\alpha = \bigsqcup_{\beta<\alpha} f(f^\beta), \tag{5.1}$$

$$\mathrm{gfp}(f) = \bigsqcap_\alpha f_\alpha, \qquad f_\alpha = \bigsqcap_{\beta<\alpha} f(f_\beta). \tag{5.2}$$

5.1.3 Markov Branching Plays

Branching plays and MBPs are defined for *stochastic tree games* [Mio11, Chapter 4]. Stochastic tree games are somewhat similar to RSSGs and BSSGs [EY15, ESY15], which we mentioned in Section 3.4. There is a game arena $\mathcal{A} = \langle (S, E), (S_1, S_2, S_N, B), \pi \rangle$, with the set of states S partitioned into the maximizing, minimizing, probabilistic, and branching states, E being the transition relation, and $\pi : S_N \to \mathcal{D}(S)$ providing the distribution for the probabilistic states.

Trees in the game arena are defined in the standard way, and there is the intuitive notion of a node being uniquely or fully branching. A branching play is fully branching only on branching nodes and uniquely branching on the others. An MBP is fully branching on both branching and probabilistic nodes [Mio11, Definitions 4.2-4.6].

5.1.4 Partial Model Checking and pLμ

We view as useful two alternative viewpoints on compositional model checking. We may consider a process as representing a PLTS, with some additional information used for composing with other processes. When composition is complete, we could then extract a PLTS, and the properties to be verified would also be written in pLμ. This is similar to what the PRISM Model Checker [KNP11] does, with Markov chains and MDPs as its model.

Alternatively, we may consider the effect of quotienting directly on PLTSs and pLμ. This broadens the interpretation of the model and the logic, and the PLTSs assume a specialized form, but without requiring any additional extensions. This will generally be our paradigm, with the former alternative serving to help with intuition.

Example 5.1 (Running Example)**.** *There is some number of identical processes, each of which probabilistically chooses to be active on one of two channels or to remain idle. We require that at least one process be active, but not both channels.*

Thus, in Example 5.1, if we have only a few processes, failure is likely to result when all the processes are idle; meanwhile, if there are a lot of processes, a typical failure will be from choosing both channels. Each process could be modeled by a 4-state PLTS, and the composed system's size would be exponential in the number of processes; meanwhile, we will show that we

can quotient out the processes in Example 5.1 while the formula remains a convex combination bounded by a constant number of terms.

Example 5.2. *The formula for Example 5.1, that we want at least one active process, but not both channels active, may be written as follows:*

$$\psi = \langle p \rangle \big((\langle a \rangle \mathsf{tt} \wedge [b]\mathsf{ff}) \vee ([a]\mathsf{ff} \wedge \langle b \rangle \mathsf{tt}) \big) \tag{5.3}$$

5.2 Probabilistic Model

While our underlying model will be a PLTS, as in Definition 3.1, we will generally deal with a process algebra rather than composing PLTSs directly.

5.2.1 Process Algebra

We now present a process algebra over PLTSs. Here, we continue following [BR06], which used CCS [Mil89]. CCS has also been extended probabilistically by [JYL01].

Process Syntax

We define a simplified version of a probabilistic process algebra as follows:

Definition 5.1 (Process Algebra)**.** *A process is built by starting with S in the grammar below.*

$$\begin{aligned} S &::= \mathsf{0} \mid X \mid N \mid S +_\lambda S \mid (S|S) \\ P &::= P + P \mid S \\ E &::= A.P \mid A^0.P \\ N &::= N + N \mid E \\ D &::= X \stackrel{\text{def}}{=} S \end{aligned}$$

□

We have several dynamic operators:

- $a.P$: a simple prefix operator, pref, this means the process proceeds on action a.
- $a^0.P$: a special prefix operator, read, which is not active in itself, but can be composed with a pref to become active.

- $N_1 + N_2, P_1 + P_2$: a nondeterministic choice (nchoice), which can generalize to a choice between a set of prefix processes, with the choices being between actions and within an action, respectively.

- $S_1 +_\lambda S_2$: a probabilistic choice (pchoice), selecting S_1 with probability λ and S_2 with $(1 - \lambda)$, which can generalize to a distribution.

We also have the static operator $S_1|S_2$ (compose), the idle 0 process, and the ability to refer to processes recursively by name (ranging over X, possibly indexed, where we have $X \stackrel{\text{def}}{=} S$).

Process Semantics

To follow PLTSs, action prefixes precede probabilistic choices, so that a process corresponds to a PLTS state at the (external) nchoice operator. Additionally, while we view a and a^0 as distinct actions, they can synchronize, and (until Section 5.4) both cannot be present at the same state; we will allow a to possibly represent either a or a^0 where ambiguity is not a concern. The prefix, (internal) nchoice, and pchoice operators correspond to the transitions. When performing composition, we match similar operations: an nchoice with an nchoice, a prefix process with a prefix (on the same action), and a pchoice with a pchoice. We discuss these in order:

External choices

Let $\mathsf{action}(N)$ be the set of actions in the prefixes of an nchoice process N (cf. $\mathsf{action}(\cdot)$ in Section 2.3 and Definition 3.8 in Section 3.3). That is, $\mathsf{action}(a.P) = \mathsf{action}(a^0.P) = \{a\}$, and $\mathsf{action}(N_1 + N_2) = \mathsf{action}(N_1) \cup \mathsf{action}(N_2)$, where we also require $\mathsf{action}(N_1) \cap \mathsf{action}(N_2) = \varnothing$. The composition depends on whether one or both of the processes have a particular action:

- Let $N_1 = a.P + N_1'$ and $a \notin \mathsf{action}(N_2)$. Then, $N_1|N_2 = a.(P|N_2) + N_1'|N_2$.
- Let $N_1 = E_1 + N_1'$ and $N_2 = E_2 + N_2'$, where $\mathsf{action}(E_1) = \mathsf{action}(E_2)$. Then, $N_1|N_2 = E_1|E_2 + N_1'|N_2'$.

Prefixes

A pref process corresponds to an a transition. A read process corresponds to an a^0 transition. These can be composed as follows:

- $a.P|a.Q$ yields $a.(P|Q)$;
- $a.P|a^0.Q$ yields $a.(P|Q)$;
- $a^0.P|a^0.Q$ yields $a^0.(P|Q)$.

Internal and probabilistic choices
: $(P_1+P_2)|P_3$ and $(S_1+_\lambda S_2)|S_3$ are treated similarly, yielding $P_1|P_3+P_2|P_3$ and $S_1|S_3 +_\lambda S_2|S_3$, respectively. In certain cases, symmetry reduction [CTV06] may be possible, though we do not cover it in this thesis.

An important aspect of a PLTS is that when we ask it to perform an action that is absent at its current state, it terminates. Given our composition operator, however, we have something of a third option, where an action is semi-present at a state. With a read operator, we may even make a transition at a component, but the action is nonetheless not yet fully present, pending a later composition with a pref operator. Hence, we shift the focus from absence of an action to a stronger requirement for presence, via the pref and read operators.

Example 5.3. *The processes in Example 5.1, which can choose one of two channels (each with* 0.1 *probability) or remain idle (with* 0.8 *probability), can be described with the following expression:*

$$P \stackrel{\text{def}}{=} p.(P +_{0.8} (a.0 +_{0.5} b.0)) \tag{5.4}$$

5.3 Quotienting

Quotienting [And95] is at the heart of partial model checking. The essence of quotienting is *transforming* a property that needs to be satisfied based on the contribution of the component quotiented out [BR06]. Interestingly, an obligation that appears to be satisfied may be undone by further quotienting.

5.3.1 Probability Function

It is useful to define the semantics of pLμ directly in terms of processes: $[\![(\psi, P)]\!] = [\![(\psi, s)]\!]_\perp$, as a process P corresponds to a state s of a PLTS L. We may also use an equational syntax for pLμ's fixed points, where we write $X =_\sigma \psi$, with $\sigma \in \{\mu, \nu\}$.

The transitions (s, a, s') correspond to $a.P$, where the internal (P_a) and probabilistic (P_a^i) choices lead to some s'. Then, we can write the semantics as follows:

$$[\![(\langle a\rangle\psi, P)]\!] = \max_{P_a}\left(\sum_i p_a^i \cdot [\![(\psi, P_a^i)]\!]\right), \tag{5.5}$$

$$[\![([a]\psi, P)]\!] = \min_{P_a}\left(\sum_i p_a^i \cdot [\![(\psi, P_a^i)]\!]\right). \tag{5.6}$$

We define an additional notation for a case we call the ordered action interpretation. We label this as $[\alpha]_u$ and $\langle\alpha\rangle_u$, where α is a set of actions, of which we want to select at most one, while otherwise adhering closely to the expected meaning of *box* and *diamond*. This notation assumes a total order on the available actions (*i.e.*, when multiple actions may be active in the same state, the order determines which one has priority), and, *e.g.*, if a is the greatest available action in α and $\alpha' = \alpha \setminus \{a\}$, we have:

$$[\alpha]_u\psi = [a]\psi \wedge \big(\langle a\rangle\mathsf{tt} \vee [\alpha']_u\psi\big),$$
$$\langle\alpha\rangle_u\psi = \langle a\rangle\psi \vee \big([a]\mathsf{ff} \wedge \langle\alpha'\rangle_u\psi\big).$$

5.3.2 Quotienting Rules

With pLμ, quotienting is a function from a pLμ formula and a process to a pLμ formula. Convex combinations are introduced as a result of quotienting.

The quotienting rules can be seen in Table 5.2.

Next, we describe the rules.

Rules 1-3 demonstrate the cases where the process does not change the formula, either because the formula is already tt or ff, or because the process is $\mathsf{0}$, the identity of the composition operator.

Rules 4-7 handle conjunction, disjunction, and convex combinations, simply by treating them separately and combining the results.

Rules 8-10 deal with process and formula names, as well as composition. Note that, in Rule 9, we are essentially designating a new formula to be defined. While we could compose the processes and then quotient on the composed process in Rule 10, we want to be able to quotient them separately; this is essentially our whole goal in defining quotienting in the first place.

Rules 11-18 deal with the modal operators. Rules 11-12 apply the modal operator once the internal choice has been made and just the probabilistic

Table 5.2: Quotienting rules

1.	$\Pi(P)(\mathsf{tt})$	$= \mathsf{tt}$
2.	$\Pi(P)(\mathsf{ff})$	$= \mathsf{ff}$
3.	$\Pi(\mathsf{0})(\psi)$	$= \psi$
4.	$\Pi(P)(\psi_1 \wedge \psi_2)$	$= \Pi(P)(\psi_1) \wedge \Pi(P)(\psi_2)$
5.	$\Pi(P)(\psi_1 \vee \psi_2)$	$= \Pi(P)(\psi_1) \vee \Pi(P)(\psi_2)$
6.	$\Pi(P)(\psi_1 +_\lambda \psi_2)$	$= \Pi(P)(\psi_1) +_\lambda \Pi(P)(\psi_2)$
7.	$\Pi(P_1 +_\lambda P_2)(\psi)$	$= \Pi(P_1)(\psi) +_\lambda \Pi(P_2)(\psi)$
8.	$\Pi(X)(\psi)$	$= \Pi(P)(\psi)$ if $X \stackrel{\text{def}}{=} P$
9.	$\Pi(P)(X_{Ps})$	$= X_{P\|Ps}$
10.	$\Pi(P_1\|P_2)(\psi)$	$= \Pi(P_1)(\Pi(P_2)(\psi))$
11.	$\Pi(a^c.S)(\langle a^n\rangle\psi)$	$= \langle a^{n-c}\rangle\Pi(S)(\psi)$
12.	$\Pi(a^c.S)([a^n]\psi)$	$= [a^{n-c}]\Pi(S)(\psi)$
13.	$\Pi(a.(P_1 + P_2))(\langle a\rangle\psi)$	$= \Pi(a.P_1)(\langle a\rangle\psi) \vee \Pi(a.P_2)(\langle a\rangle\psi)$
14.	$\Pi(a.(P_1 + P_2))([a]\psi)$	$= \Pi(a.P_1)([a]\psi) \wedge \Pi(a.P_2)([a]\psi)$
15.	$\Pi(N)(\langle a\rangle\psi)$	$= \langle a\rangle\Pi(N)(\psi)$
16.	$\Pi(N)([a]\psi)$	$= [a]\Pi(N)(\psi)$
17.	$\Pi(a.P + N)(\langle a\rangle\psi)$	$= \Pi(a.P)(\langle a\rangle\psi)$
18.	$\Pi(a.P + N)([a]\psi)$	$= \Pi(a.P)([a]\psi)$ $(a \notin \mathsf{action}(N))$

choice remains. This is the only place where the annotation on the action can change: $n - 1$ is assumed to be 0, and the a^0 appearing in modal operators represents the set $\{a, a^0\}$.

Rules 13-14 handle the internal choice for an action, which is a disjunction for *diamond* and a conjunction for *box*. Rules 15-16 handle the case when the action is absent in the process, in which case it remains idle. Rules 17-18 handle the external choice when the action is present, by choosing the process containing the action.

The fixed points, which are represented in equational form (via $X =_\sigma \psi$), can lead to additional equations when quotienting, as the same formula may need to quotient out different processes. This is straightforward to generate with an algorithm.

It is also interesting that, due to the presence of convex combinations, a formula has its own probability of being true even without any process. We can denote this as $[\![(\psi, 0)]\!]$. Note that the probability may rise or fall after quotienting out another process, depending on how it affects the action annotations.

In order to identify more formulae as equivalent by simple equality checks, we also define some basic refining rules.

Definition 5.2 (Refining rules)**.** *We apply the following rules to a formula until none of them can be applied, to get a refined formula.*

- $\langle\cdot\rangle\mathsf{ff} = \mathsf{ff}$.
- $[\cdot]\mathsf{tt} = \mathsf{tt}$.
- $\langle a^0\rangle\mathsf{tt} = \mathsf{tt}$.
- $[a^0]\mathsf{ff} = \mathsf{ff}$.
- $\mathsf{ff} \wedge \psi = \mathsf{ff}$.
- $\mathsf{tt} \wedge \psi = \psi$.
- $\mathsf{ff} \vee \psi = \psi$.
- $\mathsf{tt} \vee \psi = \mathsf{tt}$. □

Further refining is possible, as well [BR06].

Example 5.4. *If we quotient out process P (5.4) from formula ψ (5.3) once, with $\psi' = (\langle a\rangle\mathsf{tt} \wedge [b]\mathsf{ff}) \vee ([a]\mathsf{ff} \wedge \langle b\rangle\mathsf{tt})$ we get the following result:*

$$\langle p^0\rangle\psi' +_{0.8} (\langle p^0\rangle[b]\mathsf{ff} +_{0.5} \langle p^0\rangle[a]\mathsf{ff}). \tag{5.7}$$

If we quotient out another process P from the resulting formula, we get:

$$0.64 \cdot \langle p^0\rangle\psi' + 0.02 \cdot \mathsf{ff} + 0.34 \cdot (\langle p^0\rangle[b]\mathsf{ff} +_{0.5} \langle p^0\rangle[a]\mathsf{ff}). \tag{5.8}$$

Except for the values on the convex combinations, we have reached a fixed point with (5.8). Further quotienting of P from the resulting formulae only affects the λ values, for which, letting p_n^1 correspond to the weight of the first subformula, p_n^2 for the second, and p_n^3 the third and the fourth, we have $p_n^1 + p_n^2 + 2 \cdot p_n^3 = 1$ and the following recurrence relation and closed form:

$$\begin{aligned}
&p_0 = (1, 0, 0);\\
&p_n^1 = 0.8 \cdot p_{n-1}^1 = 0.8^n;\\
&p_n^2 = p_{n-1}^2 + 0.2 \cdot p_{n-1}^3 = 1 + 0.8^n - 2 \cdot 0.9^n;\\
&p_n^3 = 0.9 \cdot p_{n-1}^3 + 0.1 \cdot p_{n-1}^1 = 0.9^n - 0.8^n.
\end{aligned}$$

Our quotienting implementation can compute the probabilities numerically, and it may be plausible to derive the recurrence relation in a mostly automatic way, as well. We solved manually for the closed form, although both p_n^1 and p_n^3 are quite intuitive here, and p_n^2 is just $1 - (p_n^1 + 2 \cdot p_n^3)$.

Theorem 5.1 (Quotienting is sound)**.** *For all P, Q, and ψ, $[\![(\psi, Q|P)]\!] = [\![(\Pi(P)(\psi), Q)]\!]$.*

Proof. Proceeds by induction on the size of the formula and the process expression.

- Rules 1-2: A standard process P cannot affect tt or ff. The probability will be 1 for tt and 0 for ff, regardless of the process.
- Rule 3 ($P = \mathsf{0}$): We have $P|\mathsf{0} = P$ for any process P, so:

$$\begin{aligned}
[\![(\psi, (Q|\mathsf{0}))]\!] &= [\![(\psi, Q)]\!]\\
&= [\![(\Pi(\mathsf{0})(\psi), Q)]\!]
\end{aligned}$$

- Rule 4 ($\psi = \psi_1 \wedge \psi_2$): We induct on formula size.

$$\begin{aligned}
[\![(\psi_1 \wedge \psi_2, Q|P)]\!] &= [\![(\psi_1, Q|P)]\!] \sqcap [\![(\psi_2, Q|P)]\!]\\
&= [\![(\Pi(P)(\psi_1), Q)]\!] \sqcap [\![(\Pi(P)(\psi_2), Q)]\!]\\
&= [\![(\Pi(P)(\psi_1) \wedge \Pi(P)(\psi_2), Q)]\!]\\
&= [\![(\Pi(P)(\psi_1 \wedge \psi_2), Q)]\!]
\end{aligned}$$

  The proofs for Rules 5-7 are similar.

- Rule 8 ($P = X$): Here, we just substitute in the definitions for the process, and we can induct on the size of the process expression (given $X \stackrel{\text{def}}{=} P$).

$$\begin{aligned}[[(\psi, Q|X)]] &= [[(\psi, Q|P)]] \\ &= [[(\Pi(P)(\psi), Q)]] \\ &= [[(\Pi(X)(\psi), Q)]]\end{aligned}$$

- Rule 9 ($\psi = X_{Ps}$): The rule itself here is just substitution. The equation for $X_{P|Ps}$ is created with the equation-generating algorithm.

- Rule 10 ($P = P_1|P_2$): This is the composition of three processes. We use the associativity of composition as a binary operator and induct on the size of the process expression.

$$\begin{aligned}[[(\psi, Q|(P_1|P_2))]] &= [[(\psi, (Q|P_1)|P_2)]] \\ &= [[(\Pi(P_2)(\psi), Q|P_1)]] \\ &= [[(\Pi(P_1)(\Pi(P_2)(\psi)), Q)]] \\ &= [[(\Pi(P_1|P_2)(\psi), Q)]]\end{aligned}$$

- Rule 11 ($P = a^c.S$, $\psi = \langle a^n \rangle \psi'$): We assume $Q = a^q.Q_a + Q'$, where $Q_a = \sum_i S_i$ (if $a \notin \mathsf{action}(Q)$, then $q = 0$ and $Q_a = Q$). The case $c + q < n$ is trivial, so we assume $c + q \geq n$ and omit the action annotations.

$$\begin{aligned}[[(\langle a \rangle \psi, Q|a.S)]] &= [[(\langle a \rangle \psi, (a.Q_a + Q')|a.S)]] \\ &= [[(\langle a \rangle \psi, a.(Q_a|S))]] \\ &= [[(\langle a \rangle \psi, a.(\sum_i S_i|S))]] \\ &= \bigsqcup_i [[(\langle a \rangle \psi, a.(S_i|S))]] \\ &= \bigsqcup_i [[(\psi, S_i|S)]] \\ &= \bigsqcup_i [[(\Pi(S)(\psi), S_i)]] \\ &= \bigsqcup_i [[(\langle a \rangle \Pi(S)(\psi), a.S_i)]] \\ &= [[(\langle a \rangle \Pi(S)(\psi), a.Q_a)]] \\ &= [[(\Pi(a.S)(\langle a \rangle \psi), Q)]]\end{aligned}$$

  The proof for the *box* operator in Rule 12 is similar.

- Rule 13 ($P = a.(P_1 + P_2)$, $\psi = \langle a \rangle \psi'$): We assume Q as above and $P_2 = S$, and induct on the size of the process expression.

$$\begin{aligned}
[\![(\langle a \rangle \psi, Q | a.(P + S))]\!] &= [\![(\langle a \rangle \psi, (a.Q_a + Q') | a.(P + S))]\!] \\
&= [\![(\langle a \rangle \psi, a.(Q_a | (P + S)))]\!] \\
&= [\![(\langle a \rangle \psi, a.(Q_a | P + Q_a | S))]\!] \\
&= [\![(\langle a \rangle \psi, a.(Q_a | P))]\!] \sqcup [\![(\langle a \rangle \psi, a.(Q_a | S))]\!] \\
&= [\![(\langle a \rangle \psi, a.Q_a | a.P)]\!] \sqcup [\![(\langle a \rangle \psi, a.Q_a | a.S)]\!] \\
&= [\![(\langle a \rangle \psi, Q | a.P)]\!] \sqcup [\![(\langle a \rangle \psi, Q | a.S)]\!] \\
&= [\![(\Pi(a.P)(\langle a \rangle \psi), Q)]\!] \sqcup [\![(\Pi(a.S)(\langle a \rangle \psi), Q)]\!] \\
&= [\![(\Pi(a.(P + S))(\langle a \rangle \psi), Q)]\!]
\end{aligned}$$

  The proof for the *box* operator in Rule 14 is similar.

- Rule 15 ($P = N$ idle on a, $\psi = \langle a \rangle \psi'$): We can utilize the definition of idling to reduce this to the proof for Rule 11 by letting $P = a^0.N$, and likewise for Rule 16.

- Rule 17 ($P' = a.P + N$, $\psi = \langle a \rangle \psi$, $a \in \mathsf{action}(N_1)$): This follows from our process and composition definitions (likewise for Rule 18), as we compose on a and other actions become irrelevant. We assume Q as above and induct on the size of the process expression.

$$\begin{aligned}
[\![(\langle a \rangle \psi, Q | (a.P + N))]\!] &= [\![(\langle a \rangle \psi, (a.Q_a + Q') | (a.P + N))]\!] \\
&= [\![(\langle a \rangle \psi, a.(Q_a | P) + Q' | N)]\!] \\
&= [\![(\langle a \rangle \psi, a.(Q_a | P))]\!] \\
&= [\![(\langle a \rangle \psi, a.Q_a | a.P)]\!] \\
&= [\![(\langle a \rangle \psi, Q | a.P)]\!] \\
&= [\![(\Pi(a.P)\langle a \rangle \psi, Q)]\!] \\
&= [\![(\Pi(a.P + N)(\langle a \rangle \psi), Q)]\!]
\end{aligned}$$

□

5.4 Read Operator

In Definition 5.1, we defined two separate prefix operators, pref and read. At least one pref was needed to make an action active, and beyond that

was immaterial. Thus, in our semantics, the composition of a pref with a pref yielded the same process as composition of a pref with a read. In certain cases, though, we may count the number of processes actively performing an action, *e.g.*, to model collisions between processes. We can do this with an *active* (count-dependent) read operator, which may produce different effects based on this number. This requires additional process extensions and quotienting rules.

5.4.1 Effect on Processes

On the process side, this affects the prefix operators. We define the change to the process algebra as follows:

Definition 5.3 (Active read operator)**.** *The addition of the active* read *operator changes E in Definition 5.1 as shown below.*

$$E ::= A^c.P \mid A.P + A^0.P + \cdots + A^{-n}.P$$

□

The change for the pref operator is straightforward: we simply provide the count, which is nonnegative. We also define c_p to refer to the count of a prefix process.

An active read process depends on the value of the count, which we model by annotating its actions with counts $c \leq 1$. An initial active read process is not, despite our name for it, active, *i.e.*, it does not have a response on a 1 count, but active read processes formed by composition with pref processes are active in this sense, as well; thus, c_p may be 0 or 1 for an active read process. For practical purposes, we disallow the presence of multiple active read processes on the same action. Additionally, we define n_p for prefix processes, which is n for a process *ending* with $a^{-n}.P_{a^{-n}}$. Then, the semantics for composing prefix processes is as follows:

- If we compose $a^{c_p}.P_a$ and $a^{c_q}.Q_a$; the processes synchronize on action a. The new count is the sum of the two counts, $c = c_p + c_q$, and we compose the attached processes, resulting in $a^c.(P_a|Q_a)$.
- Letting $c = c_p, n = n_q$, if we compose $a^c.P_a$ and $a.Q_{a^1} + a^0.Q_{a^0} + \cdots + a^{-n}.Q_{a^{-n}}$, we get:
  - if $c \leq n$, $a.(P_a|Q_{a^{1-c}}) + \cdots + a^{c-n}.(P_a|Q_{a^{-n}})$;

– if $c > n$, $a.(P_a | Q_{a^{-n}})$.

Note that the actual (positive) value of the count on a pref process only matters if it will be composed with an active read process, as otherwise all positive values are equivalent. Thus, when an active read process is merged with a pref process and only one possibility remains, we convert it into a pref process with its count at 1.

Effect on Quotienting

We can derive $\langle a^n \rangle_e \psi$ and $[a^n]_e \psi$ operators, for *exact* modalities, as follows:

$$\langle a^n \rangle_e \psi = \langle a^n \rangle \psi \wedge [a^{n+1}] \mathsf{ff}$$
$$[a^n]_e \psi = [a^n] \psi \vee \langle a^{n+1} \rangle \mathsf{tt}$$

These operators will be useful in writing clearer formulae for quotienting. Also, we have the following inverse relations:

$$\langle a^n \rangle \psi = \langle a^n \rangle_e \psi \vee \langle a^{n+1} \rangle \psi = \bigvee_{i \geq n} \langle a^i \rangle_e \psi$$
$$[a^n] \psi = [a^n]_e \psi \wedge [a^{n+1}] \psi = \bigwedge_{i \geq n} [a^i]_e \psi$$

The interpretation of Rules 11-12 changes to reflect that the counts are no longer limited to 0 and 1 and account for the exact a^n modalities. We add Rules 19-20 for the active read process. The new rules are seen in Table 5.3.

Proof. (Additional rules for Theorem 5.1)
Rule 19 ($P = a.P_{-1} + a^0.P_0 + \cdots + a^{-n}.P_n$, $\psi = \langle a^l \rangle \psi'$): Here, we assume that P at least responds to a and a^0. For a fixed Q, P becomes equivalent to a pref process, so we can reuse the steps from the proofs for Rules 11 and 13. Then, for the *diamond* case, the other values are vacuously equal to 0. There

Table 5.3: Additional Quotienting Rules

19.	$\Pi(a.P_{-1} + \cdots + a^{-n}.P_n)(\langle a^l\rangle\psi)$	$= \langle a^{l-1}\rangle_e\Pi(P_{-1})(\psi) \vee$
		$\vdots$
		$\vee \langle a^{l+n-1}\rangle_e\Pi(P_{n-1})(\psi)$
		$\vee \langle a^{l+n}\rangle\Pi(P_n)(\psi)$
20.	$\Pi(a.P_{-1} + \cdots + a^{-n}.P_n)([a^l]\psi)$	$= [a^{l-1}]_e\Pi(P_{-1})(\psi) \wedge$
		$\vdots$
		$\wedge [a^{l+n-1}]_e\Pi(P_{n-1})(\psi)$
		$\wedge [a^{l+n}]\Pi(P_n)(\psi)$
		$(c_p = 1 \text{ and } n = n_p \geq 0)$

exists $j \in \{-1..n\}$ such that $a.P_j$ is an equivalent process.

$$
\begin{aligned}
[\![(\langle a^l\rangle\psi, Q|P)]\!] &= [\![(\langle a^l\rangle\psi, Q|a.P_j)]\!] \\
&= [\![(\Pi(a.P_j)(\langle a^l\rangle\psi), Q)]\!] \\
&= [\![(\langle a^0\rangle\Pi(P_j)(\psi), Q)]\!] \\
&= \bigsqcup_{i=-1..n-1} [\![(\langle a^{l+i}\rangle_e\Pi(P_{a^{-i}})(\psi), Q)]\!] \sqcup [\![(\langle a^{l+n}\rangle\Pi(P_{a^{-n}})(\psi), Q)]\!] \\
&= [\![(\bigvee_{i=-1..n-1} \langle a^{l+i}\rangle_e\Pi(P_{a^{-i}})(\psi) \vee \langle a^{l+n}\rangle\Pi(P_{a^{-n}})(\psi), Q)]\!] \\
&= [\![(\Pi(P)(\langle a^l\rangle\psi), Q)]\!]
\end{aligned}
$$

In the *box* case for Rule 20, the proof is similar. □

5.5 Case Studies

We looked at two examples, the ECo-MAC protocol [ZBA10] and Choice Coordination [NM10], both of which have also been modeled with the PRISM Model Checker [KNP11]. We model both of these examples by using an ordered action interpretation, with differently named actions used only for synchronization of the composed processes.

5.5.1 Rabin's Choice Coordination Problem Encoding

In the choice coordination problem [NM10], we have some number of tourists who want to choose between two locations, without communicating directly with each other, so that, eventually, all tourists enter the same location. This is accomplished with a noticeboard at each location that the tourists update, according to predetermined rules, and the tourists separately keeping track of a value in personal notepads. We model the state of the two locations with a single process, and each tourist as her own process. Storing each noticeboard and tourist's value as a number, while a straightforward encoding, would make this an infinite-state system; therefore, we adapt the idea in [NM10] to make this a finite-state system. The key idea is to observe that the difference between the two noticeboards will never exceed one tier (the difference may be 1, 2, or 3 in this case), and termination is assured once the values are in the same tier, but differ by 1. Meanwhile, each tourist's value is essentially one of four possibilities: equal to one tier, but smaller (or greater) than the other; equal to both; or smaller than both. Additionally, we need to encode the next destination of each tourist; the total number of states is finite.

Process Encoding

Aside from visiting the left or the right location next, we encode the relationship between the tourists' own value with the noticeboard's, and update it not only when the tourist reaches the next location, but also should the noticeboard's value change. While the original system nondeterministically selects the next tourist to advance to her next location, we have a probabilistic choice for each tourist on each time step. We also prevent tourists from reaching their next location if they currently have a higher-tier value; this would not occur in the worst case of the original model with nondeterminism. There are process states corresponding to each location, which are mostly symmetric between tourists next headed for the right and left locations, so we will describe one direction:

- Locations begin with equality, eq_w:
  - A tourist may be an updater ($t_r^{u_1}$), with her own value equal to the locations.
  - If the tourist has a lower value, this value is expired ($t_r^{e_1}$).

When the location is eq_w, each tourist is in the set $\{t_r^{u_1}, t_l^{u_1}, t_r^{e_1}, t_l^{e_1}\}$, with at least one equal to each of $t_r^{u_1}$ and $t_l^{u_1}$ in a fully instantiated case.

- On an update, the location changes to a *difference*-2 state, d_2^w. The probabilistic choice at d_2^w is a key element of the coordination strategy.
  - If a tourist has a temporarily higher value than the next location, she is in a waiting state (t_r^w).
  - A tourist can again be an updater ($t_r^{u_2}$).
  - A tourist can still be expired ($t_r^{e_2}$ and $t_r^{e_3}$), with different states depending on which location is currently greater.

  The tourists actually encode which location has the higher value in this case. When the left one is higher, each tourist is in the set $\{t_r^w, t_r^{u_2}, t_l^{e_2}, t_r^{e_3}\}$, with at least one equal to each of t_r^w and $t_r^{u_2}$ in a fully instantiated case. When the right one is higher, we switch the rs and ls in the above.

- If the locations do not return to an equality state, they change to a *difference*-1 state, d_1, at which point there will be just one more update, to mark the location with the lower value with "Here".
  - All the tourists become potential finishers ($t_r^{f_1}$, $t_r^{f_2}$, $t_r^{f_3}$), along with the number of visits they still have to make to write "Here".
  - Once someone does, there are two done states, where the tourists just have to visit the set location to enter ($t_r^{d_1}$ and $t_r^{d_2}$).

  In the finishing case, when the right one will be it, the tourists are in the set $\{t_r^{f_1}, t_l^{f_2}, t_r^{f_3}\}$, with at least one tourist being $t_r^{f_1}$ in the fully instantiated case. In the done case, they are in the set $\{0, t_r^{d_1}, t_l^{d_2}\}$.

Example Property

A simple property here is that all the tourists eventually enter the same location, and thus there are no actions with positive counts:

$$\psi_l =_\mu [-]_u \psi_l.$$

Given the property, we can then quotient out the location and arbitrarily many tourists. Multiple tourists may move simultaneously in this formulation, which leads to a quickly growing number of formulae. However, it is still true that at the end of a turn, when all eligible tourists make a probabilistic choice, each of the tourists will be in at most four different states, so we could write down something like a recurrence relation for how the system evolves.

5.5.2 ECo-MAC Encoding

The ECo-MAC protocol is a protocol for wireless communications. Here, we model a simplified version of its backoff procedure [ZBA10], one that makes it similar to a Synchronous Leader Election Protocol [IR81] in nature.

In order to model this protocol, we use the active read operator. We model two kinds of processes, a single receiver and arbitrarily many senders. The simplified example covers a loop over contention stages, each involving several time steps.

Process Encoding

The receiver waits for a *request* r and reads the count of rs to return either *clear* to send, c, or *jam*, j. We encode the receiver as follows:

$$rec = r^0.c.rec + r^{-1}.j.rec.$$

The senders probabilistically decide to try sending a request after some number of time steps (in our simplified example, there are two choices). Sending a request is complementary to the receiver, and encoded as follows:

$$req = r.(c^0.0 + j^0.send).$$

The sender's probabilistic choice, when between 2 steps, is encoded as follows:

$$send = p.(req +_{0.5} (c^0.send + j^0.send + u.req)).$$

Example Property

Finally, we encode the property that all of the senders eventually receive a *clear* (*i.e.*, no action has a positive count):

$$\psi_c =_\mu [-]_u \psi_c.$$

Let us refer to $\psi_{c,[send^n,rec]}$ as $loop^n$. For $n \geq 2$, we have:

$$\begin{aligned} loop^n =& [\langle p^0\rangle(\langle r^2\rangle\langle j^0\rangle loop^n \vee \langle r\rangle_e\langle c^0\rangle loop^{n-1} \vee (\langle r^0\rangle_e \mathsf{tt} \wedge \langle u^0\rangle\langle r^0\rangle\langle j^0\rangle loop^n)) \\ &+_{\lambda_1} \langle p^0\rangle(\langle r\rangle\langle j^0\rangle loop^n \vee \langle r^0\rangle_e\langle c^0\rangle loop^{n-1})] \\ &+_{\lambda_2} \langle p^0\rangle\langle r^0\rangle\langle j^0\rangle loop^n. \end{aligned}$$

Here, $p_1 = \lambda_1 \cdot \lambda_2$ corresponds to the probability that a *jam* occurs in the first step, $p_2 = (1 - \lambda_1) \cdot \lambda_2$ to exactly one sender making a request in the first step, and $p_3 = (1 - \lambda_2)$ to no requests in the first step, which, for $n \geq 2$, precludes a *clear* in the second step. For this specific case, we can write down a recurrence relation for p_i^n:

$$\begin{aligned} p^2 &= (0.25, 0.5, 0.25), \\ p_1^n &= p_1^{n-1} + 0.5 \cdot p_2^{n-1} = 1 - (n+1)0.5^n, \\ p_2^n &= 0.5 \cdot p_2^{n-1} + 0.5 \cdot p_3^{n-1} = n \cdot 0.5^n, \\ p_3^n &= 0.5 \cdot p_3^{n-1} = 0.5^n. \end{aligned}$$

When quotienting this in our current implementation, we distinguish between the cases for the top part based on the number of processes contributing to a collision. Additional refinement would allow us to collapse all of those into a single formula in a distribution as shown above.

5.6 Conclusion

This chapter suggests that partial model checking may be practical in the probabilistic domain, although our results on case studies were more theoretical than experimental. In particular, even parameterized partial model checking [BR06] is plausible, with an ideal outcome being a formula that reaches a fixed point *except* for the values of λ on the $+_\lambda$ operators, and it may be extended to a wider class of problems via techniques such as symmetry reduction [CTV06]. Additionally, the paradigm with the active read and counting, which we used in the ECo-MAC case study for collisions, extends nicely into the definitions of PLTSs and quotienting in pLμ, and we are not aware of a similar paradigm elsewhere. Other extensions of pLμ, to add independent product or truncated sum (and their respective duals), appear straightforward to support in quotienting, as well.

Chapter 6

Conclusion

In non-probabilistic systems, there were two paradigms for time: linear and branching. Essentially, this reflected whether nondeterministic choices were presumed to be made before or after a property was to be verified. With the introduction of probabilistic choices, the nature of branching time becomes substantially more ambiguous. Nondeterministic choices have been typically resolved before the probabilistic ones, which we have called linear nondeterminism. Meanwhile, branching nondeterminism has more commonly appeared implicitly in specialized systems, *e.g.*, via recursion or nesting. An RPLTS is explicitly a probabilistic branching-time system, but without linear nondeterminism.

GPL was a creative extension of μ-calculus, being a logic over what we have called probabilistic branching time. Branching nondeterminism is particularly interesting in the case of entanglement, and we expect novel applications for entangled formulae in the future. The effect of linear nondeterminism on decidability, as we find in Chapter 3, is that the properties must be separable. Intuitively, this has to do with the difference in commitment between probabilistic and linear-nondeterministic choices. For the latter, the choice is made when a state is reached and *it is based on a property*: the property affects the choice, so it must be unique. This is consistent with the results for RMDPs and BMDPs [EY15], as extensions of RMCs and BPs. Additionally, we can allow both maximizing and minimizing choices at the same time in a system. One way to attempt this, in a way that retains the d-tree semantics, is to partition actions into minimizing and maximizing sets. Then, $\mathsf{Pr}^{\min}$ and $\mathsf{Pr}^{\max}$ would revert to a single Pr operator, but we would lose the ability to negate a formula; negation could be achieved in a dual PLTS where the minimizing and maximizing action sets were swapped. Similarly, if we were to designate certain actions as purely probabilistic, we could guarantee dependency graph construction for some non-separable formulae,

i.e., if at some point in the construction we would otherwise give up, but all of the actions were part of this purely probabilistic set, we could apply the GPL model checking algorithm.

In Chapter 4, we extended probabilistic logic programming to support probabilistic model checking. The FED data structure builds on tabling to support probabilistic branching time, and thus supports the model checking of logics such as PCTL [HJ94], PCTL* [Bai98], GPL [CIN05], and separable XPL [GR16]. The FED construction is also oblivious on whether a linear-nondeterministic choice is minimizing or maximizing, so allowing both in one system and property may entail no extra cost. Curiously, however, while the model checking of bounded-time PCTL properties could ostensibly have been done via PLP with the existing approaches prior to our contribution, this would be quite inefficient for large time bounds, and neither do we address this case. Indeed, even with [HJ94]'s PCTL model checking algorithm, it is true that PCTL properties with large time bounds lead to a more difficult computation than for similar unbounded properties, even though they are reduced to simple reachability, so further exploration of handling bounded time properties via logic programming is warranted. Meanwhile, in this thesis, we focused instead on branching time and fixed-point properties.

In Chapter 5, we extended partial model checking [And95, BR06] to the probabilistic domain. In order to incorporate the quotienting of probabilistic choices into the logical formula, we essentially allow a formula to be a convex combination of formulae. With simple examples, we can reach what looks like a fixed point, if we were to remove the probabilities in the convex combination. On the model side, we described a simple process algebra motivated by CCS [Mil89, JYL01], and introduced a special *active* read operator to enhance the ways different processes may synchronize. Our refinement rules are currently basic, and could likely be extended a la [BR06], so that we could identify more formulae as equivalent. Overall, this chapter may be viewed as a collection of interesting and potentially useful ideas.

Bibliography

[ACM11] Rajeev Alur, Swarat Chaudhuri, and P. Madhusudan. Software model checking using languages of nested trees. *ACM Trans. Program. Lang. Syst.*, 33(5):15:1–15:45, November 2011.

[AH99] Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Form. Methods Syst. Des.*, 15(1):7–48, July 1999.

[And95] Henrik Reif Andersen. Partial model checking (extended abstract). In *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 398–407. IEEE Computer Society Press, 1995.

[Bai98] Christel Baier. On algorithmic verification methods for probabilistic systems. Habilitation thesis, Fakultät für Mathematik & Informatik, Universität Mannheim, 1998.

[BHHK03] Christel Baier, Boudewijn Haverkort, Holger Hermanns, and Joost-Pieter Katoen. Model-checking algorithms for continuous-time Markov chains. *IEEE Transactions on Software Engineering*, 29(7):2003, 2003.

[BR06] Samik Basu and C. R. Ramakrishnan. Compositional analysis for verification of parameterized systems. *Theoretical Computer Science*, 354(2):211–229, 2006.

[CDK12] Taolue Chen, Klaus Dräger, and Stefan Kiefer. Model checking stochastic branching processes. In *Proceedings of the Mathematical Foundations of Computer Science 2012: 37th International Symposium, MFCS 2012, Bratislava, Slovakia, August 27-31, 2012*, pages 271–282, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8:244–263, April 1986.

[CG09] Henning Christiansen and John P. Gallagher. Non-discriminating arguments and their uses. In *Logic Programming*, volume 5649 of *Lecture Notes in Computer Science*, pages 55–69. Springer Berlin Heidelberg, 2009.

[CI00] Rance Cleaveland and S Purushothaman Iyer. Branching time probabilistic model checking. In *ICALP Workshops*, volume 8, pages 487–500. Citeseer, 2000.

[CIN05] Rance Cleaveland, S. Purushothaman Iyer, and Murali Narasimha. Probabilistic temporal logics via the modal mu-calculus. *Theoretical Computer Science*, 342(2-3):316–350, 2005.

[CKP15] Pablo Castro, Cecilia Kilmurray, and Nir Piterman. Tractable Probabilistic mu-Calculus That Expresses Probabilistic Temporal Logics. In *32nd International Symposium on Theoretical Aspects of Computer Science (STACS 2015)*, volume 30 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 211–223, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[CTV06] Edmund M. Clarke, Muralidhar Talupur, and Helmut Veith. Environment abstraction for parameterized verification. In *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*, volume 3855 of *Lecture Notes in Computer Science*, pages 126–141. Springer, 2006.

[DGJP02] Jose Desharnais, Vineet Gupta, Radha Jagadeesan, and Prakash Panangaden. Weak bisimulation is sound and complete for PCTL*. In *CONCUR 2002 Concurrency Theory*, volume 2421 of *Lecture Notes in Computer Science*, pages 355–370. Springer Berlin Heidelberg, 2002.

[DP99] Giorgio Delzanno and Andreas Podelski. Model checking in CLP. In *TACAS*, pages 223–239, 1999.

[DRKT07] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog: A probabilistic Prolog and its application in link discovery. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 2462–2467, 2007.

[DRS00] Xiaoqun Du, C. R. Ramakrishnan, and Scott A. Smolka. Tabled resolution + constraints: A recipe for model checking real-time systems. In *IEEE Real Time Systems Symposium (RTSS)*, Orlando, Florida, November 2000.

[EH86] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. *J. ACM*, 33:151–178, January 1986.

[EKM04] Javier Esparza, Antonín Kucera, and Richard Mayr. Model checking probabilistic pushdown automata. In *LICS*, pages 12–21, 2004.

[ESY12a] Kousha Etessami, Alistair Stewart, and Mihalis Yannakakis. Polynomial time algorithms for branching Markov decision processes and probabilistic min(max) polynomial Bellman equations. In *Proceedings of the Automata, Languages, and Programming: 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Part I*, pages 314–326, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[ESY12b] Kousha Etessami, Alistair Stewart, and Mihalis Yannakakis. Polynomial time algorithms for multi-type branching processes and stochastic context-free grammars. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, STOC '12, pages 579–588, New York, NY, USA, 2012. ACM.

[ESY15] Kousha Etessami, Alistair Stewart, and Mihalis Yannakakis. Greatest fixed points of probabilistic min/max polynomial equations, and reachability for branching Markov decision processes. In *Proceedings of the Automata, Languages, and Programming: 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Part II*, pages 184–196, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[EY09] Kousha Etessami and Mihalis Yannakakis. Recursive Markov chains, stochastic grammars, and monotone systems of nonlinear equations. *J. ACM*, 56(1):1:1–1:66, February 2009.

[EY15] Kousha Etessami and Mihalis Yannakakis. Recursive Markov decision processes and recursive stochastic games. *J. ACM*, 62(2):11:1–11:69, May 2015.

[FGKP99] Nir Friedman, Lise Getoor, Daphne Koller, and Avi Pfeffer. Learning probabilistic relational models. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'99, pages 1300–1307, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[FL04] Berndt Farwer and Michael Leuschel. Model checking object Petri nets in Prolog. In *Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '04, pages 20–31, New York, NY, USA, 2004. ACM.

[GBM+07] Gopal Gupta, Ajay Bansal, Richard Min, Luke Simon, and Ajay Mallya. Coinductive logic programming and its applications. In *Logic Programming, 23rd International Conference, ICLP 2007, Porto, Portugal, September 8-13, 2007, Proceedings*, volume 4670 of *Lecture Notes in Computer Science*, pages 27–44. Springer, 2007.

[GJD10] Bernd Gutmann, Manfred Jaeger, and Luc De Raedt. Extending ProbLog with continuous distributions. In *Proceedings of the 20th International Conference on Inductive Logic Programming (ILP–10)*, Firenze, Italy, 2010.

[GP97] Gopal Gupta and Enrico Pontelli. A constraint-based approach for specification and verification of real-time systems. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97), December 3-5, 1997, San Francisco, CA, USA*, pages 230–239. IEEE Computer Society, 1997.

[GR16] A. Gorlin and C. R. Ramakrishnan. XPL: An extended probabilistic logic for probabilistic transition systems. *ArXiv e-prints*, April 2016.

[GRS12] Andrey Gorlin, C. R. Ramakrishnan, and Scott A. Smolka. Model checking with probabilistic tabled logic programming. *TPLP*, 12(4-5):681–700, 2012.

[GT07] Lise Getoor and Ben Taskar. *Introduction to Statistical Relational Learning*. The MIT Press, 2007.

[GTK+11] Bernd Gutmann, Ingo Thon, Angelika Kimmig, Maurice Bruynooghe, and Luc De Raedt. The magic of logical inference in probabilistic programming. *TPLP*, 11(4–5):663–680, 2011.

[HJ94] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

[HKQ11] H. Hansen, M. Kwiatkowska, and H. Qu. Partial order reduction for model checking Markov decision processes under unconditional fairness. In *Eighth International Conference on Quantitative Evaluation of Systems (QEST)*, pages 203–212, Sept 2011.

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, New York, NY, USA, 2004.

[IN96] Purush Iyer and Murali Narasimha. "Almost always" and "definitely sometime" are not enough: probabilistic quantifiers and probabilistic model-checking. Technical report, North Carolina State University at Raleigh, 1996.

[IR81] Alon Itai and Michael Rodeh. Symmetry breaking in distributive networks. In *FOCS*, pages 150–158, 1981.

[IRR12] Muhammad Asiful Islam, C. R. Ramakrishnan, and I. V. Ramakrishnan. Inference in probabilistic logic programs with continuous random variables. *Theory and Practice of Logic Programming*, 12(4–5):505–523, 2012.

[JYL01] Bengt Jonsson, Wang Yi, and Kim G Larsen. Probabilistic extensions of process algebras. *Handbook of process algebra*, pages 685–710, 2001.

[KDR01a] Kristian Kersting and Luc De Raedt. Adaptive Bayesian logic programs. In *Proceedings of the Inductive Logic Programming, 11th International Conference, ILP 2001, Strasbourg, France, September 9-11, 2001*, volume 2157 of *Lecture Notes in Computer Science*, pages 104–117. Springer, 2001.

[KDR01b] Kristian Kersting and Luc De Raedt. Bayesian logic programs. *CoRR*, cs.AI/0111058, 2001.

[KEM06] Antonín Kucera, Javier Esparza, and Richard Mayr. Model checking probabilistic pushdown automata. *Logical Methods in Computer Science*, 2(1), 2006.

[KLE07] Stefan Kiefer, Michael Luttenberger, and Javier Esparza. On the convergence of Newton's method for monotone systems of polynomial equations. In *STOC*, pages 217–226, 2007.

[KNP11] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *23rd CAV*, volume 6806 of *LNCS*, pages 585–591, 2011.

[Koz83] Dexter Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[LSWZ15] Wanwei Liu, Lei Song, Ji Wang, and Lijun Zhang. A simple probabilistic extension of modal mu-calculus. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 882–888. AAAI Press, 2015.

[Mar98] Donald A. Martin. The determinacy of Blackwell games. *The Journal of Symbolic Logic*, 63(4):1565–1581, 1998.

[Mil89] Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[Mio11] Matteo Mio. Probabilistic modal mu-calculus with independent product. In *Proceedings of the Foundations of Software Science and Computational Structures: 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken,*

Germany, March 26–April 3, 2011, pages 290–304, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[Mio12] Matteo Mio. Game semantics for probabilistic modal mu-calculi. PhD thesis, The University of Edinburgh, 2012.

[MM15] Henryk Michalewski and Matteo Mio. On the problem of computing the probability of regular sets of trees. In Prahladh Harsha and G. Ramalingam, editors, *35th IARCS Annual Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2015, December 16-18, 2015, Bangalore, India*, volume 45 of *LIPIcs*, pages 489–502. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[MP99] Supratik Mukhopadhyay and Andreas Podelski. Beyond region graphs: Symbolic forward analysis of timed automata. In *FSTTCS*, pages 232–244, 1999.

[MP00] Supratik Mukhopadhyay and Andreas Podelski. Model checking for timed logic processes. In *Computational Logic*, pages 598–612, 2000.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Parts I and II. *Information and Computation*, 100(1):1–77, 1992.

[MS13a] Matteo Mio and Alex Simpson. Łukasiewicz mu-calculus. In *Proceedings Workshop on Fixed Points in Computer Science, FICS 2013, Turino, Italy, September 1st, 2013*, volume 126 of *EPTCS*, pages 87–104, 2013.

[MS13b] Matteo Mio and Alex Simpson. *A Proof System for Compositional Verification of Probabilistic Concurrent Processes*, pages 161–176. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[Mug96] Stephen Muggleton. Stochastic logic programs. In *New Generation Computing*. Academic Press, 1996.

[NBKJ10] Per Närman, Markus Buschle, Johan König, and Pontus Johnson. Hybrid probabilistic relational models for system quality

analysis. In *Proceedings of the 14th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2010, Vitória, Brazil, 25-29 October 2010*, pages 57–66. IEEE Computer Society, 2010.

[NM95] Ulf Nilsson and Jan Maluszynski. *Logic, Programming, and PROLOG*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 1995.

[NM10] Ukachukwu Ndukwu and AK McIver. An expectation transformer approach to predicate abstraction and data independence for probabilistic programs. In *Proc. 8th Workshop on Quantitative Aspects of Programming Languages (QAPL'10)*, 2010.

[Poo08] David Poole. The independent choice logic and beyond. In *Probabilistic Inductive Logic Programming*, volume 4911 of *Lecture Notes in Computer Science*, pages 222–243. Springer Berlin Heidelberg, 2008.

[PRR02] Giridhar Pemmasani, C. R. Ramakrishnan, and I. V. Ramakrishnan. Efficient model checking of real time systems using tabled logic programming and constraints. In *International Conference on Logic Programming (ICLP)*, LNCS. Springer, 2002.

[Ram01] C. R. Ramakrishnan. A model checker for value-passing mucalculus using logic programming. In *PADL*, volume 1990 of *LNCS*, pages 1–13. Springer, 2001.

[RD06] Matthew Richardson and Pedro Domingos. Markov logic networks. *Machine Learning*, 62(1-2):107–136, February 2006.

[RRR+97] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Terrance Swift, and David S. Warren. Efficient model checking using tabled resolution. In *CAV*, volume 1254 of *LNCS*, pages 143–154. Springer, 1997.

[RRS+00] C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Yifei Dong, Xiaoqun Du, Abhik Roychoudhury, and V. N. Venkatakrishnan. XMC: A logic-programming-based verification toolset. In *Twelfth International Conference on Computer*

Aided Verification (CAV), volume 1855 of *Lecture Notes in Computer Science*, pages 576–580, Chicago, Illinois, July 2000. Springer.

[RS10a] Fabrizio Riguzzi and Terrance Swift. An extended semantics for logic programs with annotated disjunctions and its efficient implementation. In *Italian Conference on Computational Logic*, volume 598 of *CEUR Workshop Proceedings*, 2010.

[RS10b] Fabrizio Riguzzi and Terrance Swift. Tabling and answer subsumption for reasoning on logic programs with annotated disjunctions. In *Technical Communications of the International Conference on Logic Programming*, pages 162–171, 2010.

[RS11] Fabrizio Riguzzi and Terrance Swift. The PITA system: Tabling and answer subsumption for reasoning under uncertainty. *TPLP*, 11(4–5):433–449, 2011.

[SCPQC03] Vítor Santos Costa, David Page, Maleeha Qazi, and James Cussens. CLP($\mathcal{BN}$): Constraint logic programming for probabilistic knowledge. In *Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence (UAI03)*, pages 517–524, Acapulco, Mexico, August 2003.

[Seg95] Roberto Segala. A compositional trace-based semantics for probabilistic automata. In *Proceedings of the CONCUR '95: Concurrency Theory, 6th International Conference, Philadelphia, PA, USA, August 21-24, 1995*, volume 962 of *Lecture Notes in Computer Science*, pages 234–248. Springer, 1995.

[SK97] Taisuke Sato and Yoshitaka Kameya. PRISM: a language for symbolic-statistical modeling. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 1330–1335, 1997.

[SM14] Taisuke Sato and Philipp Meyer. Infinite probability computation by cyclic explanation graphs. *Theory and Practice of Logic Programming*, 14(06):909–937, 2014.

[Son04] Arvind Soni. Probabilistic and nondeterministic systems. Masters thesis, North Carolina State University, 2004.

[SRS08] Anu Singh, C. R. Ramakrishnan, and Scott A. Smolka. A process calculus for mobile ad hoc networks. In *10th International Conference on Coordination Models and Languages (COORDINATION)*, volume 5052 of *LNCS*, pages 296–314. Springer, 2008.

[SSR03] Beata Sarna-Starosta and C. R. Ramakrishnan. Constraint-based model checking of data-independent systems. In *International Conference on Formal Engineering Methods (ICFEM)*, volume 2885 of *Lecture Notes in Computer Science*, pages 579–598. Springer, 2003.

[ST05] Roberto Segala and Andrea Turrini. Comparative analysis of bisimulation relations on alternating and non-alternating probabilistic models. In *Second International Conference on the Quantitative Evaluaiton of Systems (QEST 2005), 19-22 September 2005, Torino, Italy*, pages 44–53. IEEE Computer Society, 2005.

[Ste09] William J. Stewart. *Probability, Markov Chains, Queues, and Simulation: The Mathematical Basis of Performance Modeling*, chapter Markov Chains, pages 191–382. Princeton University Press, Princeton, NJ, USA, 2009.

[Ste15] Alistair Stewart. Efficient algorithms for infinite-state recursive stochastic models and newton's method. PhD thesis, The University of Edinburgh, 2015.

[SW+12] Terrance Swift, David S. Warren, et al. The XSB logic programming system, Version 3.3, 2012. `http://xsb.sourceforge.net`.

[VDB09] Joost Vennekens, Marc Denecker, and Maurice Bruynooghe. CP-logic: A language of causal probabilistic events and its relation to logic programming. *TPLP*, 9(3):245–308, 2009.

[VVB04] Joost Vennekens, Sofie Verbaeten, and Maurice Bruynooghe. Logic programs with annotated disjunctions. In *ICLP*, pages 431–445, 2004.

[WD08] Jue Wang and Pedro Domingos. Hybrid Markov logic networks. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2*, AAAI'08, pages 1106–1111. AAAI Press, 2008.

[WE07] Dominik Wojtczak and Kousha Etessami. PReMo: An analyzer for probabilistic recursive models. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4424 of *Lecture Notes in Computer Science*, pages 66–71. Springer, 2007.

[YRS04] Ping Yang, C. R. Ramakrishnan, and Scott A. Smolka. A logical encoding of the pi-calculus: Model checking mobile processes using tabled resolution. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(1):38–66, 2004.

[ZBA10] Hafedh Zayani, Kamel Barkaoui, and Rahma Ben Ayed. Probabilistic verification and evaluation of backoff procedure of the WSN ECo-MAC protocol. *International Journal of Wireless & Mobile Networks*, 2(2):156–170, 2010.